Distance-in-Time versus Distance-in-Space

Mahmut Taylan Kandemir
mtk2@psu.edu
Pennsylvania State University
State College, PA, USA

Xulong Tang
tax6@pitt.edu
University of Pittsburgh
Pittsburgh, PA, USA

Hui Zhao
Hui.Zhao@unt.edu
University of North Texas
Denton, TX, USA

Jihyun Ryoo
jihyunryoo@gmail.com
Pennsylvania State University
State College, PA, USA

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

Abstract

Cache behavior is one of the major factors that influence the performance of applications. Most of the existing compiler techniques that target cache memories focus exclusively on reducing data reuse distances in time (DIT). However, current manycore systems employ distributed on-chip caches that are connected using an on-chip network. As a result, a reused data element/block needs to travel over this on-chip network, and the distance to be traveled – reuse distance in space (DIS) – can be as influential in dictating application performance as reuse DIT. This paper represents the first attempt at defining a compiler framework that accommodates both DIT and DIS. Specifically, it first classifies data reuses into four groups: G1: (low DIT, low DIS), G2: (high DIT, low DIS), G3: (low DIT, high DIS), and G4: (high DIT, high DIS). Then, observing that reuses in G1 represent the ideal case and there is nothing much to be done in computations in G4, it proposes a "reuse transfer" strategy that transfers select reuses between G2 and G3, eventually, transforming each reuse to either G1 or G4. Finally, it evaluates the proposed strategy using a set of 10 multithreaded applications. The collected results reveal that the proposed strategy reduces parallel execution times of the tested applications between 19.3% and 33.3%.

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Software and its engineering → Compilers**.

Keywords: Data locality, Manycore architecture, Code transformation

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454069

ACM Reference Format:
Mahmut Taylan Kandemir, Xulong Tang, Hui Zhao, Jihyun Ryoo, Mustafa Karakoy. 2021. Distance-in-Time versus Distance-in-Space. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454069

1 Introduction

One of the main factors affecting application performance in different single-core and multicore architectures is "data locality", which corresponds to the fraction of "data reuses" that are caught in on-chip caches. Compiler, OS and hardware literature is full of data locality enhancement proposals and experimental evidence clearly demonstrates the success of these proposals in both single-core [14, 35, 36] and multi-core systems [4, 31, 44, 68, 69]. As a result, many of the published locality enhancement techniques have found, their ways into commercial hardware, compiler, and OS.

For a long time, data locality has been interpreted in a very specific sense that has strong ties to cache hierarchies [15, 16, 27, 51]. In this specific interpretation, a data access is said to exhibit data locality if it *hits* in a cache memory. In a similar vein, one can talk about a program exhibiting good L1 (resp. L2) locality if a majority of the program's data accesses are satisfied from L1 (resp. L2) cache. From a compiler angle, to maximize the number of cache hits, one of the main optimization targets has been reducing (ideally, minimizing) "reuse distance" – the distance between two accesses to the same data element (temporal reuse) or data block (spatial reuse). Generally speaking, the smaller this reuse distance, the higher the chances of converting that data reuse into locality, i.e., catching the reused data element/block while it is still in the cache. It is important to emphasize that, the distance we are talking about is actually the *distance-in-time* (DIT), irrespective of how we measure it – e.g., in terms of intervening loop iterations, unique data accesses, program instructions, or execution cycles.

While this specific "time centric" view of data reuse [70] served very well in the past and has been the cornerstone of numerous compiler and hardware based data locality optimizations, it is being increasingly insufficient in capturing

locality, especially with the emergence of large manycore systems. This is primarily because such memory systems typically employ multiple L2/L3 cache banks connected to one another using an on-chip network, and consequently, the ultimate latency experienced by a data access is *not* just a function of whether it hits in the cache or not, but also a function of the "physical distance" between the core that makes the request and the cache/memory component that has the requested data in it. Consider as an example two different data accesses (issued by the same core) that hit in the on-chip L2 cache, which is physically distributed (in forms of "banks") across the 2D space of a manycore chip. If the L2 bank from which the first access is satisfied is much closer to the issuing core than the L2 bank from which the second access is satisfied, the latency observed with the first access would normally be much lower than that observed with the second access, though both the accesses are technically considered to be "L2 cache hits". Similarly, two different last-level cache (LLC) misses that go to different memory banks can also experience significantly different latencies, depending on the distance between the requesting core and the memory bank that holds the requested data.

It is interesting to note that, while this "distance-from-data" (or, simply "distance") concept is crucial in shaping the performance of applications running on emerging manycores, unfortunately, it has not been thoroughly investigated so far in the literature. Further, this concept of *distance-in-space* (DIS) is entirely different from DIT – while the latter captures the intervening events in time (e.g., loop iterations, unique data accesses, instructions, or execution cycles) between two reuses of a given data element/block, the former captures the physical distance (e.g., the number of on-chip network links in a modern manycore) between two reuses.

To better highlight the difference between DIS and DIT, let us consider the simple code fragment shown in Figure 1(a). In this example, there are two uses (i.e., a reuse) of, say, $a[3]$, via reference $a[i]$ in iteration $i = 3$ and via reference $a[i-2]$ in iteration $i = 5$. In this case, DIT for this reuse is 2 (iterations), i.e., the next use of $a[3]$ is 2 iterations later from its current use, whereas, as depicted in Figure 1(b), DIS for the same reuse is 6 links. We want to make it clear at this point that, while DIT is architecture agnostic (however whether it will lead to a cache hit is not), DIS depends on the computation-to-core mapping, data-to-cache bank mapping, as well as the underlying architectural topology and network routing policies. While DIT has been investigated extensively in literature, DIS has not. We believe that, in the absence of such investigation, existing compiler and hardware based approaches to data locality, motivated primarily with the goal of reducing DIT, are being increasingly *insufficient* to capture and optimize data access latencies, especially given the fact that distances are increasing in each new generation of (larger) architecture. The overreaching goal of this paper is to fill this gap in research, by exploring the relationship and

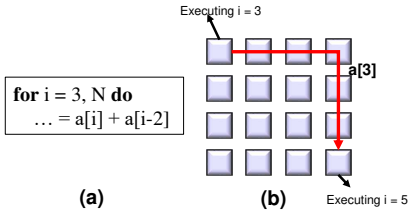

Figure 1. An example to highlight DIS and DIT.

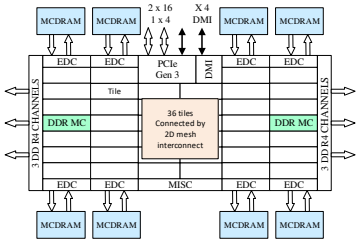

Figure 2. Knight's Landing (KNL) block diagram.

interaction between DIT and DIS. More specifically, focusing on affine programs with known loop bounds and scenarios where parallelism decisions are made at compile time (static parallelism), this paper makes the following contributions:

- It classifies a given set of data reuses (e.g., extracted from a loop nest) into four groups: G1: (low DIT, low DIS), G2: (high DIT, low DIS), G3: (low DIT, high DIS), and G4: (high DIT, high DIS). This classification is guided by two separate target thresholds, Δ_r, specifying the high/low boundary for DIT and Δ_s specifying the high/low boundary for DIS. The experimental data collected from 10 multithreaded application programs indicate that, in a 36 core single-chip architecture, G1, G2, G3 and G4 contribute, on average, to 38.0%, 13.5%, 12.8% and 35.6%, of all reuses (averaged over all loop nests of all programs).
- Observing that reuses in G1 represent the ideal case (where a data element is reused quickly – increasing chances that it will be caught in one of the on-chip caches at the time of reuse – and the distance between that cache and the requesting core is short) and there is nothing much to be done in computations in G4, it proposes a "reuse transfer" strategy that *transfers* select reuses from G2 and G3, eventually, transforming each reuse to either G1 or G4. Applying this optimization strategy leads to performance improvements (reduction in parallel execution times) ranging between 19.3% and 33.3%, averaging on 26.2%.

2 Target Architecture

In this work, we use Intel (KNL) [60] – shown in Figure 2 – as our target system, though our approach is applicable to other manycore architectures as well. More specifically, it is applicable to any multicore/manycore system (not just mesh-based ones), as long as the underlying network topology is

exposed to the compiler, as well as to multi-chip systems (see Section 5.14). The KNL CPU accommodates 36 active physical tiles (nodes), with each tile comprising two cores, with two vector processing units (VPUs) per core. There is a 1-Mbyte level-2 (L2) cache (34 nsec access latency [55]) that is shared between the two cores. Two types of memory are supported in KNL – Multi-Channel DRAM (MCDRAM) and Double Data Rate (DDR) memory. MCDRAM can be used in 3 modes: (1) cache mode, in which it is a cache for DDR; (2) flat mode, in which it is treated like standard memory in the same address space as DDR; and (3) hybrid mode, in which a portion of it is cache and the remainder is flat.

KNL employs a 2D mesh interconnect to connect the tiles, memory controllers, I/O controllers, and other components on the chip. KNL leverages a distributed tag directory to keep the coherency of its L2 caches. It supports three "cluster modes" for address space allocation: (1) all-to-all, (2) quadrant, and (3) sub-NUMA. In all-to-all, addresses are uniformly distributed. For the quadrant mode, the on-chip interconnect is partitioned into four quadrants. In the sub-NUMA mode, the mesh network is divided into four NUMA regions and accesses from both L2 and memory are limited within a subregion. While most of our experiments are collected using KNL, our approach can work with different cache management strategies as well, as long as that strategy is exposed to the compiler. That is, as long as we know how the last-level cache is managed, we can modify our reuse transfer formulation accordingly.

3 Proposed Compiler Support

3.1 Background on Affine Computations

In this paper, our focus is on affine programs, where each loop index (iterator) has affine bounds, and array subscript expressions are affine functions of enclosing loop index variables and loop-independent variables. Note however that, if a loop nest contains both affine and non-affine references, our approach still attempts to transform it considering only the affine references for data reuse optimization while making conservative dependency assumptions (regarding non-affine references). Our approach handles all affine programs that satisfy the SCoP-constraints [9], including imperfectly-nested loops. Also, the transformation-matrices that our approach explores are not restricted to unimodular transformations (in fact, in our experiments, more than 40% of the determined transformations were non-unimodular). Our approach is, however, restricted to iteration re-ordering; currently, we do not handle statement re-ordering within an iteration. Finally, our compiler also includes a limited symbolic analysis to handle non-affine references – e.g., it can determine, for a loop iterator i, $x[i^2]$ and $x[i^2+1]$ refer to different memory-locations.

