Compiler Support for Near Data Computing

Mahmut Taylan Kandemir
Penn State University
USA
mtk2@cse.psu.edu

Jihyun Ryoo
Penn State University
USA
jihyunryoo@gmail.com

Xulong Tang
University of Pittsburgh
USA
tax6@pitt.edu

Mustafa Karakoy
TUBITAK-BILGEM
Turkey
m.karakoy@yahoo.co.uk

Abstract

Recent works from both hardware and software domains offer various optimizations that try to take advantage of near data computing (NDC) opportunities. While the results from these works indicate performance improvements of various magnitudes, the existing literature lacks a detailed quantification of the potential of NDC and analysis of compiler optimizations on tapping into that potential. This paper first presents an analysis of the NDC potential when executing multithreaded applications on manycore platforms. It then presents two compiler schemes designed to take advantage of NDC. The first of these schemes try to increase the amount of computation that can be performed in a hardware component, whereas the second compiler strategy strikes a balance between optimizing NDC and exploiting data reuse, by being more selective on when to perform NDC (even if the opportunity presents itself) and how. The collected experimental results on a 5×5 manycore system reveal that our first and second compiler schemes improve the overall performance of our multithreaded applications by, respectively, 22.5% and 25.2%, on average. Furthermore, these two compiler schemes are only 6.8% and 4.1% worse than an oracle scheme that makes the best near data computing decisions for each and every computation.

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

Keywords: near-data computing, data locality, code transformation, manycore architectures

1 Introduction

With the end of Dennard scaling, the gap between the computational demand exhibited by large-scale datasets and computational capabilities that can cater to that demand is widening. As a result, the conventional computational paradigm based on the clear separation of storage location (cache/memory/SSD/disk) and compute location (cores) that has prevailed for more than six decades now is becoming increasingly less successful in meeting the needs of emerging big data computations. Lacking a holistic perspective on how this widening gap between what is desired and what can actually be delivered, application, computer architecture, compiler, and runtime systems/OS communities each are proposing their own various solutions to the problem.

One of the research directions promoted in recent years is based on the idea of *Near Data Computing* (NDC), which brings computation closer to the data, as opposed to the more conventional paradigm which brings data to computation. A typical example of NDC is processing-in-memory (PIM) where simple operations (e.g., transpose operation or addition operation) are computed inside memory array [56]. One can roughly divide the existing body of NDC-related studies into two broad categories. The first category includes the studies that try to reorder computations in an attempt to reduce the distance between the memory location that holds data and the core that requests the data. In such approaches, the computation is still executed by conventional compute units (cores) in the architecture, and as such, these approaches can just be considered as a variant of conventional data locality optimizations [7, 14, 27, 32, 36, 37, 40, 54, 61–63]. In contrast, the works in the second category enhance an underlying (baseline) architecture with additional (custom) compute units to execute computations near data. Example locations augmented with compute units include DRAM [3, 5, 24, 25, 56], memory controllers [15, 16, 28, 52], cache controllers [2, 46, 65], and on-chip network controllers [19, 47, 60]. Clearly, one of the main questions that the works in this second category need to tackle is how frequently all of the data needed by a given computation happen to reside at the same time in the same hardware component of interest (e.g., cache controller or memory controller). For example, if a computation c needs data elements A and B, these two data elements should meet in the same hardware component at the same time, so that c can be performed in that component (assuming that the component in question has been equipped with proper compute units to perform c). More frequently than not, one of the required data elements, say A, arrives in a hardware component but the other, say B, has not arrived yet. A critical question is then when B

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441600

will arrive at the same component, or whether it will arrive at all. A related question is how long we can tolerate to wait for B to arrive, beyond which further waiting does not make sense. Yet, another critical question is what can be done to make A wait less for B – or better, is it possible to rearrange computations such that A and B meet "around the same time" in the target component? Unfortunately, in spite of a flurry of recent works in NDC, these critical questions have not been properly addressed in the literature. Going back to our simple example above that involves computation c and data elements A and B, one can make the following observations:

- Making sure that A and B arrive at the target component at around the same time is critical, mainly because i) if B is late, A will occupy resources till B arrives and ii) more A waits, lower the chances that exploiting NDC will lead to more efficient execution.
- When there are multiple potential hardware components where c can be performed (i.e., multiple locations A and B can potentially meet), selecting the most appropriate one will be important. Note that, at a particular time, A can be in one component, whereas B in another one. Depending on timing and nature of computation, it may be better to move A to B, or B to A, or to move both A and B to a third component, to perform computation.
- In a manycore/multithreaded execution scenario, the best component to perform a given computation near its data depends, in general, on the behavior of all threads that affect the movement of the data involved. In particular, any NDC decision made solely based on the viewpoint of one thread may not be globally optimal when all threads are considered together.
- It is not trivial to decide when to exploit NDC opportunities versus just going ahead with the conventional execution paradigm. In particular, there are cases where performing NDC *conflicts* with what a potential *data locality optimization* prefers. In such cases, performing the right tradeoff, depending on the context (mostly dictated by the degree of data reuse), can be critical.

Based on the observations above, focusing on manycore architectures and multithreaded CPU applications, this paper makes the following two **contributions**:

- It presents results from a detailed analysis of the NDC potential when executing multithreaded applications on manycore platforms. In particular, i) for a given (hardware component, computation) pair, it presents data measuring how long each involved data item is waited for, if the computation is to be performed in that component; ii) conversely, it also provides data measuring the amount of computation that can be performed in a component if the maximum waiting time is set to a specific value; and iii) the ideal hardware component to perform a given computation and the related performance improvements. Our detailed experimental analysis reveals that an *oracle scheme*, which i) makes the right "tradeoff" between NDC and data locality optimization and ii) selects the "ideal" hardware component to perform NDC, brings an average performance improvement of 29.3% across 20 multithreaded application programs tested.
- It presents two *compiler-based* NDC schemes designed to take advantage of NDC. The first scheme tries to increase the amount of computation that can be performed in a hardware component by bringing the operands needed for the computation close to one another in time. The collected experimental data reveals that the proposed approach brings about 22.5% performance improvement over the original application programs. Our second compiler strategy on the other hand strikes a balance between optimizing NDC and exploiting data reuse (cache locality), by being more selective on when to perform NDC (even if the opportunity presents itself) and how. Our experimental evaluation of this second strategy indicates around 25.2% average improvement over the original applications.

2 Assumptions on Architecture

In this work, we focus on a manycore architecture (shown in Figure 1) in which multiple nodes are connected to one another using a scalable on-chip network (also called network-on-chip (NoC)). Each node in this architecture can have one or more cores, a private L1 cache per core, and an L2 bank shared with other nodes. We assume static x-y routing as it is the most frequently used routing in NoC-based manycores. As for the management of data, we assume a static NUCA (non-uniform cache access) architecture for our discussion, where each cache block is assigned to one of the L2 banks and one of the memory banks in the system (based on its address). Therefore, a data access issued by a core in this architecture first checks the local L1 cache of the core, then (if it misses in the L1 cache) the L2 bank (determined by the address of the data), and then (if it misses in the L2 bank) the off-chip memory bank (again, determined by the address of the requested data). Note that this architectural template is similar to several commercial manycores [1, 21, 53].

To enable the NDC capability in this architecture, we leverage a similar design from prior work [48]. We consider near data computing in four hardware locations, namely, *link buffers/routers*, *cache controllers*, *memory controllers*, and *main memory* itself, as highlighted in Figure 1 using ⓐ, ⓑ, ⓒ, and ⓓ, respectively. Specifically, the load/store (LD/ST) unit is augmented with an offload table that tracks the offload operations, which are also called "pre-compute" instructions defined shortly. Each NDC ALU in the NDC architecture is also equipped with a service table. The service table is used to track the received NDC packages. The operations queued in the service table are processed in order. If the service table is full, the time-out mechanism will be triggered and

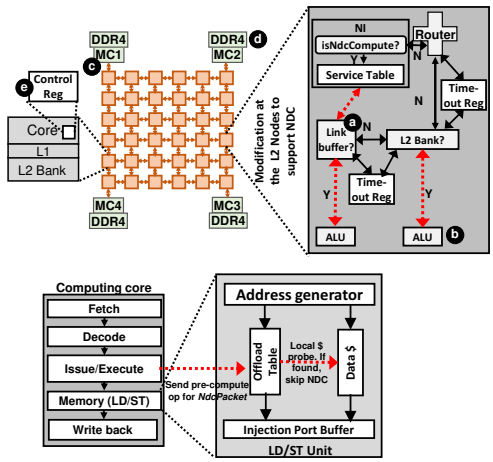

Figure 1. Target manycore architecture (one core per node). To enable the NDC capability, the LD/ST unit is enhanced with an offload table to track the offload instructions (i.e., pre-compute instructions). The NDC components (e.g., link buffer and L2 bank) are enhanced with ALU and time-out registers. The architecture also includes a control register in the CPU to control which NDC components are disabled.

the operation is eventually performed in the original location. Specifically, the time-out mechanism sends signal to the original core and updates an offload table there in the LD/ST unit, indicating that the computation will be performed in the original core.

