Computing with Near Data

XULONG TANG, Pennsylvania State University, USA
MAHMUT TAYLAN KANDEMIR, Pennsylvania State University, USA
HUI ZHAO, University of North Texas, USA
MYOUNGSOO JUNG, Yonsei University, SOUTH KOREA
MUSTAFA KARAKOY, TOBB University of Economics and Technology, TURKEY

One cost that plays a significant role in shaping the overall performance of both single-threaded and multi-thread applications in modern computing systems is the cost of moving data between compute elements and storage elements. Traditional approaches to address this cost are code and data layout reorganizations and various hardware enhancements. More recently, an alternative paradigm, called Near Data Computing (NDC) or Near Data Processing (NDP), has been shown to be effective in reducing the data movements costs, by moving computation to data, instead of the traditional approach of moving data to computation. Unfortunately, the existing Near Data Computing proposals require significant modifications to hardware and are yet to be widely adopted.

In this paper, we present a software-only (compiler-driven) approach to reducing data movement costs in both single-threaded and multi-threaded applications. Our approach, referred to as Computing with Near Data (CND), is built upon a concept called "recomputation", in which a costly data access is replaced by a few less costly data accesses plus some extra computation, if the cumulative cost of the latter is less than that of the costly data access. If implemented carefully, CND can successfully trade off data access with computation, and considering the continuously increasing latency gap between the two, doing so can significantly reduce the execution latencies of both sequential and parallel application programs.

We i) quantify the intrinsic recomputability of a set of single-threaded and multi-threaded applications, ii) propose a practical, compiler-driven approach that automatically transforms a given application code fragment to a version that employs recomputation, iii) discuss an optimization strategy that increases recomputability; and iv) compare CND, both qualitatively and quantitatively, against NDC. Our experimental analysis of CND reveals that i) the average recomputability across our benchmarks is 51.1%, ii) our compiler-driven strategy is able to exploit 79.3% of the recomputation opportunities presented by our workloads, and iii) our enhancements increase the value of the recomputability metric significantly. As a result, our compiler-driven approach with the proposed enhancements brings an average execution time improvement of 40.1%.

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

Additional Key Words and Phrases: Manycore systems, data movement, recomputation

Authors' addresses: Xulong Tang, Pennsylvania State University, State College, PA, 16801, USA, xzt102@psu.edu; Mahmut Taylan Kandemir, Pennsylvania State University, State College, PA, 16801, USA, mtk2@psu.edu; Hui Zhao, University of North Texas, Denton, TX, 76203, USA, hui.zhao@unt.edu; Myoungsoo Jung, Yonsei University, Seoul, SOUTH KOREA, m.jung@yonsei.ac.kr; Mustafa Karakoy, TOBB University of Economics and Technology, Ankara, TURKEY, m.karakoy@yahoo.co.uk.

2476-1249/2018/1-ART42 $15.00
https://doi.org/10.1145/3287321

ACM Reference Format:
Xulong Tang, Mahmut Taylan Kandemir, Hui Zhao, Myoungsoo Jung, and Mustafa Karakoy. 2018. Computing with Near Data. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 3, Article 42 (December 2018), 30 pages. https://doi.org/ 10.1145/3287321

1 INTRODUCTION

Conventional computing uses memory to store input data, intermediate data, and output data. Data are read from memory whenever needed, computations are performed on the read data, and the updated data are stored back in memory. As a result, in conventional computing, "memory access" plays a central role that determines performance as well as power consumption. Observing this, many prior research papers considered efficient management of data movements across various layers in cache-memory-storage hierarchy. The proposed techniques include data locality optimizations in software [17, 31, 36, 37, 39, 56, 61, 64] and hardware [13, 28, 33, 46, 59, 60, 70], careful design and management of cache and memory hierarchies [13, 54, 69, 71], as well as proposals oriented towards taking advantage of memory-level parallelism [38, 43, 53, 58].