Consider as an example the following loop nest:

for $i_1 = 1, N_1$ **do**
  for $i_2 = i_1, N_2$ **do**
    ... $B[i_1 + 2, i_1 + i_2 - 1]$

Here, the loop bounds can be expressed as $1 \leq i_1 \leq N_1$ and $i_1 \leq i_2 \leq N_2$. On the other hand, the array access shown can be expressed as $Lt + o$, where $L = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $t = \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$, and $o = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$. When there is no confusion, we will use $\{L, o\}$ to denote this array access. When a given loop nest contains a mix of affine and non-affine references (as most loops in our tested programs), our approach focuses only on the affine ones.

A linear loop transformation represented by matrix T transforms results in a different scheduling of the iterations of the loop it is applied to.[1] For example, if, before the transformation, we have $t = \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$, after the transformation represented by $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, the new loop indices can be expressed as $t' = \begin{pmatrix} i'_1 \\ i'_2 \end{pmatrix} = Tt = \begin{pmatrix} i_2 \\ i_1 \end{pmatrix}$, that is, i'_1 and i'_2 are the new loop indices. After this transformation, both loop bounds and array subscript expressions can be re-written as follows by using Fourier-Motzkin Elimination (FME) [21]:

for $i'_1 = 1, N_2$ **do**
  for $i'_2 = 1, min(i'_1, N_1)$ **do**
    ... $B[i'_2 + 2, i'_2 + i'_1 - 1]$

For a given array A, Φ_a is used the denote the set of references to it – each member of Φ_a can be represented $\{L_q, o_q\}$, where L_q and o_q are as defined above.

For a given loop nest, we use matrix N to denote the iteration-to-core assignment. Specifically, for a given loop iteration t, Nt gives the core it is mapped to. In other words, Nt gives the ID of the node to which computation (loop iteration) t is assigned for execution. Since our target architecture is two-dimensional, N is a matrix of $2 \times n$, where n is the number of loops in the loop nest being optimized. Note that entries of N include mod (core count) operations, to map a large iteration space into a much smaller core space. Further, our approach can work with any N function that maps iterations to cores (it is input to our approach and determining a good N is not a goal of this paper). In fact, instead of having an explicit N, our approach can directly accept iteration-to-core mapping constraints in our formulation.

3.2 DIT and DIS

Next, we formally define DIT and DIS. Let $\{L_1, o_1\} \in \Phi_a$ and $\{L_2, o_2\} \in \Phi_a$ two references to array A[2]. Let us assume, for two iterations t_1 and t_2 ($t_1 < t_2$), we have $L_1 t_1 + o_1 = L_2 t_2 + o_2$,

[1] While in this work we mainly focus on loop transformations, data transformations can also be potentially used for reuse-transfer, either standalone or in conjunction with loop transformations.
[2] It is possible that $L_1 = L_2$.

that is, the same data element is accessed by both $\{L_1, o_1\}$ and $\{L_2, o_2\}$, i.e., it is *reused.* In this case, $t_2 - t_1$ is referred to as the *reuse distance in time* (DIT).[3] Let N denote a matrix that maps a given loop iteration to a location in the 2D grid. Assuming that $l_1 = Nt_1$ and $l_2 = Nt_2$ (l_1 and l_2 are called location vectors), in this work, $l_2 - l_1$ is termed as the *reuse distance in space* (DIS). The operation "−" calculated the Manhattan Distance between two location vectors.

Before proceeding any further, let us explain $t_2 - t_1$ and $l_2 - l_1$ in more detail. $t_2 - t_1$ represents the time difference (measured in terms of loop iterations) between two successive accesses to the same data element. The higher this distance, the lower the chances that the data will be caught while it is in one of the on-chip caches in the manycore system. In comparison, $l_2 - l_1$ represents the *physical distance* (measured in terms of the *number of communication links* – hops – in the two-dimensional mesh) between two successive reuses of the same data element. Note that, the larger this distance, the higher latency in the second access (one occurring at iteration t_2) would experience. This higher latency is due to two main reasons: firstly, more links to travel mean higher latency, and secondly, more links also mean higher chances for contention on the on-chip network. As an example, let us assume that, two iterations, $(2\ \ 3)^T$ and $(4\ \ 3)^T$, access the same data element and these iterations are mapped to cores $(1\ \ 1)^T$ and $(3\ \ 5)^T$, respectively. That is, when executing iteration $(2\ \ 3)^T$, core $(1\ \ 1)^T$ accesses the data element, and then later, when executing iteration $(4\ \ 3)^T$, core $(3\ \ 5)^T$ accesses the same data element. In this case, the time difference between the two reuses (DIT) is $(4\ \ 3)^T - (2\ \ 3)^T = (2\ \ 0)^T$,[4] whereas the spatial difference (DIS) is $(3\ \ 5)^T - (1\ \ 1)^T = (2\ \ 4)^T$.

3.3 Classification of Data Reuses

Based on these temporal and spatial distances of reuse,[5] we can classify reuses in a loop nest [6] into four disjoint groups:

- **G1:** $\forall t_1, t_2, l_1,$ and l_2 involved in a data-reuse : $t_2 - t_1 \leq \Delta_r$ and $l_2 - l_1 \leq \Delta_s$
- **G2:** $\forall t_1, t_2, l_1,$ and l_2 involved in a data-reuse : $t_2 - t_1 \leq \Delta_r$ and $l_2 - l_1 > \Delta_s$
- **G3:** $\forall t_1, t_2, l_1,$ and l_2 involved in a data-reuse : $t_2 - t_1 > \Delta_r$ and $l_2 - l_1 \leq \Delta_s$
- **G4:** $\forall t_1, t_2, l_1,$ and l_2 involved in a data-reuse : $t_2 - t_1 > \Delta_r$ and $l_2 - l_1 > \Delta_s$

[3] As mentioned earlier, this distance concept has taken a lot attention in the past, and in most compiler-based works, it is customary to measure it in terms of "loop iterations".

[4] Assuming that the inner loop iterations K times, this corresponds to $2K$ iterations.

[5] Our term "spatial distance of reuse" is not to be confused with the conventional concept of "spatial reuse", which is essentially a temporal reuse at a cache block granularity.

[6] For now, we focus on one loop nest a time. Our proposed approach can consider multiple loop nests at the same time.

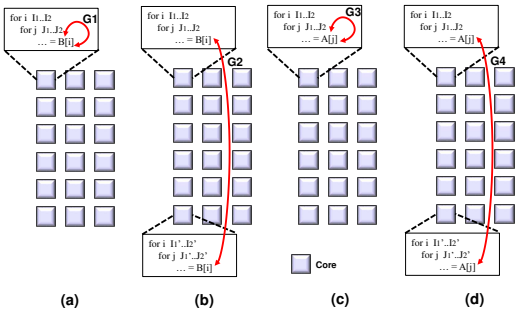

Figure 3. An example illustrating the four groups of reuses.

Here, Δ_r represents a "threshold" that draws the boundary between short and long "temporal distance", and Δ_s represents a "threshold" that draws the boundary between short and long "spatial/physical distance". Later we discuss how we can select suitable values for Δ_r and Δ_s. These four groups (types) of reuses are illustrated in Figure 3.

It is important to note that the reuses in the G1 category represent the ideal case where both temporal distance (DIT) and spatial distance (DIS) are short, and at the opposite end, G4 represents the worst case where both temporal and spatial distances are long. In comparison, the reuses in G2 and G3 are good from one aspect (temporal or spatial distance) and bad (not preferable) from the other. Based on this, our optimization strategy presented and evaluated in the rest of this paper, which targets reducing both temporal and spatial reuse distances, is built upon the following philosophy:

We do not want to change the reuses in G1 as they are already good, and changing the reuses in G4 is not going to be trivial as it cannot be done without affecting (distorting) reuses in other groups. Considering G2 and G3 however, we may want to transfer reuses (by transforming the loop iterations as explained below) between them such that the reuses are mapped (after the transformation) to either G1 or G4, so that at least some of them (those that are mapped to G1) will benefit the execution. Clearly, in the ideal case, we want to maximize the number of reuses moved to G1.

This strategy is based on an observation that will be quantified later (Figures 13 and 14): if the the compiler indiscriminately tries to move each and every data reuse to the G1 category, it ends up having too many constraints to satisfy, and as a result, many computations/iterations go untransformed (unoptimized), as it cannot find a suitable transformation matrix for them. Instead, our approach performs a tradeoff by transferring some reuses to G1 and some others to G4.

3.4 Reuse Transfer

Figure 4 depicts the high level view of the concept of "reuse transfer", the strategy proposed in this work to improve performance. Let us first explain why the reuses in categories G2 and G3 may not be very beneficial as far as data locality and application performance are concerned. In the case of

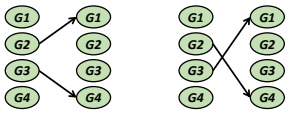

Figure 4. Reuse transfer from G2/G3 to G1/G4.

G2 where DIT is short and DIS is long, the reused data is not likely to arrive at the requested core at the time of reuse, as illustrated Figure 3(b). That is, when the data access request is issued by core, the data has still a long way to come. In contrast, reuses in G3 exhibit a different type of problem. Specifically, in this case, DIS is short (as shown in Figure 3(c)), but this (nearby) data will not be needed for a long time (as DIT is long). Note that this nearby location could instead be used for some other data, one with a shorter reuse in time. Now, considering these two types of reuses (one belonging to G2 and the other one belonging to G3), we ask the question of whether one can create two different reuses – one in G1 and one in G4. In this way, we can expect good performance from the (newly created) reuse in G1.

It is important to emphasize that "reuse transfer" is implemented in this work by using computation (loop) transformations. While one can potentially use also data layout transformations (i.e., changing the layout of data arrays in memory) to create a similar impact of reuse transfer, since layout transformation requires in general complex whole program analysis, we postpone its investigation in the context of DIT+DIS optimization to a future study.