Throughout our discussion, we use "+" for "op" but our proposed approach handles any arithmetic or logic operation (op) implemented in a given location of interest (i.e., any of the four locations mentioned above). In the case of near data computing in memory (in-memory computing), we assume that A op B is performed in memory if both A and B are currently residing in the same memory bank. In the case of cache controller, we assume a compute unit attached to each cache controller and can perform an operation in there within "arrival window" (to be defined shortly) if both the operands are available in the cache. Each (L2) cache controller in the system is assumed to be equipped with the same compute unit. In the case of link routers/buffers, on the other hand, we assume that a compute unit is attached to each router that can perform a computation there if both the operands are available in the corresponding link buffer. Similarly, we also assume that each memory controller in the system is equipped with a compute unit that can perform computations that need operands currently queued in the memory queue. Note that, while we assume that each memory bank is augmented with a compute unit that can perform select computations there, a computation will be performed in the memory bank only if the most updated values of both of the required operands are currently in that bank. Note also that if the offloaded instructions have register operands, they are also transferred to the location where the computation will be performed.

The NDC capabilities offered by this architecture can be exploited in multiple ways. First, in the most general case, before offloading, the LD/ST unit first probes if any of the operands is in the local cache. If so, the operation is performed within the core to take advantage of the locality in the local cache. Otherwise, an "NDC compute package" is formed and injected into the network port buffer. In the routers' network interface, the NDC compute package is checked whether the operands are available in the link buffer. If so, computation is performed by leveraging the ALUs attached to the link buffer. Otherwise, if the node contains the L2 home bank of both operands and the operands are available in the L2 bank, computation is performed at the L2 bank. Similarly, the NDC compute package is checked in the memory controller and the main memory, and is performed in the corresponding component when the operands are available there.

Alternatively, using a specialized "control register", shown as ⓔ in Figure 1, the specific components can be skipped as potential NDC locations. For example, we can indicate that NDC needs to be performed only in network router/link buffer, or only in link buffer or cache controller, etc. In such cases, the NDC compute package is directly sent to the target NDC location. In both the usage scenarios explained above, when an NDC compute package arrives at a target NDC location, the execution waits for the operands. Optionally, "time-out" registers (also shown in Figure 1) can be used to limit the amount of time to wait in each NDC component, before resorting to the default (non-NDC) computation. Figure 1 shows the time-out registers in link buffer/router and the L2 bank. Similarly, there are time-out registers in the memory controller and the main memory to bound the waiting time of operands. That is, once the first operand arrives at the destination, we can limit the time period it stays there before aborting NDC and performing the computation in a conventional fashion (i.e., in the original target core). Also, when the computation is performed near data, the CPU is signaled using a "CPU-feed" signal.

We want to emphasize that, such NDC-enabling architecture designs have been explored by prior work, and are not the main focus of this paper. We refer the reader to recent works [5, 9, 22, 28, 48] for further details. Instead, our main focus in this work is on the evaluation of the potential of NDC and exploring compiler optimizations that enhance NDC opportunities. Note that, in our approach, the compiler explicitly marks the instructions to be performed near data using one of the unused fields in the instruction format. In particular, we introduce a new instruction called "pre-compute" for the offloaded operations, which will be discussed later. The contents of this pre-compute instruction are mapped to an NDC compute package.

3 Applications and Setup

In this study, we evaluated application programs from two different benchmark suites, SPECOMP [10] and SPLASH-2 [8], to conduct our experimental evaluations. The input sizes of these benchmarks are increased from their original values to put more pressure on on-chip hardware resources (e.g., caches, network, and memory controllers). In our evaluations, the dataset sizes for the SPECOMP benchmarks vary between 754 MB and 4.2 GB, and those for the SPLASH-2 benchmarks range from 821 MB to 5.1 GB.

Since current commercial systems do not include compute units to implement the near data computing strategies evaluated in this work, we conduct a simulation-based study. Also, since we are interested in measuring the potential and limits of near data computing, we make an assumption where the cycle cost of performing the computation in our compute units (added to the baseline architecture) is the same as performing it on the original core (note however that NDC still cuts overall data access latencies significantly as it performs computations in locations close to data). Of course, if the result an operation that has been performed near data is required by a subsequent operation in a different place, the cost of transferring it there is included/modeled in the simulator. Also, all our presented results include the additional overheads brought by near data computing.

We enhanced a multicore/manycore simulator, GEM5 [12], to collect our results. GEM5 is a modular toolset for computer system architecture research, including system-level architecture as well as processor microarchitecture. Using GEM5, we modeled a manycore architecture whose main characteristics are given in Table 1. Note that our simulator models all the components of the NDC-enabling architecture discussed in Section 2. The proposed compiler support has been implemented using LLVM-9.0.1 [13].

Table 1. The simulated configuration.

Parameter	Default Value
Cores and Caches	Processor: two-issue OoO, SPARC processor
	Data/Instr. L1 Config.: 32 KB (per node), 64 byte lines, 2 ways, 2 cycle access latency
	L2 Config.: 512 KB (per node), 256 byte lines, 64 ways, cache line interleaved, 20 cycle access latency
On-Chip Network (NoC)	Size: 5 × 5 2D Mesh
	Delays/Routing: 16B links, 3-cycle pipeline, XY-routing
Memory System	Number of Memory Controllers: 4 [same as page size]
	Interleaving Granularity: 4KB
	Scheduling Policy: FR-FCFS
	Capacity: 32GB
	Device Parameters: Micron MT47H64M8 DDR2-800 4 banks/device, 16384 rows/bank, 512 columns/row
	Row Buffer Size: 4KB [same as page size] 4 active row buffers per DIMM
Computation/ Mapping	Thread count per core: 1
	Types of offloading: All arithmetic and logic operations

4 Quantification of NDC

This section first introduces the metrics used in our evaluations of NDC and then presents the detailed characterization results collected from the executions of our original application programs.

4.1 Evaluation Metrics

We start below by first introducing the three evaluation metrics considered in this work, focusing on a single operation and two operands (A + B) and a location of interest (referred to as `loc`) where the said operation is to be performed.[1]

• *Arrival Window:* This metric represents the period of time the first arriving operand waits for the second one. That is, it captures the difference in cycles between the arrivals of operands A and B at `loc`. Clearly, in the ideal case, we want the value of this metric to be 0, indicating that both the operands arrive at `loc` at "exactly" the same time. However, in many cases, A can tolerate some waiting time for B (and the resulting NDC can still be more efficient than performing the computation in the conventional way).

• *Breakeven Point:* The breakeven point refers to the time (arrival window size), which leads to a better result compared to the conventional execution. That is, if the breakeven point is 25 cycles, it means that, if the arrival window (the time the first operand waits for the second operand to arrive at the location of interest) is less than or equal to 25 cycles, near data computing will generate a "better result" than the conventional computing in the core. If however the arrival window is more than 25 cycles, it means the NDC will generate worse results than the conventional computing.

• *Performance Benefit:* This metric captures the relative improvement brought by an NDC optimization scheme (measured in *execution cycles* saved). Note that, the optimal decision either performs near data computation if the breakeven point is within the arrival window, or resorts to the conventional computation if the breakeven point is outside the arrival window.

Below, we present an experimental analysis of these metrics, and identify the potential of near data computing, when the original benchmark programs are executed *without* any near data computing-specific code optimization.

4.2 Arrival Window Analysis

Each graph in Figure 2 gives an analysis of arrival windows (in the form of cumulative distribution function (CDF) that is truncated to 50%) for different locations of interest (`loc`) and different application programs. Note that in these plots 500+ represents arrival window lengths that are larger than 500 cycles, and also includes the cases where the second operand *never* arrives at the location of interest (e.g., when the location of interest is the link buffer but the paths of the two operands do not intersect on the network).

[1] As will be discussed later, in this work, we consider four potential locations to perform NDC, namely, link buffers, cache controllers, memory controllers, and memory banks. These locations are selected mainly because they are natural station over the path of a data access.

It can be observed from the CDF plots in Figure 2 that, the arrival window lengths vary significantly depending on the benchmark and the specific location under consideration. For example, in *swim*, approximately 14.3% of the arrival windows at cache controller are less than or equal to 20 cycles, whereas, for the same benchmark, only 7.71% of the arrival times are less than 20 cycles when considering memory controller. Further, given a location (loc), the arrival windows of two benchmark programs can exhibit quite different behaviors. For instance, consider *applu* and *raytrace* and cache controller. In *applu*, around 26.7% of all arrival windows are less than or equal to 20 cycles, whereas in *raytrace* the corresponding fraction is 8.6%. Since arrival window puts an upper limit for the waiting time (our second metric defined above), we can conclude from the results in Figure 2 that different applications are expected to get different amounts of benefits from near data computing.

4.3 Breakeven Point Analysis

The first arriving operand has the option of leaving the location (loc) any time within the arrival window. A question at this point is how breakeven points compare against the arrival windows. Figure 3 plots the distribution of arrival windows and breakeven points when *averaged* over all 20 benchmark programs. It can be observed from these results that breakeven points are in general much lower than the arrival windows, meaning that, *if the early operand waits for the late operand, it would be waiting beyond the breakeven point,* and as a result, the performance would degrade. Although the graph in Figure 3 shows the results averaged over all benchmarks, we found that, in each of our benchmarks, the breakeven points were lower than the arrival windows.