Unfortunately, the relative cost of a data access is continuously increasing compared to the relative cost of a computation, and today, depending on the location of data, performing a data access can incur a much higher cost (latency measured in terms of CPU cycles) than performing a computation. Table 1 gives the costs of data accesses from different layers in a memory hierarchy for representative CPU and GPU based systems. While these numbers already point to a problem for application programs that make frequent data accesses, one can expect the situation to be even worse as big data becomes more prevalent. In particular, despite the large cache memory spaces made available with modern computer architectures, increases in sheer data sizes and workloads with irregular data accesses (e.g., graph processing applications) make it extremely difficult to guarantee that most of data accesses would be satisfied from the caches that are close to compute units and would, as a result, incur relatively low overheads.

Motivated by this, more recently, a new computation paradigm called "near data computing" (NDC) has emerged [8, 16, 19, 27, 40, 44]. The main idea behind NDC is to perform computation in or around where data currently is, that is, moving computation to data, instead of moving data to computation (as in the traditional computing paradigm). Popular incarnations of this approach include emerging technologies, such as Hybrid Memory Cube (HMC [30]) that enable processing-in-memory (PIM [24]) functionality as well as computation-in-network [25, 47]. It needs to be emphasized that such techniques can significantly reduce data access latencies and in turn boost application performance. However, most of the existing NDC proposals require significant architectural modifications and current NDC-based architectures also lack software support.

Table 1. Latency of data accesses on modern architectures.

	L1 cache hits	L2 cache hits	LLC cache hits	DRAM
Intel Xeon [2]	4 cycles	12 cycles	~38 cycles	~38 cycles + ~46 ns
Intel i7 [3]	4 cycles	14 cycles	68 cycles	79 cycles + 50 ns
Nvidia V100 [5]	~28 cycles	~193 cycles	N/A	~1029cycles
AMD Ryzen 7 [4]	4 cycles	17 cycles	~37 cycles	40 cycles + 90 ns

In this paper, we take a different look at the growing costs of data accesses, and propose "computing with near data", CND, as another way of reducing data access and movement latencies. As opposed NDC, our CND does *not* require any hardware modification, and instead employs a software-based concept called *recomputation*. In recomputation, a costly data access is replaced by a few less costly data accesses plus some extra computation, if the cumulative cost of the latter is less than that of the costly data access. An example would be recomputing the value of a data element that resides in off-chip memory using data elements that are available in on-chip caches,

instead of directly performing the costly off-chip access. Thus, if implemented carefully, CND can successfully trade off data access with computation, and considering the continuously increasing latency gap between the two, doing so can significantly reduce the execution latencies of both sequential and parallel application programs.

Targeting emerging manycore systems and using a diverse set of single-threaded and multi-threaded workloads, this paper makes four main contributions:

- It defines a new metric called *recomputability* which captures the possibility of recomputing the value of a variable. It then presents a detailed analysis of various application workloads from a recomputability perspective. This experimental analysis also reveals the "limits" of recomputability if no extra effort is made to improve it. The collected experimental results indicate that the average recomputability across our benchmarks is about 51.1%.
- It presents a practical approach to computing with near data (CND) that employs recomputation in a "data locality-aware" fashion. This approach relies on compiler for both code analysis and code transformation, e.g., replacing a data reference (access) with its "recomputed version". Our experimental data indicate that the proposed compiler-based approach is able to exploit 79.3% of the recomputation opportunities presented by our workloads. The paper also discusses the reasons behind the untapped potential.
- It presents a technique to improve the value of the recomputability metric. This technique is based on the duplication/copy of a variable's value that is critical for recomputation. Our experimental evaluation of this technique shows that it increases the value of the recomputability metric significantly. As a result, now, our compiler-based approach brings an average execution time improvement of 40.1%.
- It compares CND against three "ideal" incarnations of NDC, and shows that CND outperforms all three NDC approaches for all the benchmark programs tested.

The next section gives an overview of the manycore architecture used in this study. Section 3 provides a high-level overview of our approach through several motivational examples, and also point out various issues that would be faced by any recomputation-based optimization strategy. Section 4 compares CND and NDC qualitatively, and Section 5 presents our workloads and experimental methodology. Section 6 proposes several metrics and quantifies the inherent recomputability of our workloads. Section 7 presents a compiler-driven strategy oriented towards taking advantage of recomputation opportunities in our workloads. Section 8 discusses a strategy to increase the opportunities for recomputation. A quantitative comparison of CND against NDC is given in Section 9. Section 10 discusses the prior works related to our proposal, and the paper is concluded, in Section 11, with a summary of its major contributions and a brief outline of the planned future work.