3.5 Problem Formulation

Let us start by formulating the conditions that need to be satisfied for a reuse to belong to groups G1-G4. In the following formulation, we use t_i to denote a **loop iteration vector** and l_j to represent the **location vectors** in the 2D grid. We assume that t_i and l_j used below are within their respective bounds; so, we do not list the boundary conditions explicitly. Also, L_k and o_k correspond to an access matrix and offset vector for a reference to the array whose elements are reused. Finally, as before, N is used to represent the mapping from iteration space (vectors) to 2D grid.

In mathematical terms, for G1, we have:

$$L_1 t_1 + o_1 = L_2 t_2 + o_2 \quad (1)$$
$$t_2 - t_1 \leq \Delta_r \quad (2)$$
$$l_2 - l_1 \leq \Delta_s \quad (3)$$
$$l_1 = N t_1 \quad (4)$$
$$l_2 = N t_2 \quad (5)$$

It is important to note that, the first constraint here ensures the "existence of data reuse" (that is, two references access the same memory location), whereas the next two constraints guarantee that the reuse belongs to G1. Finally, the last two constraints establish the connection between iteration space and core space (in two-dimensional manycore architecture). Similarly, for G2, we have the following constraints:

$$L_1 t_1 + o_1 = L_2 t_2 + o_2 \quad (6)$$
$$t_2 - t_1 \leq \Delta_r \quad (7)$$
$$l_2 - l_1 > \Delta_s \quad (8)$$
$$l_1 = N t_1 \quad (9)$$
$$l_2 = N t_2 \quad (10)$$

The corresponding constraints for G3 are:

$$L_1 t_1 + o_1 = L_2 t_2 + o_2 \quad (11)$$
$$t_2 - t_1 > \Delta_r \quad (12)$$
$$l_2 - l_1 \leq \Delta_s \quad (13)$$
$$l_1 = N t_1 \quad (14)$$
$$l_2 = N t_2 \quad (15)$$

And finally, we have the following constraints for G4:

$$L_1 t_1 + o_1 = L_2 t_2 + o_2 \quad (16)$$
$$t_2 - t_1 > \Delta_r \quad (17)$$
$$l_2 - l_1 > \Delta_s \quad (18)$$
$$l_1 = N t_1 \quad (19)$$
$$l_2 = N t_2 \quad (20)$$

Clearly, the third constraints for G2 and G3 can be rewritten as

$$N(t_2 - t_1) > \Delta_s \quad \text{and} \quad N(t_2 - t_1) \leq \Delta_s,$$

respectively. Let us assume now that the corresponding iteration space is transformed using a loop transformation matrix T. We focus on two different reuses – (t_1, t_2) belonging to G2 and (t_3, t_4) belonging to G3. After the transformation, we would want to see either **(i)** the reuse in G2 is transformed to G1 and the reuse in G3 is transformed to G4 (as shown in Figure 4(a)), or **(ii)** the reuse in G2 is transformed to G4 and the reuse in G3 is transformed to G1 (as shown in Figure 4(b)). The constraints that capture the first possibility can be expressed as follows:[7]

$$L_1 T^{-1} t_1 + o_1 = L_2 T^{-1} t_2 + o_2 \quad (21)$$
$$T(t_2 - t_1) \leq \Delta_r \quad (22)$$
$$NT(t_2 - t_1) \leq \Delta_s \quad (23)$$
$$L_3 T^{-1} t_3 + o_3 = L_4 T^{-1} t_4 + o_4 \quad (24)$$
$$T(t_4 - t_3) > \Delta_r \quad (25)$$
$$NT(t_4 - t_3) > \Delta_s \quad (26)$$

And, similarly, for the second possibility, we have:

$$L_1 T^{-1} t_1 + o_1 = L_2 T^{-1} t_2 + o_2 \quad (27)$$
$$T(t_2 - t_1) > \Delta_r \quad (28)$$
$$NT(t_2 - t_1) > \Delta_s \quad (29)$$
$$L_3 T^{-1} t_3 + o_3 = L_4 T^{-1} t_4 + o_4 \quad (30)$$
$$T(t_4 - t_3) \leq \Delta_r \quad (31)$$
$$NT(t_4 - t_3) \leq \Delta_s \quad (32)$$

If we can find a T that satisfies either the first group of constraints above or the second group, this means that we

[7] Note that $Tt_j - Tt_i = T(t_j - t_i)$, as T is non-singular.

transfer either 1) one reuse from G2 to G1 and one reuse from G3 to G4, or 2) one reuse from G3 to G1 and one reuse from G2 to G4.

Of course, any such T should also satisfy data dependencies. More specifically, if there is a dependence from iteration t_1 to iteration t_2, i.e., $t_1 \longrightarrow t_2$, after the transformation $Tt_1 \longrightarrow Tt_2$ should hold. In other words, if $d = t_2 - t_1$ is the dependence vector, after the transformation T, Td should be lexicographically positive. We refer to this as dependence constraint. Since it needs to be satisfied in all loop transformations considered in this work, we will not show it explicitly in the remainder of the paper.

Let us now consider the $\{(G2 \rightarrow G1), (G3 \rightarrow G4)\}$ reuse transfer in more detail. First, from Expressions (8-10), we have $N(t_2 - t_1) > \Delta_s$, which gives us $(t_2 - t_1) > N^{-1}\Delta_s$. Considering this along with Expression (7), we can obtain $N^{-1}\Delta_s < (t_2 - t_1) \leq \Delta_r$. Similarly, from Expressions (13-15), we can derive $(t_4 - t_3) \leq N^{-1}\Delta_s$, and combining this with Expression (12) gives us $\Delta_r < (t_4 - t_3) \leq N^{-1}\Delta_s$. Now, after transformation T, based on Expressions (22-23), we have $T(t_2 - t_1) \leq min\{\Delta_r, N^{-1}\Delta_s\}$ and, based on Expressions (25-26), we have $T(t_4 - t_3) > max\{\Delta_r, N^{-1}\Delta_s\}$. Putting these four newly derived constraints together,

$$N^{-1}\Delta_s < (t_2 - t_1) \leq \Delta_r \tag{33}$$

$$\Delta_r < (t_4 - t_3) \leq N^{-1}\Delta_s \tag{34}$$

$$T(t_2 - t_1) \leq min\{\Delta_r, N^{-1}\Delta_s\} \tag{35}$$

$$T(_4{-}t_3) > max\{\Delta_r, N^{-1}\Delta_s\} \tag{36}$$

gives us the conditions that need to be satisfied to have the $\{(G2 \rightarrow G1), (G3 \rightarrow G4)\}$ reuse transfer. The next subsection discusses our solution strategy for this system of constraints, and gives our formal compiler algorithm that implements that strategy. Note that expressions similar to Expressions (33-36) can be dereived for the reuse transfer $\{(G2 \rightarrow G4), (G3 \rightarrow G1)\}$ as well.

Discussion: At this point, one may ask why we only focus on reuse transfers $\{(G2 \rightarrow G1), (G3 \rightarrow G4)\}$ and $\{(G2 \rightarrow G4), (G3 \rightarrow G1)\}$, and do not consider other possibilities, such as $\{(G2 \rightarrow G1), (G3 \rightarrow G1), (G4 \rightarrow G1)\}$. There are two main reasons for this. First, there is a limit on how much reuse distance in space and reuse distance in time can be exploited. For example, as we demonstrate later in our experimental evaluations, trying to move all data reuses to the G1 category creates too many "conflicts" in finding loop transformation (T) matrices, and as a result, a large fraction of the original loop iterations remain untransformed (unoptimized). Second, from an optimization viewpoint, G2 and G3 are the best categories to target, as they can be transitioned to G1 by improving only one of the metrics (either DIT or DIS). In comparison, there is no point in trying to transfer the reuses in the G1 category to other categories, and reuses in G4 are hard to transfer in general, as in order to do so, one needs to improve both DIT and DIS (i.e., it is very difficult to find loop transformations to improve DIT and DIS as the same time, if both are not good to begin with).

3.6 Solution Strategy and Compiler Algorithm

Based on the formulation above, the goal is to find a T, a loop transformation matrix, such that one of the two scenarios identified in the previous subsection is satisfied, for t_1, t_2, t_3 and t_4. It is to be noted however that, in a given loop nest, there are typically lots of iterations (and consequently lots of data reuses) and, if we write inequalities for all iterations, the resulting system can be *overly constrained*. Further, there is the question of how to determine suitable Δ_r and Δ_s values to target. Clearly, depending on the values of Δ_r and Δ_s, the membership of reuses to groups (G1-G4) can change.

Below, we first discuss our proposed approach at a high level, and then give a detailed compiler algorithm that implements it. As discussed earlier in Section 3.4, our approach is based on the concept of "reuse transfer", which transfers reuses from G2 and G3 to G1 and G4. Let us assume, for now, that both Δ_r and Δ_s are given a priori. Later in experimental evaluation we study the impact of different Δ_r/Δ_r values.

Our approach starts by identifying all data reuses in the target loop nest and assigning them to four groups described above (G1, G2, G3 and G4). After this partitioning, the rest of our approach focuses only on the iterations that are involved in the reuses mapped to G2 and G3 but *not* mapped to G1 and G4. The reason behind this is to ensure that we do *not* disturb the reuses that are originally mapped to G1 and G4. Now, let Q denote the set of iterations that are involved in the reuses mapped to G2 and G3, but not mapped to G1 and G4. Further, we use symbol R_k to denote that set of reuses that are originally in group Gk.

Next, we select a reuse $< t_1, t_2 >$ from G2 and a reuse $< t_3, t_4 >$ from G3, and determine a loop transformation matrix (using Fourier-Motzkin Elimination (FME)). FME [21] is a technique that is used to solve a system of inequalities (note that a given equality can always be converted to two inequalities, i.e., $a = b$ means $a \leq b$ and $b \leq a$). FME employs a method to reduce the dimension of a linear system of inequalities by one without changing feasibility. By keeping reducing the dimension one by one, we eventually reach the one-dimensional case, for which we can easily test feasibility. Note that T maps either 1) $< t_1, t_2 >$ to G1 and $< t_3, t_4 >$ to G4, or 2) $< t_1, t_2 >$ to G4 and $< t_3, t_4 >$ to G1. Since this system is not overly constrained, it is very likely to find a T. Then, we use this T to transfer "as many data reuses as possible" from G2 and G3 to G1 and G4 (more specifically, at each step, we either move one reuse from G2 to G1 and one reuse from G3 to G4, or, one reuse from G3 to G1 and one reuse from G2 to G4). In mathematical terms, as we transfer reuses, we also drop the corresponding loop iterations from Q; that is, at any given point, Q contains the loop iterations "yet to be transformed". When none of the remaining reuses in G2 and G3 can be transferred using this T anymore, we

Algorithm 1 Algorithm of Reuse Transfer.