4.4 Benefit Analysis

Figure 4 plots the performance benefits over the original case when the first operand that arrives at the location of interest uses different waiting strategies (times). The first bar corresponds to the performance benefits (a negative benefit value indicates an increase in execution time over the original version), when the first arriving operand waits until the second operand arrives. It can be seen that, this strategy does not perform well at all, leading, on average, to 16.7% increase in the original execution times. That is, waiting until both the operands meet in a location to perform computation is not a promising NDC strategy.

The second bar, for each benchmark, gives the benefits when an *oracle strategy* guarantees that the first thread waits only *till the breakeven point.* That is, if the first operand needs to wait beyond the breakeven point, this oracle strategy chooses not to perform the corresponding computation near data, i.e., it resorts to the conventional computing. Further, this oracle scheme also exercises NDC "more selectively". More specifically, if there is a reuse of one of the operands following the target computation, the oracle scheme favors data

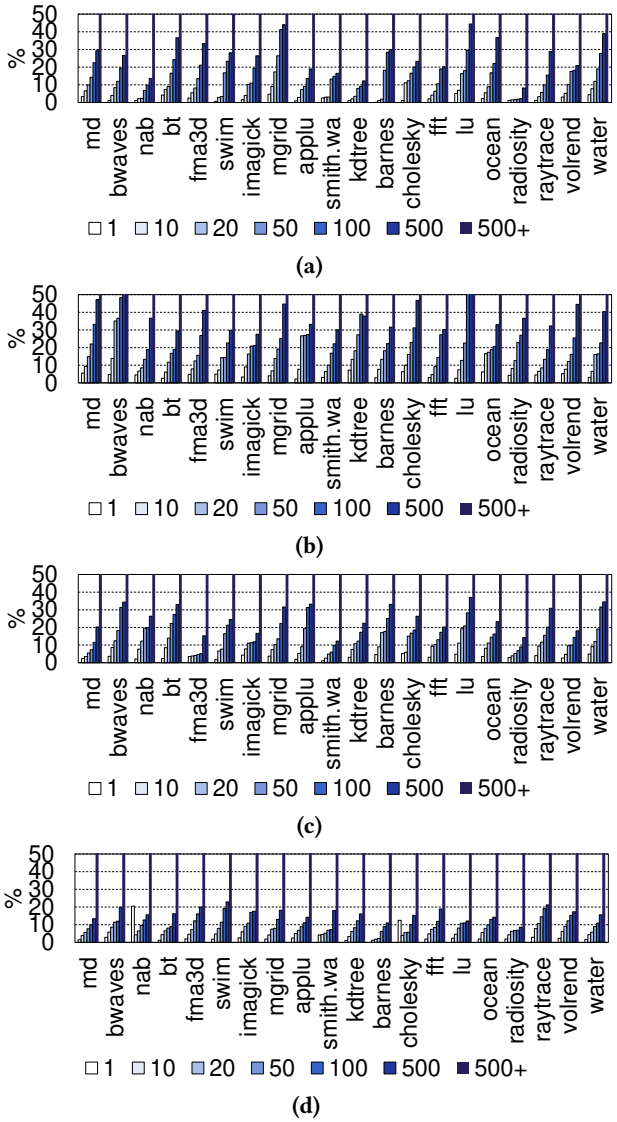

Figure 2. Distribution of arrival windows, (a) Link buffer, (b) L2 controller, (c) Memory controller, and (d) Main memory.

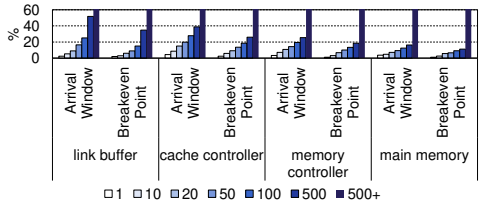

Figure 3. Comparison of breakeven points versus arrival windows.

locality and skips NDC.[2] That is, it strikes a balance between NDC and data locality optimization. Therefore, the second bar for each benchmark, represents, in a sense, the maximum

[2]In this evaluation, we assumed a single reuse is sufficient to favor data locality over NDC; but a similar, analysis can be made by requiring more than one reuse as well. Also, the reuse does not need to occur within a given number of cycles.

potential benefits. Clearly, since the oracle scheme uses future information, it is not practical. However, it provides an "upper-bound" for NDC potential (within the limits of our assumptions) and we use it in this paper as a bar against which our practical compiler-based strategies (discussed in Section 5) are compared. Note that, all benchmark programs benefit from the oracle scheme, leading to an average execution time improvement of 29.3% (geometric mean).

Figure 6 plots how the NDC opportunities exploited by the oracle scheme are distributed across our four target components. It can be observed from these results that the oracle scheme is able to exploit the NDC opportunities across all four components, leading to an average distribution of 25.9%, 36%, 21.7% and 16.4% for cache, network, memory controller and memory, respectively. Wait(x%) in Figure 4, on the other hand, gives the benefits when the first thread waits (for the second one) in the location of interest at most x% of the entire arrival window. That is, if the second operand arrives before x% of the arrival window, the computation is performed at the location; if not, the first operand leaves the location (and the computation is eventually performed in a core). The graph gives the results for x=5, 10, 25 and 50. While these results are better than those collected when waiting for the second operand to arrive, they are still below the original performance numbers. More specifically, wait(5%), wait(10%), wait(25%), and wait(50%) bring average slowdowns, by 15.1%, 14.7%, 13.9% and 13.4%, respectively.

At this point, one may consider employing a *predictor*, using which one can predict how much to wait for the second operand. A simple strategy would be assuming that, for a given program counter (PC) value, the next arrival window (i.e., the arrival window in the next invocation of the same instruction) is going to be the same as the current arrival window. Note that, if this prediction turns out to be accurate, then we can decide the optimal strategy for the first operand (i.e., the amount of time it needs to wait or not wait at all). The bar tagged as "Last Wait" shows the results (execution time improvements) when using such an arrival time predictor. The results shown in Figure 4 reveal that this predictor does not perform very well, bringing an average slowdown of 4.3%. To explain why this is the case, we give in Figure 5 two representative curves, one belonging to a PC value during the execution of *ocean* and the other belonging to a PC value during the execution of *radiosity*. Each curve plots 30 consecutive arrival windows. We see that the arrival times are *not* easily predictable (although not presented here, even a Markov Chain-based predictor generated similar results), which explains why the predictor-based approach brings improvement over the original execution only in two cases (*mgrid* and *volrend*). Overall, the results plotted in Figure 4 clearly show that neither implementing a fixed waiting time nor employing a prediction-based waiting strategy generates better results than the original case, and they are clearly far from the optimal savings achieved by the oracle scheme.

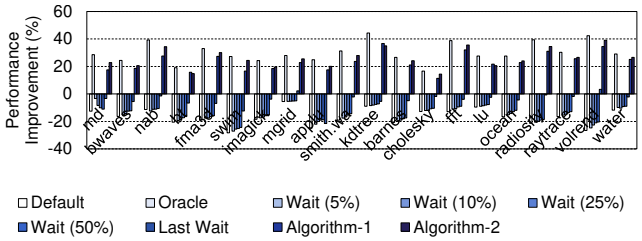

Figure 4. Performance benefits with different NDC schemes.

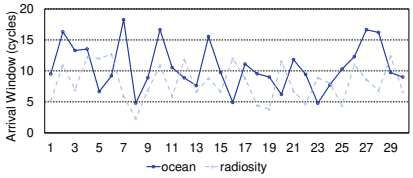

Figure 5. Arrival window sizes for 30 consecutive executions of a given instruction in two applications.

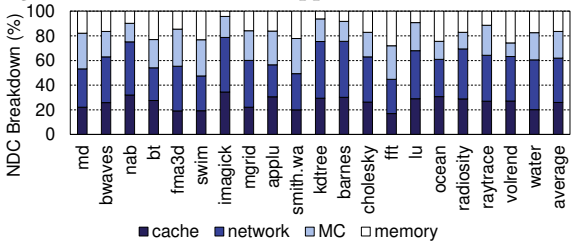

Figure 6. Distribution of locations where NDC is performed.

5 Enhancing NDC Opportunities

The results presented in the previous section clearly show that (i) the original execution style of the applications does *not* lend itself well to exploiting near data computing opportunities, and (ii) yet an oracle scheme can potentially bring significant benefits. (i) can be easily seen by comparing the first and second bars in Figure 4. Furthermore, strategies such as waiting for a specific duration (for the second operand to arrive at the location of interest) or using a predictor to predict the arrival window length do not bring much benefits either. Motivated by these observations, in this section, we present two novel compiler-directed approaches to near data computing. The primary goal of the first approach is to *reduce the arrival window lengths,* ideally making them small enough so that they become closer to breakeven points, whereas the second approach tries to balance between NDC and data locality optimization.

5.1 High-Level View of the Compiler

Figure 7 shows where our two compiler algorithms fit in the overall compilation process. After conventional parallelism (if needed) and locality optimizations, one of our algorithms (either Algorithm 1 or Algorithm 2) is invoked. The input to the algorithms are application code (coming from the parallelization and locality optimization steps) and an architecture description, which captures the hardware parameters such as the number of nodes, cores per core, target NDC locations, types of computations that can be performed in NDC locations. Both our algorithms also use a cache miss estimator.

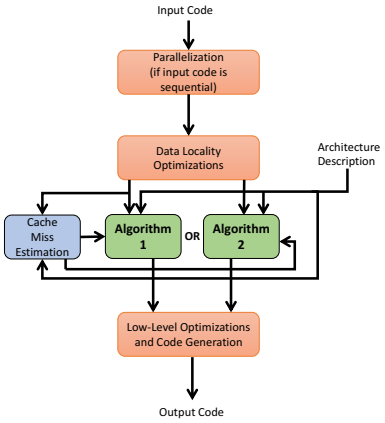

Figure 7. Illustration of where our compiler algorithms fit in the overall compilation process.

Following our algorithms, the low-level code optimizations are performed and the output code is generated.

5.2 Compiler-Directed NDC Optimizations

Recall from Section 2 that, in this work, we consider near data computing in four different hardware locations: link buffers, cache controllers, memory controllers, and memory banks. Accordingly, in this section, we evaluate a code restructuring strategy that helps one increase the NDC opportunities in one or more of these four locations. As stated earlier, the main goal of this restructuring is to reduce arrival window lengths. One necessary component is the identification of cache hits and misses in both the L1 and L2 layers.

For this cache miss estimation, we employ a variant of the approach described in [23], called Cache Miss Equations (CME). CME is built upon traditional compiler reuse analysis to generate linear Diophantine equations that summarize a given computation's memory behavior. Each solution of these Diophantine equations corresponds to a potential cache miss. The mathematical precision of CMEs allows one to reason about the impact of compiler optimizations on cache performance. The version of the CME implemented in our compiler closely follows the original work [23], and accurately models *cold*, *capacity* and *conflict* misses. In addition, it adds a few enhancements over [23]. In particular, our implementation can handle imperfectly nested loops, non-affine loop bounds and subscript expressions, and non-constant array sizes/loop bounds. It also works with record/union based data structures. In addition to these enhancements, we also engineered CME to reduce the time needed to solve the Diophantine equations resulting from data reuse analysis. However, at the time of this writing, our CME implementation does not model *coherence* misses.

Table 2 gives the cache hit/miss estimation accuracies in both L1 and L2 layers. It can be observed from these results that, on average, the accuracy of miss predictor is about 81.1% for L1 and 72.9% for L2. In the majority of the cases where our estimator mispredicts, the reason is coherence

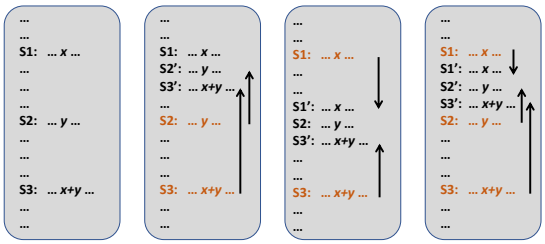

Figure 8. (a) An example code fragment. (b-d) Different access and computation movement strategies.

Table 2. L1 and L2 miss estimation accuracies.

	md	bwaves	nab	bt	fma3d	swim	imagick
L1	80.5	82.5	78.4	76.7	86.1	85	82.3
L2	77.7	79.2	74.4	66.7	81	80.6	80.1
	mgrid	applu	smith.wa	kdtree	barnes	cholesky	fft
L1	88.6	90.6	86.7	78	84.3	66.8	91.1
L2	83.4	85.6	74.4	71.2	70.5	55.3	72.3
	lu	ocean	radiosity	raytrace	volrend	water	average
L1	89	68	77.2	83.3	80.6	66.6	81.115
L2	70.7	55.4	74.1	80.1	70.6	55.5	72.94

misses, which are not modeled in our estimator. Overall, we believe that the estimation accuracies shown in Table 2 are reasonable and acceptable, considering the fact that they are achieved through static compile-time analysis alone.

Let us now discuss the details of our compiler-directed code restructuring strategy (our first compiler algorithm). Consider the code fragment shown in Figure 8(a). In this fragment, first, variable x is accessed in statement S1 and then variable y is accessed in statement S2. After these accesses, in statement S3, the program performs $x + y$. Now, from an NDC viewpoint, we want to perform this computation in, say, L2 controller (assuming that both x and y are destined for the same L2 bank). Note that if, for a given period of time, both x and y happen to be in the L2 bank, we can perform $x + y$ there, *before* the execution even reaches S3. However, this may not always be the case. For example, it is possible that the distance between S1 and S2 is too long and, as a result, x is replaced from the L2 cache before y reaches there.

In general, our compiler optimization tries to *reduce* what can be termed as **use-use** distance – in our example the distance between x and y with respect to a target location (L2 bank in the example above). This, in general, is a difficult optimization to implement because of the reasons explained below. However, first, we want to emphasize that, just bringing accesses to x and y close to one another in the "program code" would *not* work, as they can be in locations with different (physical) distances from the target location where the computation is to be performed (e.g., L2 bank).

5.2.1 Challenges and Solutions. First, as stated earlier, the main goal is to ensure that, x and y are in the same location of interest *around the same time* so that we can perform $x + y$. For example, if x and y are in the same L2 bank, we can compute $x + y$ there. For this to happen, i) both x and y should miss in the L1 cache (so that they access

L2) and ii) there should be a period of time during which both x and y should be L2 resident. Note that, i) can be checked using CME (explained earlier), but for ii), we need to make sure that the difference between the time x reaches the L2 bank and the time y reaches the same L2 bank is as *small* as possible. The solution we propose for this is to employ a novel instruction (an addition to ISA) called "pre-compute".[3] This pre-compute instruction is similar to conventional compute instructions in the ISA, except that it performs the specified computation in one of our four ("non-conventional") target locations. Our compiler inserts a pre-compute instruction for each computation it wants to offload to one of the four target locations.

Now, let us assume, for the sake of concreteness, that we want to perform $x + y$ in the L2 cache. Our compiler first checks whether x in S1 and y in S2 result in L1 misses. If this is the case, it then moves either x or y or both such that the time difference between their arrival time at the L2 bank is as small as possible. Following that, it moves computation $x + y$ to the point right after accesses to x and y.

Figures 8(b) through (d) show three potential data access (x and/or y) and computation (x+y) movements for the example code fragment in Figure 8(a). For a given $x + y$, our current implementation explores these three access movements in order. More specifically, first, it estimates the time at which x will be in the cache and then moves y with the goal of ensuring that it arrives the cache at around the same time as x. Finally, $x + y$ is moved. It needs to be noted that, such access movement is subject to the inherent *data* and *control dependencies* in the program code, i.e., the compiler moves the access to y, only if doing so does *not* violate any data or control dependencies in the program. If this movement of y is not possible (e.g., due to data dependencies), the compiler then tries to bring x close to y. If this also fails, the compiler then tries to bring both x and y close to one another. If this final attempt fails too, the next target component in the list (e.g., on-chip network) is tried (for more details, please see the discussion of the second challenge below). Note that, no matter what data access movement option is exercised (whether x or y or both are moved), the computation itself ($x + y$) is always moved to the point (place in the program code) right after the second variable is accessed.

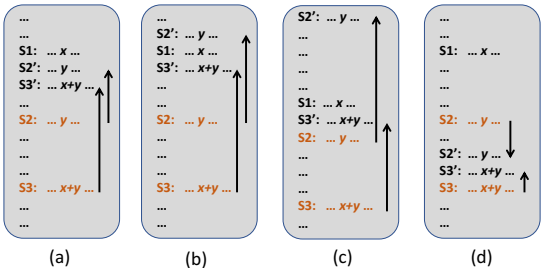

Figure 9. Different possible new locations for a variable (y).

[3]Note that many ISAs provide means to define new instructions.

To explain what estimations and tasks need to be performed by the compiler to enable such data access and computation movements, let us now focus on the scenario depicted in Figure 8(b). In this scenario, access to x is kept in its original place and y will be moved (by Δ). To determine Δ, our compiler i) considers the *physical locations* of x and y and determine (if we want to perform $x + y$ in the L2 bank) when x is expected to be in the L2 bank. It then moves y to a place in the code such that y will be co-residing in the cache with x. To do this, the compiler first estimates how many cycles the access to y needs to be moved and then translates this cycle count to program instructions. It is to be noted that, depending on the distances of x and y to the target L2 bank, the new location of y can be, with respect to its original position, (a) closer to x but still after x, or (b) closer to x but now before x, or (c) farther from x but above x, or (d) farther from x but still below x. These four possibilities (the new positions the access to variable y can take) are illustrated in Figure 9. (b) can occur, for example, when y is close to the target L2 bank, whereas (c) may be necessary when y is really far away from the target L2 bank and, as a result, its access needs to start earlier than the access to x. The second challenge is that we have 4 potential target locations (`loc`) to perform $x + y$ (not just L2 banks). For a given computation, our current implementation checks these possibilities *one by one*, in the order of network routers, L2 banks, network routers (the second attempt, since NoC is accessed in both L1 misses and L2 misses), memory queues, and memory banks. That is, first the on-chip network router is considered to perform the computation. If this fails, the approach tries to perform $x + y$ in the L2 bank. If this attempt also fails, the on-chip network router is tried again (this time the routers on the path of L2 miss). If that also fails, the memory queue is tried, and finally, if this attempt also fails, memory bank is tried. This trial order makes sense as it tries to perform $x + y$ at the *earliest component first*, and the order of components tried exactly matches the path followed by a data access.