INPUT: Nested Loop (D); Temporal Threshold Δ_r; Spatial Threshold Δ_s; Iteration-to-core Assignment (N);
OUTPUT: $G1$, $G2$, $G3$, and $G4$
1: $G1$, $G2$, $G3$, $G4$, G_{temp2}, *and* $G_{temp3} \leftarrow \emptyset$
2: // Initial partition of loop nest D
3: <$G1$, $G2$, $G3$, $G4$> $\leftarrow$ $Reuse\_Identifying(D)$
4: **while** $G2 \neq \emptyset$ and $G3 \neq \emptyset$ **do**
5: $< t_i, t_j >$ $from$ $G2$, $< t_m, t_n >$ $from$ $G3$
6: // Find the transformation matrix
7: $T \leftarrow Fourier\_Motzkin\_Elimination(N, < t_i, t_j >, < t_m, t_n >, \Delta_r, \Delta_s)$
8: **for** $< t_l, t_k >$ *in* $G1$ **do**
9: **if** T transfers $< t_l, t_k >$ to $G2$, $G3$, or $G4$ **then**
10: // Drop T and continue to find another T
11: $T \leftarrow NULL$
12: **for** t_p, t_q, t_r, t_s that have been transferred by any T **do**
13: **if** There exist dependency between t_p, t_q, t_r, t_s and t_i, t_j **then**
14: $d_a = t_i - t_p$, $d_b = t_q - t_i$, $d_c = t_j - t_r$, and $d_d = t_s - t_j$
15: **if** Any of Td_a, Td_b, Td_c, and Td_d is not lexicographically positive **then**
16: $T \leftarrow NULL$
17: Conduct the same dependence check for t_m, t_n
18: **if** $T \neq NULL$ **then**
19: Select and remove $< t_i, t_j >$ from $G2$ and $< t_m, t_n >$ from $G3$
20: Add $T < t_i, t_j >$ to $G1$ (or $G4$) and Add $T < t_m, t_n >$ to $G4$ (or $G1$)
21: **while** $G2 \neq \emptyset$ and $G3 \neq \emptyset$ **do**
22: Select and remove $< t_i, t_j >$ $from$ $G2$, $< t_m, t_n >$ $from$ $G3$
23: **if** $T < t_i, t_j >$ $belongs\ to$ $G1$ (or $G4$) and $T < t_m, t_n >$ $belongs\ to$ $G4$ (or $G1$) **then**
24: Add $T < t_i, t_j >$ to $G1$ (or $G4$) and Add $T < t_m, t_n >$ to $G4$ (or $G1$)
25: **Continue**
26: $G_{temp2} \leftarrow G_{temp2} \cup < t_i, t_j >$, $G_{temp3} \leftarrow G_{temp3} \cup < t_m, t_n >$,
27: $G2 \leftarrow G2 \cup G_{temp2}$ and $G3 \leftarrow G3 \cup G_{temp3}$
28: Output $G1$, $G2$, $G3$, and $G4$

select another loop transformation matrix and repeat the steps above (i.e., transferring as many reuses as possible from G2 and G3 to G1 and G4 and dropping the corresponding iterations from Q). This process is repeated until either G2 or G3 becomes empty, or we cannot find any transformation that can transfer the remaining reuses from G2/G3 to G1/G4.

We want to emphasize that, this approach is distinguished from the previous approaches in at least two important ways. First, unlike the previous approaches, our strategy considers *both* reuse in time and reuse in space. Second, as opposed to the existing approaches, ours can transform the different iterations of a given loop nest using different transformation matrices (whereas in existing approaches a single transformation matrix is used for the entire loop nest).

The pseudo-code for our compiler algorithm is given in Algorithm 1. It takes four input parameters including a loop nest (D), the temporal distance threshold (Δ_r), the spatial distance threshold (Δ_s), and the iteration to core mapping (N). The output of our algorithm is the four groups of loop iterations after applying our reuse transfer optimization. At the beginning, we identify all the reuse opportunities inside a loop nest, to form the initial four sets $G1$, $G2$, $G3$, and $G4$ (line 2). Then, we pick up reuses from $G2$ and $G3$ and use FME to find a transformation matrix T (line 7). Note that, we only choose a *legal* transformation matrix T that does not violate any dependencies in the loop body. More specifically, if there is a data reuse r (between iterations t_1 and t_2) which belongs to category Gk where $k \in \{2, 3\}$, we transfer it, using T, to a category Gl where $l \in \{1, 4\}$ only if, for all dependences $d_a = t_1 - t_p$, $d_b = t_q - t_1$, $d_c = t_2 - t_r$, and $d_d = t_s - t_2$, where t_p, t_q, t_r, and t_s have already been transformed (via, potentially, other transformation matrices), Td_a, Td_b, Td_c, and Td_d are lexicographically positive. This means that, when we are transforming t_1 and t_2 using a transformation matrix, we ensure that all dependences that involve t_1 or t_2 (d_a, d_b, d_c, and d_d) are preserved. Note also that, we choose/use the calculated T *only if it does not hurt the reuses which have been already captured in* $G1$ (lines 8 to 11). Otherwise (i.e., if this T cannot transfer any remaining reuses from $G2$ or $G3$ to other groups without affecting any reuse in $G1$), it means we are done with this T, and use other reuses from $G2$ and $G3$ to find other transformation matrices. This step guarantees that the sizes of $G2$ and $G3$ (i.e., q) are decreasing monotonically. Then, we use T to eliminate as many reuses as possible from $G2$ and $G3$ (lines 21 to 26). The asymptotic complexity of this algorithm is $O(PQ^2)$, where P is the total number of reuses in $G1$, and Q is the total number of reuses in $G2$ and $G3$ combined. Since we calculate/evaluate transformation matrix for each reuse pair from $G2$ and $G3$, and each transformation matrix T is checked against the reuses in $G1$ in case it affects the reuses in $G1$.

3.7 Implementation

We wrote a new compiler pass for manipulating integer tuple relations and sets (represented by Presburger Formulas [29]), and added it to LLVM [43]. This pass analyzes a given loop nest, identifies all temporal and spatial data reuses, and determines the loop transformations (T matrices) needed to transfer data reuses from groups G2 and G3 to groups G1 and G4. As output, our pass produces sub-loop nests, each with its own sub-iteration space. This code generation is possible as, using Presburger Formulas, we are able to represent a given non-convex set (iteration space) as a union of two or more convex sets. Clearly, using multiple transformations for a given loop nest makes rewriting the transformed code quite challenging. Our code rewriting phase generates code to scan the points in the union of a number of convex sets (each corresponding to one of the loop transformation matrices found). The result for a given input nest is, in most cases, a large, imperfectly-nested output loop nest. Also, while we performed experiments targeting an Intel KNL type of system, our approach is applicable to any manycore architecture whose data access protocol (e.g., SNUCA vs DNUCA) and underlying topology are exposed to the compiler.

4 Experimental Setup

Platforms: Our experiments are carried out using both a detailed multicore/manycore simulator [11] and a 2D mesh-based manycore system [60]. The simulation-based experiments are mainly performed to collect fine-grain execution statistics (that could not be collected from the execution on

Table 1. System setup.

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

an actual manycore system) to show the effectiveness of our approach, as well as to conduct sensitivity experiments with respect to various architectural parameters. Table 1 gives the simulation parameters as well as the default values.

The proposed compiler support is implemented in LLVM version 9.0.0 [43], using it as a source-to-source translator. LLVM is a collection of modular and reusable compiler and tool-chain technologies. It provides compiler designers with a state-of-the-art SSA (static single assignment) based compilation strategy capable of supporting both static and dynamic compilation of various programming languages. The resulting optimized source codes are compiled using the node compilers (*gcc* in the simulator and *icc* in the commercial system), to generate the executable code. Most of the results presented below are simulation results; the results collected from Intel manycore machine are explicitly indicated.

Benchmark Programs: In most of our experimental evaluation, we use a set of ten multithreaded benchmark programs from PARSEC [10] to test our proposed compiler scheme. We could not execute the original versions of three PARSEC benchmarks – *bodytrack*, *freqmine*, and *swaptions* – in our simulator, so we do not consider them in our evaluations. Note that, the input dataset sizes of these applications are increased from their original values to put pressure on the cache and memory systems. More specifically, in *blocksholes*, we tripled the number of options; in *canneal*, the number of elements is increased to 1,600,000; in *dedup*, the input dataset size is increased to 1.2 GB; in *facesim*, the number of triangular pyramids is increased to 1,244,826 and the number of frames is increased to 32; in *ferret* the number of queries and number of images are increased to 4,096 and 164,088, respectively; in *fluidanimate*, the number of frames and number of particles are increased to 20 and 18M, respectively; in *raytrace*, we used 1,920 × 1,080 pixels (HDTV resolution), 10 million polygons, and 200 frames; in *streamcluster*, the number of points per block is set to 524,288; and finally, in *vips* and *x264*, the number of images and number of frames are increased, respectively (to 128 and 8K). The resulting datasets vary between 33.1 MB and 1.8 GB, and LLC (L2) cache miss rates range between 18.1% and 39.3% (in the simulator).

The detailed information on these benchmark programs can be found in [10].