So far, our explanation focused on scalar (non-array/non-record) variables and their computations (e.g., x and $x + y$). We now explain how our approach extends to loops and arrays. The main difference between the scalar and array computations arises from the definition of "distance". As stated earlier, in scalar computations, we can define distance as the distance between two program statements (or equally, between the last use and a program statement). In array computations, on the other hand, we need to measure it in terms of "loop iterations", as, for a given program statement, each loop iteration can be thought of as a separate statement (a different dynamic instance). As an example, let us consider the code fragment in Figure 10.

In this code fragment, a reuse of array elements in array X occurs along with the iterator (i, j). For instance, array element $X[5, 4]$ is first accessed in iteration $(5, 4)$, and then accessed in iteration $(6, 3)$. The distance between these two

iterations is $(1, -1)$. Let us assume the loop bound for i is n and the loop bound for j is m. The distance between the reuse of $X[5, 4]$ is $m - 1$ statements.

In the most general case, the problem of finding a loop transformation to achieve our goal can be expressed, in mathematical terms, as follows. We represent loop nests, loop bounds and array accesses using matrices and vectors. Specifically, for a loop nest with i_k being the index of the kth loop from top (where $1 \leq k \leq n$), $\vec{I} = (i_1 i_2 \cdots i_n)^T$ represents the iteration vector, and a given access to m-dimensional array X is represented by $X(f(\vec{I}))$ where f is $F\vec{I} + \vec{f}$, where F is an m×n matrix and $\vec{f}$ is an m-entry vector. For example, if $\vec{I} = (i_1\ i_2)^T$ and $f(\vec{I}) = (i_1 + 1\ i_2 - 1)^T$, then F is the identity matrix and $f = (1\ -1)^T$. Further, in loop transformation theory [64], on applying a loop transformation T, a given original loop iteration vector $\vec{I}$ is mapped (transformed) to a new loop iteration $T\vec{I}$.

```
for i 2 ... n
  for j 1 ... m
    X[i,j] = ... X[i-1, j+1]
```

Figure 10. An example loop nest.

Now, let $\vec{I}_x$ be the iteration at which $X(f(\vec{I}_x))$ is accessed and $\vec{I}_y$ be the iteration at which $Y(g(\vec{I}_y))$ is accessed, where $\vec{I}_b \leq \vec{I}_x, \vec{I}_y \leq \vec{I}_e$, and $\vec{I}_b$ and $\vec{I}_e$ being the lower and upper bounds, respectively, for the loop nest. Similarly, let $\vec{I}_c$ be the iteration at which $X(h(\vec{k}_x)) + Y(l(\vec{k}_y))$, for some $\vec{k}_x$ and $\vec{k}_y$, is originally to be performed. If the compiler determines that access to $Y(g(\vec{k}_y))$ needs to be moved to iteration, say, $\vec{k'}_y$, it tries to find a loop transformation matrix T such that, we have $T\vec{I}_y = \vec{k'}_y$, that is, the original loop iteration $\vec{I}_y$ should be mapped, by T, to $\vec{k'}_y$. Assuming that D is the dependence matrix for the loop nest in question, for this transformation to be valid (semantics-preserving), each column of TD should be lexicographically positive.[4] Similarly, T also needs to map $\vec{I}_c$ (the original iteration where the computation is scheduled to be performed) to $\vec{I'}_c = T\vec{I}_c$ where $\vec{I'}_c$ is the iteration that *immediately follows* $\vec{k'}_y$, in the "new" iteration space. Thus, for a given access and a computation, the compiler needs to find a loop transformation matrix T that satisfies *both* $\vec{k'}_y = T\vec{I}_y$ and $\vec{I'}_c = T\vec{I}_c$. Our compiler constructs such constraints for each data access-computation pair for which it wants to perform NDC, and then tries to solve them, as explained in Section 5.2.2.

The third challenge is that, different from the cache bank, memory queue and memory bank cases, in the case of on-chip network, our compiler also considers "alternate data access paths" (routes), in an attempt to increase chances for NDC. It is known from prior research [39] that, in a 2D space there exist multiple paths with the same shortest distance from node (p_1, q_1) to node (p_2, q_2). Using the same terminology from [39], each such (minimum distance) path from (p_1, q_1) to (p_2, q_2) is represented using a signature $S\{(p_1, q_1), (p_2, q_2)\}$. Note that, for an on-chip network with a total L communication links, a signature can be represented using an $L - bit$ sequence in which the k^{th} bit is 1 if the communication uses the k^{th} link; otherwise, it is set to 0. Note that, given (p_1, q_1) and (p_2, q_2), the compiler can choose an signature $S\{(p_1, q_1), (p_2, q_2)\}$, depending on its goal.

Figure 11. Two different routings with the same distance on a 6 × 6 NoC-based manycore.

Now, given two data accesses, x and y, issued respectively from cores (p_x, q_x) and (p_y, q_y), and accessing, respectively, L2 caches located at (p_r, q_r) and (p_s, q_s), our goal is to select signatures $S\{(p_x, q_x), (p_r, q_r)\}$ and $S\{(p_y, q_y), (p_s, q_s)\}$ such that the number of common links between them is maximized, or equivalently, the total number of 1s in $S\{(p_x, q_x), (p_r, q_r)\} \cap S\{(p_y, q_y), (p_s, q_s)\}$ is maximized, where $\cap$ represents bitwise-and operation. Figure 11 illustrates an example scenario where in (a) the two data accesses do not have any common link but in (b) they have two common links. Clearly, the maximizing the number of common links between $S\{(p_x, q_x), (p_r, q_r)\}$ and $S\{(p_y, q_y), (p_s, q_s)\}$ is beneficial from an NDC perspective, as each common link represents an "opportunity" to perform computation $x + y$ in one of the associated link routers. Therefore, in the case of on-chip network, our current implementation selects signatures carefully in an attempt to maximize 1s in $S\{(p_x, q_x), (p_r, q_r)\} \cap S\{(p_y, q_y), (p_s, q_s)\}$.

Fourth, in some cases, it is simply impossible for x and y to meet in any target location. For example, x and y are mapped to different cache banks and different memory banks, or their paths on the on-chip network could not be intersected (unless we are willing to increase the number of links to be traversed by the data accesses). While in such cases changing the mapping between data space and cache/memory banks can help (to create more NDC opportunities), we postpone such data layout optimizations to a future study.

5.2.2 Compiler Algorithm. Algorithm 1 provides our compiler approach to achieve the NDC optimization. For clarity of presentation, we present the code for the array access case, and the scalar version is similar.[5]. Our approach first generates the use-use chains and, data dependency graph by analyzing the targeted loop nest L. As we mentioned earlier, our approach tries different locations one-by-one to exploit

[4] This directly follows from the legality constraint of any loop transformation [64].

[5] Due to space concern, we do not show the code generation algorithm. Further details can be found in [34]

NDC opportunities there (lines 45 - 49). For a particular location, e.g., using L2 bank as an example, we try different strategies of moving the data access and computation, as we discussed earlier in Figure 8. Specifically, we first move y and computation (z) without changing the iterator of x (lines 13 - 16). If moving y and computation to x does not generate a legal loop transformation T with all available strides, we try to move x and computation z to y (lines 18 - 21). Similarly, if both do not generate legal loop transformations, we try to move x, y, and z (lines 23 - 26). If there is no opportunity to compute the z at L2 bank (e.g., due to x and y are not present in L2 on time), we try to exploit NDC at routers (line 44). Note that, the process is similar to exploiting NDC at L2 bank, and the only difference is that we first need to select optimal signatures of NoC links for NDC (line 29). Then, the CME is replaced with checking whether x and y will be arriving at the same NoC router.

5.3 Exploring NDC-Data Locality Tradeoff

```
...
S1:  ... x ...
...
...
S2:  ... y ...
...
...
S3:  ... x+y ...
...
...
S4:  ... y*z ...
...
S5:  ... t/y ...
```

Figure 12. An example code fragment with reuse of y.

In this section, we explore the tradeoffs between NDC and data locality optimization. More specifically, we try to identify the cases where exercising NDC may *not* be the best (most beneficial) option, and one would rather consider bringing data to core. Consider for instance, the code fragment given in Figure 12. In this code, while it is possible to exercise near data computing for $x + y$ (say in L2), doing so may not be the best option. This is because, one of the operands, y, has two more *reuses* following the computation ($x + y$), and while potentially the associated two computations can also be performed in L2, this option may not be any faster than bringing y (and x) to L1 and performing the computation in the original core.

We propose a **data reuse-aware** version of NDC where the NDC is exercised, *only if* the operands involved in the target computation do not have any reuse beyond the computation. Note that this is clearly only one of the potential ways of tuning the "aggressiveness" of NDC based on "subsequent data reuse" (and it is the only one our current compiler implementation employs). An alternate strategy would be exercising NDC, only if the operands involved in the target computation are not reused more than k times beyond the target computation to be offloaded. Clearly, this version can be more successful in general; however, determining the right value for k may not be trivial. Thus, we postpone it to a future study. In a sense, our current implementation is a special case where $k = 0$, i.e., we favor data reuse over NDC even if there exists only 1 reuse of one of the operands beyond the target computation (being considered to be offloaded).

Algorithm 1 Exploiting NDC through computation restructuring.

INPUT: Loop nest (L); architecture configuration: number of cores, on-chip network topology;
OUTPUT: Transformed loop nest with NDC optimization;
1: **function** Loop_Transformation(iteration k_x, iteration k_y, loop L, dependency graph D)
2: Let $\vec{I}_x$, $\vec{I}_y$, and $\vec{I}_c$ be the iteration at which x, y, z are accessed
3: Solve T for $\vec{k}_x = T\vec{I}_x$, $\vec{k}_y = T\vec{I}_y$ and $\vec{I'}_c = T\vec{I}_c$.
4: **if** T *exist* **then**
5: **return** L' = TL
6: **else**
7: **return** L
8: **function** L2_bank_compute(x,y,z, L,D)
9: **if** CME (x, y) in L2 bank **then**
10: Loop_Transformation($\vec{k}_x$, $\vec{k}_y$, x,y,z, L, D)
11: **return** TRUE;
12: /** Fix x and try to move y close to x */
13: move y from iteration $\vec{k}_y$ to $\vec{k'}_y$ with Δ
14: **if** CME (x, y) in L2 bank **then**
15: L ← Loop_Transformation($\vec{k}_x$, $\vec{k'}_y$, x,y,z, L, D)
16: **return** TRUE;
17: /** Fix y and try to move x close to y */
18: move x from iteration $\vec{k}_x$ to $\vec{k'}_x$ with Δ
19: **if** CME (x, y) in L2 bank **then**
20: L ← Loop_Transformation($\vec{k'}_x$, $\vec{k}_y$, x,y,z, L, D)
21: **return** TRUE;
22: /** Move both x and y */
23: move (x,y) from iteration ($\vec{k}_x$, $\vec{k}_y$) to ($\vec{k'}_x$, $\vec{k'}_y$) with Δ
24: **if** CME (x, y) in L2 bank **then**
25: L ← Loop_Transformation($\vec{k'}_x$, $\vec{k'}_y$, x,y,z, L, D)
26: **return** TRUE;
27: **function** Router_compute(C, L,D)
28: /** get signatures of x, y, and reshape the routing to create more NDC opportunity*/
29: find the signatures with maximized $S_x \cap S_y$
30: Similar procedure compared with L2_bank_compute, except CME estimation is replaced with same router check under the selected signature.
31: **function** Memory_queue_compute(C, L,D)
32: Similar procedure compared with L2_bank_compute, except CME estimation is replaced with same memory queue check.
33: **function** Memory_bank_compute(C, L,D)
34: Similar procedure compared with L2_bank_compute, except CME estimation is replaced with same memory bank check.
35: /** construct all use-use chains */
36: S ← extract_use-use_chains(L)
37: D ← dependency_analysis(L)
38: **for** each use-use chain C_j in S **do**
39: /** get data accesses from C_j */
40: x ← $X(h(\vec{k}_x))$, y ← $Y(l(\vec{k}_y)$
41: z ← $X(h(\vec{k}_x)) + Y(l(\vec{k}_y))$
42: **if** L2_bank_compute (x,y,z, L,D) **then**
43: **break;**
44: **if** Router_compute (x,y,z, L,D) **then**
45: **break;**
46: **if** Memory_queue_compute (x,y,z, L,D) **then**
47: **break;**
48: **if** Memory_bank_compute (x,y,z, L,D) **then**
49: **break;**

We now give the mathematical formulation of this data reuse-aware NDC optimization problem, for array computations. Let $\vec{I}_x$ be the iteration at which $X(f(\vec{I}_x))$ is accessed and $\vec{I}_y$ be the iteration at which $Y(g(\vec{I}_y))$ is accessed. Further, let $\vec{I}_c$ be the iteration at which $X(h(\vec{k}_x)) + Y(l(\vec{k}_y))$, for some $\vec{k}_x$ and $\vec{k}_y$, is originally to be performed. As we have discussed earlier in Section 5.2.1, if the compiler determines that access to $Y(g(\vec{k}_y))$ needs to be moved to iteration, say, $\vec{k'}_y$, it needs to find a loop transformation matrix T such

Algorithm 2 Exploring NDC-data locality tradeoff.

INPUT: Loop nest (L); architecture configuration: number of cores, on-chip network topology;
OUTPUT: Transformed loop nest with NDC optimization;
1: **function** Loop_Transformation(iteration k_x, iteration k_y, loop L, dependency graph D)
2: Let $\vec{I}_x$, $\vec{I}_y$, and $\vec{I}_c$ be the iteration at which x, y, z are accessed
3: Solve T for $\vec{k}_x = T\vec{I}_x$, $\vec{k}_y = T\vec{I}_y$ and $\vec{I'}_c = T\vec{I}_c$.
4: /** consider data reuse */
5: **if** Exist $\vec{I}_m$ and references p, q. & $\vec{I}_e > \vec{I}_m > \vec{I}_c$ & $\{f(\vec{I}_x) = p(\vec{I}_m)$ or $g(\vec{I}_y) = l(\vec{I}_m)\}$ **then**
6: **return** L
7: **if** T *exist* **then**
8: **return** L' = TL
9: **else**
10: **return** L
11: *... The rest is the same as in Algorithm 1*

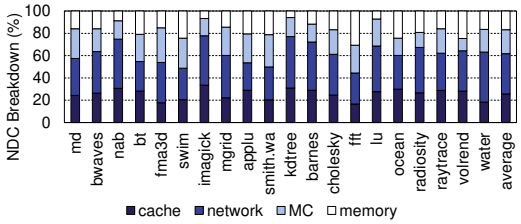

Figure 13. Distribution of locations where NDC is performed (Algorithm 1).

that, we have $T\vec{I}_y = \vec{k'}_y$, that is, the original loop iteration $\vec{I}_y$ should be mapped, by T, to $\vec{k'}_y$. However, now, in the case of the data reuse-aware version, in addition to the dependency constraint, we also need to make sure that neither $X(f(\vec{I}_x))$ nor $Y(g(\vec{I}_y))$ is reused beyond $\vec{I}_c$. So, the compiler checks whether there exists any $\vec{I}_m$ such that $\vec{I}_e > \vec{I}_m > \vec{I}_c$ and $\{f(\vec{I}_x) = p(\vec{I}_m)$ or $g(\vec{I}_y) = l(\vec{I}_m)\}$ for some references p and q to data structures X and Y, respectively. That is, the compiler will determine a T (i.e., try to exercise near data computing) only if there exists no $\vec{I}_m$ that satisfies the constraint given above. The algorithm given in Section 5.3.1 presents this idea in a pseudo-code form. If desired, this algorithm can be easily modified to check for the existence of multiple data reuses of the operands beyond $\vec{I}_c$.

5.3.1 Compiler Algorithm. The data reuse-aware version of our NDC algorithm is presented in Algorithm 2. Instead of returning the transformation matrix T (if found) as we discussed in Algorithm 1, Algorithm 2 checks whether any data reuse opportunities are compromised when applying the determined T. If any data reuse opportunity could be missed, then the compiler skips that transformation and tries to explore other potential transformations (lines 4 - 6).

5.4 Experimental Evaluation

The eighth bar for each benchmark program in Figure 4 gives the performance improvement brought by our first algorithm, Algorithm 1. We see that our approach brings about 22.5% average performance improvement (execution time reduction) over the baseline, and the improvements range between 11.4% (*cholesky*) and 37% (*kdtree*).

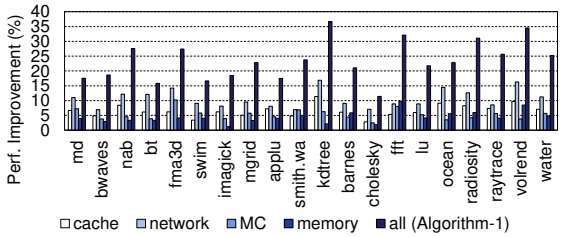

Figure 14. Results when Algorithm 1 is applied to a specific component alone.