Different Versions Tested: For each of the ten benchmark programs we have, we performed experiments with six different versions:

- **Baseline:** This is the default version. In this version, the iteration space of a loop nest is divided into p chunks, each containing a set of N/p consecutive iterations (p is the total core count and N is the total number of iterations), and each chunk is scheduled on a separate core. This baseline version already has slightly better performance than the output codes generated by two compilers – Polly [30] and Pluto [13] – as it includes a larger suite of conventional data locality optimizations. Unless otherwise stated, the results with the remaining versions are normalized with respect to this version. Later in our evaluation, we also perform experiments with several variations of this baseline scheduling.
- **Reuse Transfer:** This represents the optimization strategy presented and defended in this paper. As discussed in detail, it is based on "transferring" data reuses across different groups.
- **DIT Only:** This is a representative of traditional compiler approaches that focus on minimizing only reuse distance in time. Many previously published compiler works fall into this category, and the specific approach that we used is based on [33].
- **DIS Only:** This approach optimizes DIS aggressively, even if doing so sacrifices some DIT optimizations opportunities. Essentially, this version mimics the approach proposed in [67].
- **DIS+DIT:** In this version, we first apply DIT Only and then DIS Only. The reason why we perform experiments with this version is to demonstrate that, our proposed approach, which is based on reuse transfer, is more than just a combination of DIT and DIS.
- **Limit:** This version is designed to measure the limits of distance in time and distance in space optimization. specifically, it makes 100 experiments with DIS+DIT, each time starting with different iterations to transform, and selects the one that generates the best result. Although this does not guarantee optimality, we have not observed any case where more than 100 experiments generated a better result. A comparison of its performance against that of our proposed reuse transfer based strategy is given later.

Note that, while our simulation experiments used all six versions of each benchmark program, the experiments on

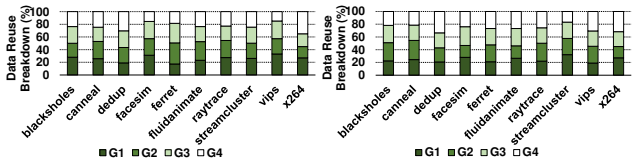

Figure 5. Distribution of data reuses across four categories (G1-G4) in the baseline. (a) temporal reuses and (b) spatial reuses.

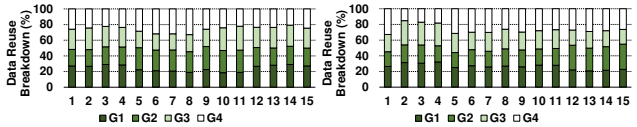

Figure 6. Distribution of reuses across four categories in 15 consecutive execution phases for two of our benchmark programs. (a) *blacksholes* and (b) *x264*.

Intel manycore system used only the first five, as the last version (Limit) can be implemented only in the simulator. We also want to emphasize that, all the versions (including the baseline) are compiled using the highest level of optimizations and have the same degree of parallelism, i.e., in all versions, the same set of computations are executed in parallel. As a result, these versions differ only in how computations are mapped to cores.

Also, unless otherwise stated, all experiments are conducted assuming the target DIT and DIS values of $\Delta_r = (0\ 0\ 0\ \ldots\ 0\ 1)^T$ and $\Delta_s \in \{(0\ 1)^T, (0\ -1)^T, (1\ 0)^T, (-1\ 0)^T\}$, respectively. Note that, this default Δ_r target indicates the best possible reuse in time (i.e., the shortest reuse distance, which means that the reused data element/block is accessed in two successive loop iterations [8]), and that Δ_s represents 1 link (hop) distance (in any of the four possible directions in the two-dimensional mesh network). Later in our evaluations we also report results with different Δ_r and Δ_s targets.

Finally, in our experiments, we used the quadrant cluster mode (we observed that the difference of original executions between the quadrant and sub-NUMA mode was quite small).

5 Experimental Results

5.1 Reuse Classification Results for Baseline

We start by presenting the distribution of data reuses in the original versions of our benchmark programs (baseline). The plots in Figure 5(a) and (b) give the distribution of temporal and spatial data reuses, respectively, into four categories (G1 through G4) when considering the entire execution, and Figures 6(a) and (b) show the variations of the reuses across time for two of our application programs – *blacksholes* and *x264*. Note that each of the plots in Figures 6(a) and (b) shows a representative execution segment that includes 15 *consecutive* phases. One can make two main observations from these plots in Figure 5 and Figure 6. First, all benchmark programs exhibit a variety of (somewhat balanced) reuses. The distributions of temporal and spatial data reuses across four reuse categories (G1-G4) are similar. And secondly, the reuse type exhibits a somewhat consistent behavior: neighboring execution phases have similar distributions, with variations among distributions corresponding mostly to loop nest boundaries.

[8] This is what many traditional data locality optimizers target to achieve.

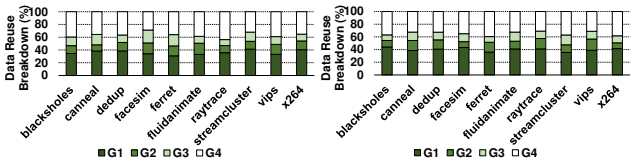

Figure 7. Distribution of data reuses across four categories (G1-G4) after reuse transfers. (a) temporal reuses and (b) spatial reuses.

Table 2. Compile-time statistics.

Benchmark	Number of Ts	Transformed Data Accesses	Increase in Code Size	Increase in Compilation Time
blacksholes	{2,6,14}	88.2%	1.4x	28%
canneal	{4,9,12}	93.2%	1.2x	16%
dedup	{5,10,18}	90.7%	1.7x	21%
facesim	{5,8,12}	84.4%	2.2x	31%
ferret	{6,9,21}	86.1%	1.9x	19%
fluidanimate	{2,2,2}	94.3%	2.4x	15%
raytrace	{2,6,14}	81.8%	1.5x	27%
streamcluster	{1,2,4}	86.7%	2.4x	29%
vips	{2,3,4}	95.1%	1.8x	26%
x264	{2,5,8}	93.4%	2.1x	27%

5.2 Reuse Classification Results

Figures 7(a) and (b) plot the same reuse distribution results, but this time with the compiler-optimized version (the reuse transfer based version). Comparing these plots with the corresponding plots in Figure 5 clearly reveals that the proposed compiler-based optimization transfers a lot of reuses from groups G2 and G3 to groups G1 and G4, for *both* the temporal and spatial data reuses. More specifically, in the case of temporal data reuses, the total contribution of G2+G3 in the baseline case was 50.9% on average. After our optimization, on the other hand, the total contribution of G2+G3 averages 27.5%, showing clearly that our approach does what it is designed to do. The corresponding numbers in the case of spatial data reuses are 49.8% (before optimization) and 30.6% (after optimization). In both cases, there is a corresponding increase – as a result of our optimization – in the contribution of the reuses in the G1 and G4 categories.

5.3 Compile-Time Statistics

Before presenting detailed runtime statistics, we first give data showing the behavior of our compiler-based approach.

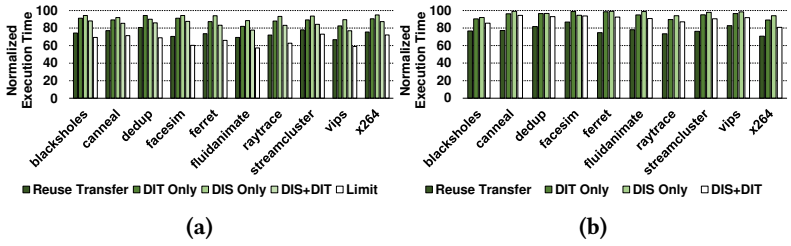

Figure 8. Normalized execution times (with respect to Baseline). (a) Simulation results, and (b) Intel KNL results.

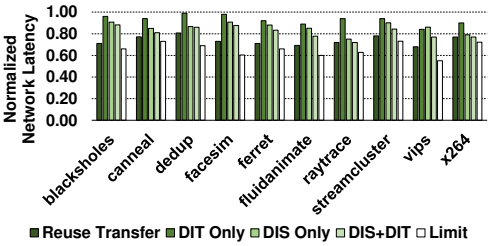

Figure 9. Normalized network latency values with different strategies.

The second column for each benchmark program in Table 2 gives the {minimum, average, maximum} number of loop transformation matrices (i.e., number of Ts) used for the loop nests in the benchmark programs (across all loop nests). It can be observed from this column that, some (original) iteration spaces (loop nest) are divided into as many as 21 sub-iteration spaces (each transformed using its own T), indicating that our approach successfully customizes loop transformation matrices (Ts) based on the loop nest being optimized. The third column of the table gives the percentage of data accesses that got optimized by our approach, averaged over all loop nests in the program. It can be observed from these results that, on average, more than 80% of data accesses of a loop nest got transformed by our approach using a transformation matrix, indicating that our approach is able to optimize most of the data accesses in these benchmarks. Note that, the main reason for our approach not being able to optimize a data access is because it is not affine. The fourth column on the other hand gives the increase in executable code size (over Baseline) when our reuse transfer based approach is employed. Finally, the last column shows the increase in compilation time, again over the baseline versions. We see from these last two columns that, as expected, our approach increases both the executable size and compilation time. We want to mention however that, when our approach is used, the largest compilation time we observed during the experiments was still less than 1 minute.

5.4 Execution Time Results

Figure 8a plots the normalized execution times (with respect to Baseline; lower, better), collected from our simulator, with different versions. One can make several critical observations from this plot. First, for all ten benchmark programs, our proposed optimization approach, which is based on reuse transfer, outperforms the remaining versions, except of course Limit. In fact, the average performance improvements (reduction in parallel execution time) brought by our approach, DIT Only, DIS Only, DIS+DIT and Limit are, respectively, 26.2%, 11.4%, 7.5%, 15.9%, and 33.9%. The reason why our approach performs better than DIT Only is that the latter does not do anything specific to reduce the distance (in terms of the number of communication links) between two successive reuses of the data. To illustrate this point, we present in Figure 9 the average (on-chip) network latency values of different versions, collected from the simulator and normalized to Baseline (lower, better). It is clear from these results that, the data assesses in DIT Only spends considerably more time in the on-chip network than those in our reuse transfer based strategy. However, trying to optimize DIS without paying attention to DIT is also not the best option, mainly because keeping data close by for long durations of time prevents other data (with shorter reuse in time) to be placed nearby.