To explain where these benefits are coming from, we give in Figure 13 the near data computations performed in different locations, as a fraction of the total amount of near data computations (when considering all four locations we target).[6] It can be observed that most of the near data computations are performed in on-chip network followed by cache banks and memory controllers. These results can be explained as follows. First, it is to be noted that, on-chip network is used in both L1 misses and L2 misses; so, it inherently has a large scope for NDC optimization. Second and more importantly, recall from our discussion in Section 5.2 that, our approach has an additional knob when considering NDC in on-chip network – changing the message routes to improve chances for two data accesses to intersect/overlap on the on-chip network. In comparison, NDC optimization targeting the remaining three locations uses only the knob of reducing the time distance between two data accesses. We also see from Figure 13 that, since the L2 cache banks receive many more data accesses than memory banks, they contain more opportunities for NDC. Finally, a memory controller controls a number of banks and is accessed before the memory banks and, therefore, it enjoys more NDC opportunities than the off-chip memory. We also performed experiments with a version where the message re-routing flexibility in the network is not exercised. Although the results are not presented here in detail, we observed that this version of NDC reduced the amount of near data computations performed in message routers by nearly 40%, on average.

It is interesting to compare these distribution results to those of the oracle scheme in Figure 6. We observe that the two distribution plots are quite similar, indicating that our compiler-based approach exploits, in most cases, the same set of NDC opportunities exploited by the oracle scheme.

On the other hand, Figure 14 shows the percentage of performance improvements achieved when our approach (Algorithm 1) is applied only to a specific location in an isolated fashion. We want to point out that the sum of the savings in this graph (for a given application) is more than the total saving shown for the same application in Figure 13. This is because, when NDC is enabled in all four locations, it

[6] Although we do not present here detailed results due to space limit, we want to mention that, when Algorithm 1 is used, on average, about 32% of the total arithmetic and logical instructions are executed in some form of NDC.

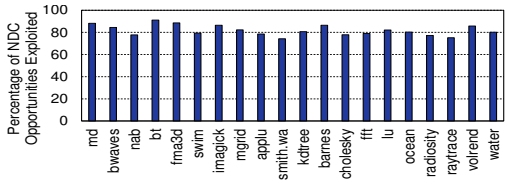

Figure 15. Fraction of NDC opportunities exercised by Algorithm 2.

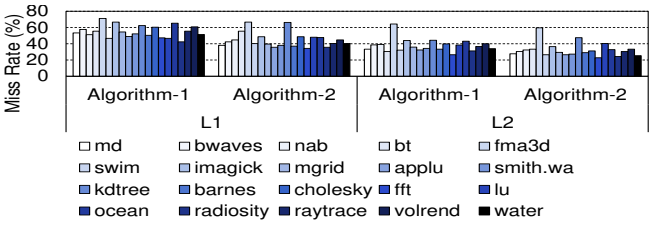

Figure 16. L1 and L2 miss rates with Algorithms 1 and 2.

will be performed in only one them and not the others. For example, once an $x+y$ is performed during the message traffic from L1 to L2, it will not be performed at L2. Similarly, if it is performed in a memory controller, it will not be performed in the memory bank itself. The results plotted in Figure 14 also indicate that it is critical to exploit the NDC opportunities in *all* four locations, for maximizing performance savings.

Next, we discuss the effectiveness of Algorithm 2 (Section 5.3). The last bar for each benchmark program in Figure 4 gives the performance improvement brought by this algorithm. We observe that this data reuse-aware version of the algorithm brings an average performance improvement of 25.2%. Comparing this with the results of Algorithm 1, we see that it brings further improvements over Algorithm 1 in all but three benchmark programs (*bt*, *kdtree*, and *lu*). To explain this further, we plot, in Figure 16, L1 and L2 miss rates of the Algorithm 1 and Algorithm 2. It can be seen that, in all 20 benchmark programs tested, Algorithm 2 generates lower cache miss rates than Algorithm 1. As explained earlier, this is because Algorithm 2 exercises near data computing *more selectively*, compared to Algorithm 1, which performs near data computing whenever opportunity arises. In fact, Figure 15 gives the number of NDC opportunities exercised by Algorithm 2, as a fraction of the total number of NDC opportunities seen during execution. It can be observed that, on average, Algorithm 2 exploits 81.8% of NDC opportunities; the remaining opportunities are *bypassed* due to data locality concerns (i.e., there is at least one reuse of one of the operands after the target (offloaded) computation is performed). In three benchmark programs (*bt*, *kdtree*, and *lu*) however, Algorithm 2 generates (less than 5%) worse results than Algorithm 1, primarily due to the inaccuracy in identifying the existence of data reuse.

Let us now discuss where the differences between the oracle scheme and Algorithms 1/2 are coming from. In the case of Algorithm 1, most of the time when it makes the wrong decision, it is due to mispredicting the cache hits/misses. At

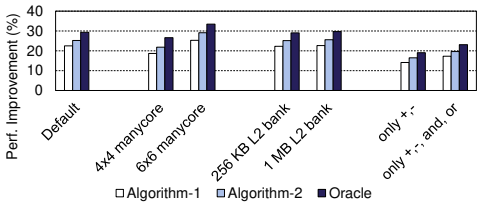

Figure 17. Results from the sensitivity experiments. The first three bars use the default simulation parameters.

other times, it fails to identify the right hardware component on which to perform NDC. Recall that, Algorithm 1 considers hardware components in a certain order and this order, while works very well in the overwhelming majority of cases, sometimes causes the algorithm to perform NDC prematurely in a non-optimal location. On the other hand, in the case of Algorithm 2, it sometimes wrongly favors data locality optimization over NDC when it catches even only 1 reuse of data. In our future work, we plan to explore the problem of optimal selection of the k parameter's value. We believe that, in the long run, these shortcomings of our algorithms can be fixed by employing more accurate cache miss predictors, more flexible NDC search space exploration strategies, and carefully determined k values.

Finally, we change the default values of some of our experimental parameters and conduct a sensitivity study. In each experiment, the value of only one parameter is modified and the values of the remaining parameters are kept at their default values given in Table 1. The results of our sensitivity experiments are shown in Figure 17. Note that each bar in this graph represents the geometric mean of performance improvements across all 20 application programs. In the first sensitivity experiment, we change the manycore size. Recall that the default size was 5×5 cores with 1 thread per core. The results with manycores of sizes 4×4 and 6×6 (again, 1 thread per core) indicate that the benefits of our approach get better with increased cores. This is mainly because a large manycore provides more locations for performing near data computing. On the other hand, the experiments with different L2 bank capacities (recall that the default L2 bank capacity per node was 512 KB) indicate that our savings are not very sensitive to the cache capacity. This is because as the cache capacity is increased, the amount of near data computation that can be done in the cache controller increases; but, this in turn reduces the amount of near data computation performed in network, memory controllers and main memory. In other words, the location at which near data computation is done changes and the amount of near data computation does not; consequently, the results are similar across the different L2 cache capacities tested. In our last set of experiments, we restricted the types of computations that can be performed near data. Recall that by default we assumed that all types of computations arithmetic and logic computations can be performed near data. The results in Figure 17 indicate that, even if we restrict the types of the

computations that can be performed near data to only "+" and "-", our two compiler algorithms still achieve average performance improvements of 14.1% and 16.5%.

We also implemented a version of our algorithms in which, instead of individual computations, a large number of computations (e.g., entire loop nest) are mapped to a location for NDC. Our experiments with this version reveal that it performs not very well in most cases, bringing only 1.2% and 2.5% average improvements for our two algorithms. Hence, we believe that fine grain (instruction level) mapping is critical to take full advantage of NDC.

6 Related Work

The idea of moving data close to the memory and computing there (processing-in-memory, PIM) is suggested in the early 70s [56]; however at that time, due to the hardware limitations, it could not be fully utilized. Recently, there is renewed interest in PIM mainly due to 3D stacked memory. Hybrid Memory Cube (HMC) [30] employs 3D stacked DRAM with computation units inside the memory component.

Software Optimizations for NDC There has been various software approaches [26, 29, 50, 58] that utilize NDC. Tang et al. [58] suggested a compiler approach that takes advantage of NDC. They could reduce the distance-to-data on the on-chip network by partitioning the computations in a nested loop into subcomputations, each assigned a different core. Hsieh et al. [29] proposed schemes for offloading code to 3D stacked memory which can minimize off-chip bandwidth usage. Hadidi et al. [26] explored a compiler-assisted strategy that takes advantage of instruction-level PIM offloading. Rafique et al. [50] proposed a conflict-aware memory-side prefetching scheme for HMC main memory to exploit the TSV bandwidth. Kim et al. [35] explored the potential of spreading data across multiple memory stacks.

Hardware Optimizations for NDC Besides software approaches, there are also numerous hardware optimizations for NDC [4, 6, 18, 49, 55]. Ahn et al. [6] proposed a processing-in-memory (PIM) architecture that automatically decides whether to run on either PIM or processors depending on the data locality. Likewise, Ahn et al. [4] proposed Tesseract, which is a programmable PIM accelerator for graph processing. Farmahini-Farahani [18] proposed and evaluated the DRAM-Accelerator (DRAMA) architecture, which can offload compute- and data-intensive operations to 3D stacked DRAM device on top of CGRAs. Pugsley et al. [49] focused on MapReduce workloads and improved performance and energy consumption by utilizing massive parallelism and largely localized memory accesses. Sterling et al. [55] designed a PIM-based architecture, which facilitates the virtualization of tasks/data and providing resource management for load balancing and latency tolerance.