Furthermore, it is interesting to observe that even DIS+DIT does *not* perform as well as our reuse transfer based strategy. The main reason for this is that, aggressively optimizing first for DIT, limits the extent/potential of DIS optimization. In particular, DIT tries to optimize the reuses in G4 as well (in addition to those in G3), and this, in turn, reduces some opportunities for transferring, later, some reuses from categories G2/G3 to categories G1/G4. We present in Figure 10 the contribution of G2+G3 to all data reuses in Baseline, our defended strategy, and DIS+DIT. It can be clearly seen that the reuse transfer based strategy transfers many more reuses, compared to DIS+DIT, from G2/G3 to G1/G4.

So far, all the results presented have been collected using our cycle-accurate simulator. Figure 8b on the other hand presents the normalized execution time results, collected from Intel manycore architecture (Xeon Phi 7290). It can be observed that these results are quite similar to those collected from the simulator. In particular, our approach reduces execution times (compared to the baseline version) between 13.2% and 29.2%, averaging on 22.3%. It also outperforms the remaining versions in all ten benchmark programs tested. While the amounts of savings observed in simulation-based experiments and KNL-based experiments are different (as the simulator cannot model all details), the observed trends across different versions are similar in both cases.

5.5 Sensitivity Experiments

In this part of our experiments, we conduct a sensitivity study, using the simulator, by changing the default values of some of our major simulation parameters. In each experiment, we change the value of one parameter from Table 1.

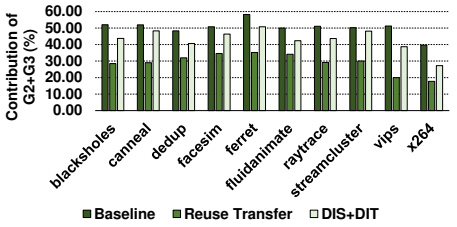

Figure 10. Contribution to the data reuses in G2+G3 to the total number of data reuses.

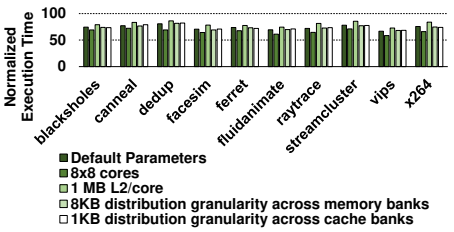

Figure 11. Results from the sensitivity experiments.

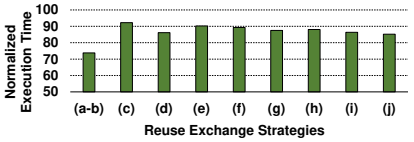

Figure 12. Results with alternate reuse transfer strategies.

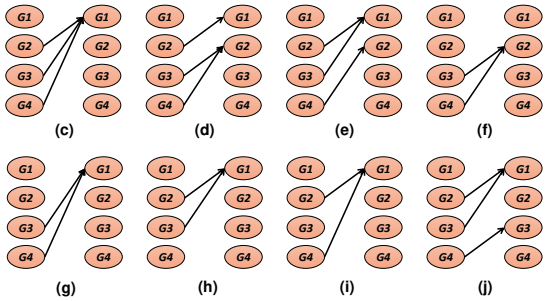

Figure 13. Illustration of different reuse transfer strategies. Note that, (a) and (b) are given in Figure 4.

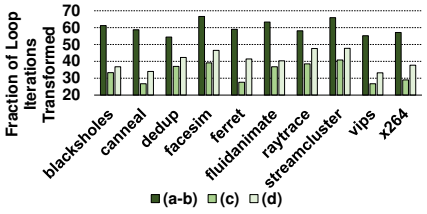

Figure 14. Fraction of loop iterations that have been transformed under different reuse transfer strategies. This in a sense also gives the fraction of time an approach cannot find a loop transformation to transform a given loop iteration.

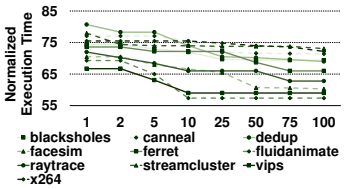

Figure 15. Behavior of our reuse transfer based strategy under different number of start points.

The second bar for each benchmark program in Figure 11 gives the normalized execution times with a larger (8x8) manycore. As can be seen, our approach generates better results with a large manycore, mainly because a larger manycore tends to increase the distances for data accesses which increases the benefits of our approach (as it reduces distance in space). Since a larger last level cache (third bar) captures more reuses, it leaves a smaller scope for our optimization. Nevertheless, we observe an average 19.9% parallel execution time reduction (when using our approach) over Baseline. The last two columns for each benchmark in Figure 11 give the results when the striping granularity over memory banks and cache banks is increased. Our savings are not very sensitive to the changes in the values of these two parameters.

5.6 Evaluating Alternate Options

In this section, our goal is to evaluate the other reuse transfer options and compare them against our defended reuse transfer strategy, i.e., $\{(G2 \rightarrow G1), (G3 \rightarrow G4)\}$ and $\{(G2 \rightarrow G4), (G3 \rightarrow G1)\}$. The reuse transfer strategies evaluated in this section are illustrated in Figure 13. It is to be noted that, the options shown in this figure are not the only possible ones; but, we believe they cover a large search space. Figure 12 plots the normalized execution times (with respect to Baseline) under the different reuse transfer strategies depicted in Figure 13. For each strategy, the bar shown represents the *average value* across all ten benchmark programs.

One can make the following observations from this graph. First, our proposed reuse transfer approach outperforms all other reuse transfer approaches. It is particularly interesting to note that, our defended reuse transfer strategy brings 18.4% additional performance improvements over $\{(G2 \rightarrow G1), (G3 \rightarrow G1), (G4 \rightarrow G1)\}$, which aggressively tries to transfer reuses from categories G2, G3, and G4 to category G1. To explain this result, we present in Figure 14 the fraction of loop iterations, averaged across all loop nests in the application, that have been transformed for reuse transfer strategies (a-b), (c), and (d). It can be seen from these results, compared to our defended reuse transfer strategy (a-b), strategy (c) transforms *much fewer* loop iterations. This is primarily because strategy (c) tries to transfer *all* types of reuses to the G1 category, and clearly, in attempting this, it faces with a lot of conflicts. As a result, a large fraction of the iterations cannot be transformed under (c). Interestingly, even a less aggressive approach, strategy (d), which tries to transfer some of the reuses to category G1 and some to category G2 performs better than strategy (c), by transferring more reuses in total (Figure 14) and improving execution time further (Figure 12).

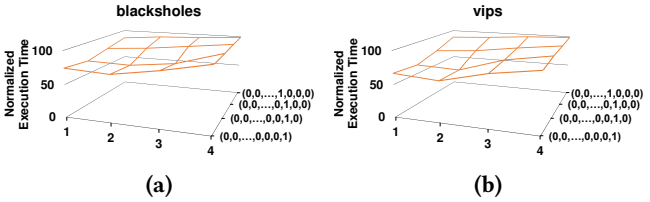

Figure 16. Results with different Δ_r (z-axis) and Δ_s (x-axis).

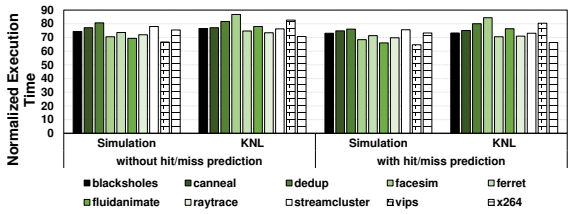

Figure 17. Results comparison of each benchmark w/ prediction against w/o prediction.

5.7 Comparison with the Limit Strategy

We see from Figure 8a that, the difference between our defended reuse transfer strategy and the limit strategy (Limit) is about 7.7%, on average. To explain the reason behind this gap between the two schemes, let us first remember how our reuse transfer strategy works. As explained in Section 3 in detail, our strategy first finds a loop transformation matrix (T), and then transfers using it as many data reuses as possible from categories G2/G3 to categories G1/G4. After that (i.e., when it comes to a point where no additional reuse can be transformed using this loop transformation matrix), it identifies another loop transformation matrix (a different T), and uses it to transfer as many remaining reuses in categories G2/G3 as possible, to categories G1/G4. Then, it determines another loop transformation matrix, and so on. It is possible that the algorithm can reach a point no further reuse can be transformed. Clearly, our approach is a *greedy strategy*, which tries to make the locally optimal choice at each stage with the intent/hope of finding a global optimum (i.e., the best set of loop transformation matrices that collectively transfer as many data reuses as possible from G2/G3 to G1/G4). However, like any greedy strategy, this approach's performance is affected by the "starting point". Clearly, a different selection for $\{t_1, t_2, t_3, t_4\}$ can lead to a different set of loop transformation (T) matrices and ultimately a different overall application performance. Since the possible values that $\{t_1, t_2, t_3, t_4\}$ can take are bounded by loop bounds as well as intrinsic data reuse characteristics of the loop nest being optimized, the entire search space is huge. In collecting the results with Limit, we made 100 experiments, each starting with a different data reuse between t_1 and t_2 and a different data reuse between t_3 and t_4, and selected the option (among 100) that led to most data reuse transfers.[9] Now, we present in Figure 15 the results when the number of experiments (start points) is varied as 1 (our default value), 5, 10, 25, 50, 75, and 100 (termed as Limit in this work). It can be observed from this plot that, the difference between 10 and 100 reuse trials seems to be quite small; so, 10 experiments with our scheme (each with a different start point) can be expected to produce a near-optimal performance.

[9]We want to mention that, when we increase the number of experiments to 200, we did not observe any better solution than the 100 experiments case.

5.8 Results with Other Δ_r and Δ_s Targets

So far in our experimental evaluation, we targeted DIT and DIS values of $\Delta_r = (0\ 0\ 0\ \ldots\ 0\ 1)^T$ and $\Delta_s \in \{(0\ 1)^T, (0\ -1)^T, (1\ 0)^T, (-1\ 0)^T\}$. In this part of our experimental evaluation, we present results with different target values for Δ_r and Δ_s. More specifically, we performed experiments by progressively increasing the values of the target Δ_r and Δ_s vectors. Figures 16(a) and (b) give the results (normalized execution times) for two of our benchmark programs, *blackscholes* and *vips*. Note that, in these plots, the Δ_r values are increased progressively from our default value (the best possible reuse vector) and similarly the Δ_s values are increased from 1 hop (our default) to 4 hops. it can be observed from these plots that, in both the benchmarks, slightly relaxing the target Δ_r and Δ_s values, i.e., setting Δ_r to $(0\ 0\ 0\ \ldots\ 1\ 0)^T$ and Δ_s to capture 2 hops generate better results than our default target thresholds. However, any further relaxation, does not yield better results. We want to emphasize that, the ideal Δ_r and Δ_s vectors to target is a function of intrinsic reuse characteristics of the loop nest being optimized as well as the parameters of the target manycore architecture. While our default targets perform reasonable well, taking additional program and architectural characteristics can result in better targets, which we are currently exploring.