Optimizations Similar to NDC Tang et al. [57] proposed compiler-driven approaches to reduce the data movement by replacing a costly data access with a few less expensive data accesses with some extra computation. Prior works such as [20, 44, 45] improved performance by keeping frequently used data close to the processor. Also, in the context of the network-on-chip (NoC) architectures, minimizing data movements has been used as a strategy to improve performance and power consumption [31, 33, 51].

In addition, there exists a large body of prior work on conventional data locality optimizations [11, 17, 38, 41–43, 59]. Since these works do not attempt to take advantage of near data computing, we do not discuss them here in detail.

How Do We Differ? To the best of our knowledge, none of the prior works explored the potential of NDC in detail. The oracle scheme presented in our paper gives us an upper-bound for potential gains that could be achieved when employing NDC. We also present two novel compiler approaches, one of which considers the tradeoff between NDC optimization and data locality optimization, a perspective that has not been considered by prior works.

7 Concluding Remarks

This paper presents a detailed evaluation of near data computing (NDC) opportunities, targeting two benchmark suites. Our results indicate that it is not trivial to take full advantage of near data computing, even if one is willing to provide specialized hardware to implement it. In particular, our experimental analysis reveals that: i) different applications achieve different amounts of gain when performing NDC in different hardware locations, ii) however, for the best results, each application typically needs to exercise near data computing in all available locations (e.g., cache controllers, memory controllers, on-chip network, and off-chip memory), and iii) if performed in an optimal fashion (an oracle scheme), near data computing can return significant performance benefits. In this paper, We also implemented and evaluated two different compiler-based NDC strategies. These compiler schemes improved the performance of 20 multithreaded application programs by 22.5% and 25.2%, on average.

Our future work includes performing a similar evaluation and investigating similar compiler strategies targeting GPUs. Work is also underway in developing NDC strategies targeting emerging deep learning workloads. We also plan to explore NDC opportunities when targeting large scale (multi-node) ensembles of CPUs and GPUs.

Acknowledgement

The authors would like to thank the PACT 2020 reviewers for their constructive feedback and suggestions. The authors would also like to thank Dr.David A Padua for shepherding the paper. This work is supported in part by NSF grants #1908793, #1629915, #1629129, #1763681, #2028929, #2008398, #2011146, and #1931531, as well as a startup funding from the University of Pittsburgh.

References

[1] 2012. The Architecture and Performance of the TILE-Gx Processor Family. http://www.tilera.com/products/processors/TILE-Gx_Family.

[2] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*.

[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proc. of the International Symposium on Computer Architecture (ISCA)*.

[5] 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 Annual International Symposium on Computer Architecture (ISCA)*.

[6] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proc. of the International Symposium on Computer Architecture (ISCA)*.

[7] Jennifer M. Anderson and Monica S. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*.

[8] Jeffery M. Arnold, Duncan A. Buell, and Elaine G. Davis. 1992. SPLASH 2. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*.

[9] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems. In *2016 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[10] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *OpenMP Shared Memory Parallel Programming*, Rudolf Eigenmann and Michael J. Voss (Eds.).

[11] Kristof Beyls and Erik H. D'Hollander. 2009. Refactoring for Data Locality. *Computer* 42, 2 (2009).

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

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

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

[15] 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 of International Symposium on High-Performance Computer Architecture*.

[16] Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. 2020. Near Data Acceleration with Concurrent Host Access. In *ISCA*.

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

[18] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. DRAMA: An Architecture for Accelerated Processing Near Memory. *IEEE Computer Architecture Letters* 14, 1 (2015).

[19] Sílvio Fernandes, Bruno C. Oliveira, and Ivan Saraiva Silva. 2009. Using NoC Routers as Processing Elements. In *Proceedings of the Symposium on Integrated Circuits and System Design: Chip on the Dunes*.

[20] Pierfrancesco Foglia, Cosimo A. Prete, Marco Solinas, and Giovanna Monni. 2010. Re-NUCA: Boosting CMP Performance Through Block Replication. In *Proc. of the Euromicro Conference on Digital System Design: Architectures, Methods and Tools*.

[21] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang. 2016. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences* 59, 7 (21 Jun 2016), 072001. https://doi.org/10.1007/s11432-016-5588-7

[22] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 113–124.

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

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

[25] Peng Gu, yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. 2020. iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture. In *ISCA*.

[26] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. 2017. CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory. *Trans. Archit. Code Optim.* 14, 4 (2017).

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

[28] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2016. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *Proccedings of the International Symposium on Computer Architecture (ISCA)*.

[29] 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 *Proc. of the International Symposium on Computer Architecture*.

[30] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *Proc. of the Symposium on VLSI Technology (VLSIT)*.

[31] Yuho Jin. 2015. Unifying Router Power Gating with Data Placement for Energy-Efficient NoC. In *Proc. of the International Symposium on Computer Architecture and High Performance Computing*.

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

[33] Mahmut Kandemir, Yuanrui Zhang, Jun Liu, and Taylan Yemliha. 2011. Neighborhood-Aware Data Locality Optimization for NoC-Based Multicores. In *Proc. of the International Symposium on Code Generation and Optimization*.

[34] Mahmut Taylan Kandemir, Jihyun Ryoo, Xulong Tang, and Mustafa Karakoy. 2021. Compiler Support for Near Data Computing. *Technical Report, Department of Computer Science and Engineering, The Pennsylvania State University* (2021).

[35] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward standardized near-data processing with unrestricted data placement for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

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

[37] 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*.

[38] Monica S. Lam and Michael E. Wolf. 2004. A Data Locality Optimizing Algorithm. *SIGPLAN Not.* 39, 4 (2004).

[39] Feihui Li, Guangyu Chen, Mahmut Kandemir, and Ibrahim Kolcu. 2007. Profile-Driven Energy Reduction in Network-on-Chips. *SIGPLAN Not.* 42, 6 (2007), 394–404.

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

[41] 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 *Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[42] Chikeung Luk and Todd C. Mowry. 1996. Compiler-based prefetching for recursive data structures. *SIGPLAN Not.* 31, 9 (1996).

[43] Kathryn S. Mckinley, Steve Carr, and Chauwen Tseng. 1996. Improving Data Locality with Loop Transformations. *Transactions on Programming Languages and Systems (TOPLAS)* 18, 4 (1996).

[44] Javier Merino, Valentin Puente, and Jose A. Gregorio. 2010. ESP-NUCA: A low-cost adaptive Non-Uniform Cache Architecture. In *Proc. of the International Symposium on High-Performance Computer Architecture*.

[45] Javier Merino, Valentín Puente, Pablo Prieto, and José Ángel Gregorio. 2008. SP-NUCA: A Cost Effective Dynamic Non-Uniform Cache Architecture. *SIGARCH Comput. Archit. News* 36, 2 (2008).

[46] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Enabling Practical Processing in and near Memory for Data-Intensive Computing. In *Proceedings of the Design Automation Conference 2019*.

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

[48] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2019. Opportunistic computing in gpu architectures. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 210–223.

[49] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Proc. of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[50] Muhammad M. Rafique and Zhichun Zhu. 2018. CAMPS: Conflict-Aware Memory-Side Prefetching Scheme for Hybrid Memory Cube. In *Proc. of the International Conference on Parallel Processing*.

[51] Qingchuan Shi, Farrukh Hijaz, and Omer Khan. 2013. Towards efficient dynamic data placement in NoC-based multicores. In *Proc. of the International Conference on Computer Design (ICCD)*.

[52] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. 2017. Pageforge: A near-Memory Content-Aware Page-Merging Architecture. In *Proceedings of the International Symposium on Microarchitecture*.

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

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

[55] Thomas L. Sterling and Hans P. Zima. 2002. Gilgamesh: A Multithreaded Processor-in-Memory Architecture for Petaflops Computing. In *Proc. of the Conference on Supercomputing*.

[56] Harold S. Stone. 1970. A Logic-in-Memory Computer. *Computers* C-19, 1 (1970).

[57] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2018. Computing with Near Data. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 3 (2018).

[58] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proc. of the International Symposium on Microarchitecture*.

[59] Xulong Tang, Mahmut Taylan Kandemir, Mustafa Karakoy, and Meena Arunachalam. 2019. Co-Optimizing Memory-Level Parallelism and Cache-Level Parallelism. In *Proceedings of the 40th annual ACM SIGPLAN conference on Programming Language Design and Implementation*.

[60] Gabriel Urzaiz, David Villa, Felix Villanueva, and Juan Carlos Lopez. 2012. Process-in-Network: A Comprehensive Network Processing Approach. *Sensors (Basel)* 12, 6 (2012), 8112–8134.

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

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

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

[64] Michael Wolfe. 1995. *high performance compilers for parallel computing*.

[65] Xu Yang, Yumin Hou, and Hu He. 2019. A Processing-in-Memory Architecture Programming Paradigm for Wireless Internet-of-Things Applications. *Sensors (Basel)* 19, 1 (2019), 140.