Table 3. Multiprogrammed Workloads.

Workloads	Benchmarks
W1	blackscholes, canneal, dedup, facesim
W2	ferret, fluidanimate, raytrace, streamcluster
W3	vips, x264, blackscholes, canneal
W4	raytrace, streamcluster, canneal, x264
W5	fluidanimate, dedup, ferret, canneal
W6	blackscholes (2 instances), facesim (2 instances)
W7	x264 (2 copies), ferret (2 instances)
W8	raytrace (4 instances)

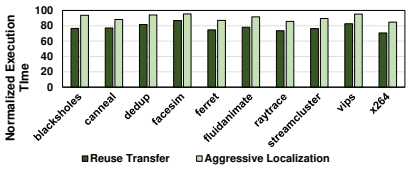

Figure 18. Comparison with aggressive localization.

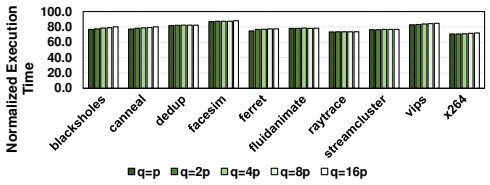

Figure 19. Comparison with different scheduling strategies.

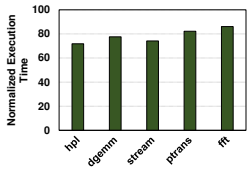

Figure 20. Results from HPC Challenge benchmark suite.

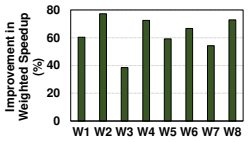

Figure 21. The percentage of weighted speedup.

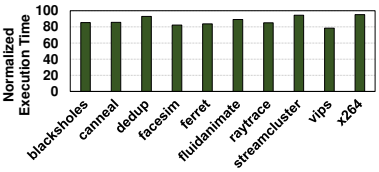

Figure 22. Results with Intel Cascade Lake.

5.9 Comparison against Aggressive Localization

We performed experiments with a modified version of our approach that aggressively localizes all data accesses, by targeting in a Δ_s threshold vector that contains all 0s. In a sense, omitting any data layout optimization, such an approach approximates the behavior of conventional strategies such as [45]. Figure 18 shows the normalized execution times of this strategy along with our reuse transfer results in KNL. The results indicate that such aggressive localization performs worse than our approach. The reason is that not considering degrees of locality prevents aggressive localization from finding a transformation matrix in many cases.

5.10 Results with Cache Hit/Miss Predictor

The exact data block location in the system may be different from the one assumed by the compiler. This can be the case in particular if the DIT is large. However, our approach can be modified to capture such scenarios as follows. We can employ a cache miss predictor to predict the location (cache or DRAM) of the data in question and use that location in our distance calculation. We implemented a hit/miss predictor based on cache miss equations [28] and reported the collected results from such an enhanced strategy in Figure 17. The results indicate that this version brings about 2.4% additional improvement over our default implementation in the case of simulation-based experiments and 2.8% in actual KNL runs.

5.11 Comparison with Other Scheduling Strategies

As stated earlier, our baseline scheduling strategy divides the iterations into p groups (where each group contains a set of consecutive iterations and p is the total core count) and assigns one group to each core. In this part of our evaluation, we measure the impact of different scheduling strategies. More specifically, instead of p, we divide the iterations into $q \ll p$ groups, and distribute the groups (blocks of iterations) over cores in a cyclic fashion. The results plotted in Figure 19 show the execution times of our reuse transfer based strategy normalized with respect to these new baselines ($q = p$ corresponds to the baseline used so far). It can be observed that our approach uniformly performs well when compared to all these block-cyclic distributions of iterations.

5.12 Results with the HPC Challenge Benchmarks

So far, all the results presented are collected using the Parsec [10] suite. In this part, we present the results collected using five benchmarks from the HPC Challenge benchmark suite [46], which contains applications with a range memory access patterns. The results, presented in Figure 20, show that our approach (DIS-DIT) brings an average performance improvement of 21.6% over the baseline execution.

5.13 Results with Multiprogrammed Workloads

We also conducted experiments with multiprogrammed workloads (of multithreaded applications), to capture cases when multiple applications co-run together and the threads of the applications can migrate during the course of execution. For this purpose, we formed 8 workloads, each having 4 applications (see Table 3). Each application is optimized using our approach and is parallelized over 36 threads. Also, each core is assigned a thread from each of the four applications. Figure 21 plots the percentage weighted speedup [18, 25, 58] improvements (compared to the original executions) with our optimization. We observe improvements ranging between 38.5% and 77.2%, indicating that our optimization performs quite well in the case of multiprogrammed workloads.

5.14 Results with a Three-Level Cache

We also performed experiments with a system that has a three-layer cache hierarchy – Intel Cascade Lake clocked at 2.5GHz. We use a two-chip configuration, each chip having 28 cores and 6 DDR4 channels and they are connected to one another using Intel UPI. L1 and L2 caches are private, whereas L3 cache is shared chip-wide (1.375 MB per core). The normalized execution times with our approach are plotted in Figure 22. As can be expected, while the execution time improvements are not as high as those achieved with KNL (due to the much larger cache capacity of Cascade Lake), we believe these savings are still good (12.9% when averaged over all 10 benchmark programs we have).

6 Related Work

Compiler-based approaches. Techniques for enhancing performance through improved data locality have been proposed for both single-core and multi-core systems [4, 6, 14, 22, 23, 26, 27, 31, 37, 38, 40, 40–42, 44, 45, 50, 51, 57, 61, 64–69, 71]. Loop and data layout transformations, have been used to enhance the temporal and spatial locality for cache performance and to expose inherent code parallelism [5, 72]. Wolf and Lam proposed unimodular transformations and tiling by estimating data reuse to improve cache locality [70]. Rivera and Tseng employed compile-time data-layout transformations to eliminate conflict misses [56]. Other proposed locality optimization works include [3, 20, 49] , targeting single-core systems. Tang et al. [67] proposed statement partitioning to reduce the data movement.

So et al. [59] presented an approach to deriving a custom data layout in multiple memory banks for multicore architectures. Sung et al. developed data layout transformation as an effective compiler performance optimization for memory-bound structured grid applications [63]. A data layout transformation targeting multithreaded applications running on multicores was proposed by [74].

Frameworks based on a polyhedral model to automate the data locality optimization transformations have been proposed by prior works [12, 13, 52–54] . Most of these works exploit transformations in polyhedral space to achieve data locality and parallelization. Compared to these polyhedral based optimizations, our approach is different from two aspects. First, we consider the core location in an architecture as a new dimension in locality optimization. Second, we optimize data locality considering data reuse in both time and space, whereas most prior efforts only focus on data reuse in time. In other words, our work is fundamentally different from these prior studies in that it tries to combine DIT and DIS optimizations under one unifying framework. As a result, it performs much better than DIT Only and DIS Only as discussed in our evaluation section.

Hardware-based approaches. Chishti et al. [19] proposed NuRAPID to decouple data placement from tag placement. Their approach enables flexible data placement where majority of frequently-accessed data are placed in the fastest subarrays. Hardavellas et al. [32] investigated Reactive NUCA which classifies the cache accesses into multiple classes and places cache blocks at the appropriate locations in the cache. Stenstrom et al. [62] proposed an adaptive protocol to accommodate migratory sharing of cache blocks. Near data computing is a recent research frontier, with specific examples such as the newly emerging Hybrid Memory Cube (HMC) [34] and High-Bandwidth Memory (HBM) [24]. Such memory architectures enable certain computations to be performed on the memory side, known as Processing in Memory (PIM) [1, 17, 39, 48, 73]. Beckmann et al. [7, 8] proposed hardware modifications to co-locate the data and computation to reduce the NUCA overheads. Ahn et al. [2] proposed a locality-aware PIM architecture to implement simple in-memory computation using compute-capable memory commands and specialized instructions. Nai et al. [47] proposed a near data graph computing architecture called GraphPIM that utilizes PIM functionality to achieves higher performance. Unlike these works where architecture modifications are required to achieve optimized data locality, our work is software-based and does not require any hardware modification. Moreover, our approach is complementary to the PIM-based execution paradigm and can potentially be combined with it.

7 Conclusions and Future Work

To our knowledge, this is the first work that considers both reuse in time and reuse in space for data locality optimization. It first formally defines (reuse) distance in time and (reuse) distance in space, and then presents a compiler-based optimization method that uses, as its primary optimization knob, a novel concept called "reuse transfer". The results collected using 10 multithreaded benchmark programs clearly show that the proposed co-optimization of reuse in time and reuse in space results in much better results than optimizing each in isolation (even if they are used in tandem).

As our future work, we plan to explore four research directions. First, we would like to investigate the interactions between our proposed approach and conventional loop parallelization techniques as well as PIM-based optimization strategies. Second, we would like to explore architectural support that can be employed to make our proposed optimization more effective. Third, work is underway in the automatic determination of Δ_r and Δ_s targets via machine learning (ML) based strategies. Finally, DIT and DIT capture different aspects of the locality. Since the work described in this paper is oriented toward co-optimization, we tried to bring them (DIS and DIT) together in the same mathematical formulation. But, by no means, this is the final word on the matter. In fact, we believe our main contribution – the idea of reuse transfer – is more general and can be employed in other optimization frameworks that may want to integrate DIT and DIS in a different fashion, e.g., by trying to interpret unit distances in DIT and DIS in terms of some (compiler-estimated) latency.

Acknowledgement

The authors sincerely thank Dr. Milind Kulkarni for shepherding the paper. The authors would also like to thank the anonymous PLDI 2021 reviewers for their constructive feedback and suggestions. 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] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *ISCA*. https://doi.org/10.1145/2749469.2750386

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*. https://doi.org/10.1145/2749469.2750385

[3] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. 1995. Data and computation transformations for multiprocessors. In *PPoPP*. https://doi.org/10.1145/209937.209954

[4] Jennifer M. Anderson and Monica S. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *PLDI*. https://doi.org/10.1145/173262.155101

[5] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. 1993. Automatic program parallelization. *Proc. IEEE* 81, 2 (1993), 211–243. https://doi.org/10.1109/5.214548

[6] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. 1999. Maps: A Compiler-Managed Memory System for Raw Machines. In *ISCA*. https://doi.org/10.1145/300979.300980

[7] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *PACT*. IEEE, 213–224. https://doi.org/10.1109/PACT.2013.6618818

[8] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *HPCA*. IEEE, 538–550. https://doi.org/10.1109/HPCA.2015.7056061

[9] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*. Springer, 283–303. https://doi.org/10.1007/978-3-642-11970-5_16

[10] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University. ftp://ftp.cs.princeton.edu/techreports/2010/890.pdf

[11] 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). https://doi.org/10.1145/2024716.2024718

[12] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI*. https://doi.org/10.1145/1375581.1375595

[13] Uday Bondhugula, J Ramanujam, and P Sadayappan. 2013. PLuTo: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler

[14] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *ASPLOS*. https://doi.org/10.1145/195470.195557

[15] Mainak Chaudhuri. 2009. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*. https://doi.org/10.1109/HPCA.2009.4798258

[16] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. 2007. Scheduling threads for constructive cache sharing on CMPs. In *SPAA*. https://doi.org/10.1145/1248377.1248396

[17] 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 *ISCA*. https://doi.org/10.1145/3007787.3001140

[18] Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut Kademir, Anand Sivasubramaniam, Onur Mutlu, and Chita R. Das. 2012. Application-Aware Prefetch Prioritization in on-Chip Networks. In *PACT*. https://doi.org/10.1145/2370816.2370886

[19] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. 2003. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *MICRO*. https://doi.org/10.1109/MICRO.2003.1253183

[20] M. Cierniak and W. Li. 1995. Unifying data and control transformations for distributed shared memory machines. In *PLDI*. https://doi.org/10.1145/223428.207145

[21] G.B. Dantzig and B.C. Eaves. 1973. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial T heory* 14, A (1973), 288–297. https://apps.dtic.mil/sti/citations/AD0750674

[22] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil Computation Optimization and Autotuning on State-of-the-art Multicore Architectures. In *ICS*. https://doi.org/10.1109/SC.2008.5222004

[23] 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)*. https://doi.org/10.1145/2737924.2737989

[24] Dong Uk Lee et al. 2014. 25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In *ISSCC*. https://doi.org/10.1109/ISSCC.2014.6757501

[25] Stijn Eyerman and Lieven Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* 28, 3 (May 2008), 42–53. https://doi.org/10.1109/MM.2008.44

[26] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *ICS*. https://doi.org/10.1145/1188455.1188543

[27] Matteo Frigo and Volker Strumpen. [n.d.]. Cache Oblivious Stencil Computations. In *ICS*. https://doi.org/10.1145/1088149.1088197

[28] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1998. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *ASPLOS*. https://doi.org/10.1145/291069.291051

[29] Seymour Ginsburg and Edwin H. Spanier. 1966. Semigroups, Presburger formulas, and languages. *Pacific J. Math.* 16, 2 (1966), 285–296. https://projecteuclid.org:443/euclid.pjm/1102994974

[30] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *IMPACT*, Vol. 2011. 1. http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-07.pdf

[31] 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*. https://doi.org/10.1109/SUPERC.1995.241596

[32] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *ISCA*. https://doi.org/10.1145/1555754.1555779

[33] https://software.intel.com/en-us/c compilers. [n.d.]. ([n. d.]).

[34] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In *Symposium on VLSI Technology (VLSIT)*. https://doi.org/10.1109/VLSIT.2012.6242474

[35] Mahmut Kandemir, Alok Choudhary, J Ramanujam, and Prith Banerjee. 1999. A matrix-based approach to global locality optimization. *J. Parallel and Distrib. Comput.* (1999). https://doi.org/10.1006/jpdc.1999.1552

[36] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. 2001. A layout-conscious iteration space transformation technique. *IEEE Trans. Comput.* (2001). https://doi.org/10.1109/TC.2001.970571

[37] Mahmut Kandemir, Hui Zhao, Xulong Tang, and Mustafa Karakoy. 2015. Memory Row Reuse Distance and Its Role in Optimizing Application Performance. In *SIGMETRICS*. https://doi.org/10.1145/2745844.2745867

[38] Mahmut Taylan Kandemir, Jihyun Ryoo, Xulong Tang, and Mustafa Karakoy. 2021. Compiler support for near data computing. In *PPoPP*. 90–104. https://doi.org/10.1145/3437801.3441600

[39] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *ISCA*. https://doi.org/10.1145/3007787.3001178

[40] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2018. Enhancing Computation-to-core Assignment with Physical Location Information. In *PLDI*. https://doi.org/10.1145/3296979.3192386

[41] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2017. POSTER: Location-Aware Computation Mapping for Manycore Processors.. In *PACT*. https://doi.org/10.1109/PACT.2017.20

[42] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. 2008. Optimistic Parallelism Benefits from Data Partitioning. *SIGPLAN Not.* (2008). https://doi.org/10.1145/1353535.1346311

[43] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. https://doi.org/10.1109/CGO.2004.1281665

[44] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. In *ICS*. https://doi.org/10.1145/305138.305197

[45] 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 *PACT*. https://doi.org/10.1109/PACT.2009.36

[46] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. 2005. *Introduction to the HPC challenge benchmark suite*. Technical Report. Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US). https://www.osti.gov/biblio/860347

[47] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyeoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *HPCA*. https://doi.org/10.1109/HPCA.2017.54

[48] Lifeng Nai and Hyesoon Kim. 2015. Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals. In *International Symposium on Memory Systems*. https://doi.org/10.1145/2818950.2818982

[49] M. F. P. O'Boyle and P. M. W. Knijnenburg. 1999. Efficient parallelization using combined loop and data transformations. In *PACT*. https://doi.org/10.1109/PACT.1999.807573

[50] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *PACT*. https://doi.org/10.1145/2967938.2967940

[51] Venkata K. Pingali, Sally A. McKee, Wilson C. Hseih, and John B. Carter. 2002. Computation Regrouping: Restructuring Programs for Temporal Data Cache Locality. In *ICS*. https://doi.org/10.1145/514191.514227

[52] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative Optimization in the Polyhedral Model: Part Ii, Multidimensional Time. In *PLDI*. https://doi.org/10.1145/1379022.1375594

[53] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. 2007. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *CGO*. https://doi.org/10.1109/CGO.2007.21

[54] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. 2000. Generation of Efficient Nested Loops from Polyhedra. *Int. J. Parallel Program.* 28, 5 (Oct. 2000), 469–498. https://doi.org/10.1023/A:1007554627716

[55] S. Ramos and T. Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *IPDPS*. 297–306. https://doi.org/10.1109/IPDPS.2017.30

[56] Gabriel Rivera and Chau-Wen Tseng. 1998. Data Transformations for Eliminating Conflict Misses. In *PLDI*. https://doi.org/10.1145/277650.277661

[57] Jihyun Ryoo, Orhan Kislal, Xulong Tang, and Mahmut Taylan Kandemir. 2018. Quantifying and Optimizing Data Access Parallelism on Manycores. In *MASCOTS*. https://doi.org/10.1109/MASCOTS.2018.00022

[58] Allan Snavely and Dean M. Tullsen. 2000. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *ASPLOS*. https://doi.org/10.1145/378993.379244

[59] B. So, M.W. Hall, and H.E. Ziegler. 2004. Custom data layout for memory parallelism. In *CGO*. https://doi.org/10.1109/CGO.2004.1281682

[60] Avinash Sodani. 2015. Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In *IEEE Hot Chips 27 Symposium (HCS)*. https://doi.org/10.1109/HOTCHIPS.2015.7477467

[61] Yonghong Song and Zhiyuan Li. 1999. New Tiling Techniques to Improve Cache Temporal Locality. In *PLDI*. https://doi.org/10.1145/301631.301668

[62] Per Stenström, Mats Brorsson, and Lars Sandberg. 1993. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *ISCA*. https://doi.org/10.1145/173682.165147

[63] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. 2010. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *PACT*. https://doi.org/10.1145/1854273.1854336

[64] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. 2016. Improving Bank-Level Parallelism for Irregular Applications. In *MICRO*. https://doi.org/10.1109/MICRO.2016.7783760

[65] Xulong Tang, Mahmut Taylan Kandemir, Mustafa Karakoy, and Meenakshi Arunachalam. 2019. Co-optimizing memory-level parallelism and cache-level parallelism. In *PLDI*. 935–949. https://doi.org/10.1145/3314221.3314599

[66] Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2019. Computing with Near Data. In *SIGMETRICS*. https://doi.org/10.1145/3376930.3376948

[67] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *MICRO*. https://doi.org/10.1145/3123939.3123954

[68] S. Verdoolaege, M. Bruynooghe, G. Janssens, and P. Catthoor. 2003. Multi-dimensional incremental loop fusion for data locality. In *ASAP*. https://doi.org/10.1109/ASAP.2003.1212826

[69] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *ASPLOS*. https://doi.org/10.1145/248208.237205

[70] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *PLDI*. https://doi.org/10.1145/113445.113449

[71] 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). https://doi.org/10.1109/71.97902

[72] M. Wolfe. 1996. Addison-Wesley, Reading, MA. In *High-Performance Compilers for Parallel Computing*. https://doi.org/10.1177/105960118200700110

[73] Jialiang Zhang, Soroosh Khoram, and Jing Li. 2017. Boosting the Performance of FPGA-based Graph Processor Using Hybrid Memory Cube: A Case for Breadth First Search. In *FPGA*. https://doi.org/10.1145/3020078.3021737

[74] Yuanrui Zhang, Wei Ding, Jun Liu, and Mahmut Kandemir. 2011. Optimizing Data Layouts for Parallel Computation on Multicores. In *PACT*. https://doi.org/10.1109/PACT.2011.20