2 TARGET MANYCORE

In this paper, we use Intel Xeon Phi Knights Landing (KNL) as our targeted manycore architecture [55], though our analysis and optimizations are applicable to other manycore systems as well. As depicted in Figure 1, KNL consists of 36 (6×6) tiles and these tiles are connected via a mesh-based on-chip network. Each tile has two cores and a 1 MB L2 cache shared between them. Note that the L2 cache is private to each tile and directory-based cache coherence is maintained across L2 caches in different tiles. Each core in a tile consists of two vector units to execute vector instructions, as well as a core-private L1 cache. The KNL is equipped with a 16 GB on-package multi-channel DRAM memory (MCDRAM), which is partitioned into 8 channels. Each channel is managed by a memory controller (EDC) and all 8 EDCs are attached to the four corners of the on-chip network. Data access requests (if missed in the L2 cache), are forwarded to corresponding EDC and the

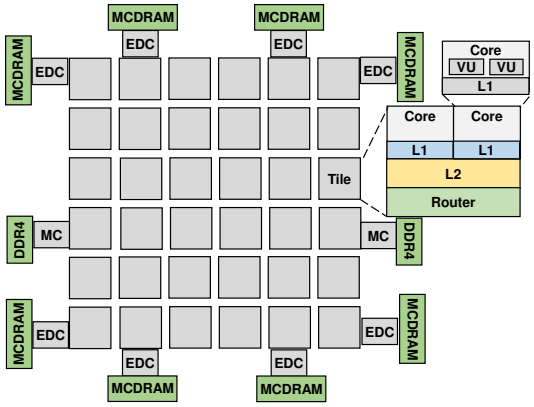

Fig. 1. Our target manycore architecture.

requested data are sent back to the requesting tile. Apart from MCDRAM, the KNL has two DDR memory controllers (MC) and each MC controls 3 channels of DDR4 off-chip memory. In total, the KNL consists of 6 channels of DDR4 off-chip memory working at 2,400 MHz, and the capacity of DDR4 supports can be up to 384 GB.

To reduce the on-chip network data movements, the KNL provides three different "cluster modes": i) "all-to-all" mode, ii) "quadrant" mode, and iii) "sub-NUMA" (SNC-4) mode. In "all-to-all" mode, addresses are uniformly spread across caches and memory channels, which means that the requested data from a tile can reside in any tile's L2 cache (subject to coherence) or in any memory channel. In "quadrant" mode, on the other hand, the on-chip network is partitioned into four quadrants; although the requested data from a tile can be in any tile's L2 cache, it can only access the two MCDRAM channels that are attached to that quadrant. Finally, in "SNC-4" mode, the mesh network is split into 4 NUMA sub-regions, which means both L2 accesses and memory accesses are limited within a sub-region.

In addition to cluster modes, the KNL also provides three different configurations for MCDRAM, know as "memory modes". In "cache mode", the MCDRAM acts entirely as a last-level cache (LLC). In "flat mode", the MCDRAM is configured as normal addressable memory. Finally, in "hybrid mode", 25% (or 50%) capacity of MCDRAM is configured as LLC and the remaining capacity is configured as addressable memory.

3 HIGH LEVEL VIEW OF CND AND POTENTIAL PROBLEMS

3.1 Motivational Examples

Figure 2a presents a code example that can be used to illustrate the idea behind recomputation and CND. We focus on two particular statements (S_x and S_y). Let us assume that execution is currently on statement S_y, and the current locations of data elements are as shown in Figure 3a. When S_y is executed, normally (i.e., in the traditional computing paradigm), a main memory access is performed for $a[i]$, as $a[i]$ is present in the off-chip main memory. In comparison, when employing CND, we *recompute* the value of $a[i]$, using $b[i]$ and $c[i]$ and an extra addition. Assuming, for illustrative purposes, that the access latencies for the current locations where $a[i]$, $b[i]$, and $c[i]$ are stored are 200 cycles, 2 cycles and 20 cycles, respectively, and the cost of an addition operation is 1 cycle, using recomputation instead of direct memory access for $a[i]$ can give us 177 (=200-(2+20+1)) cycles saving. Clearly, recomputation may not be beneficial for every case. Consider, for

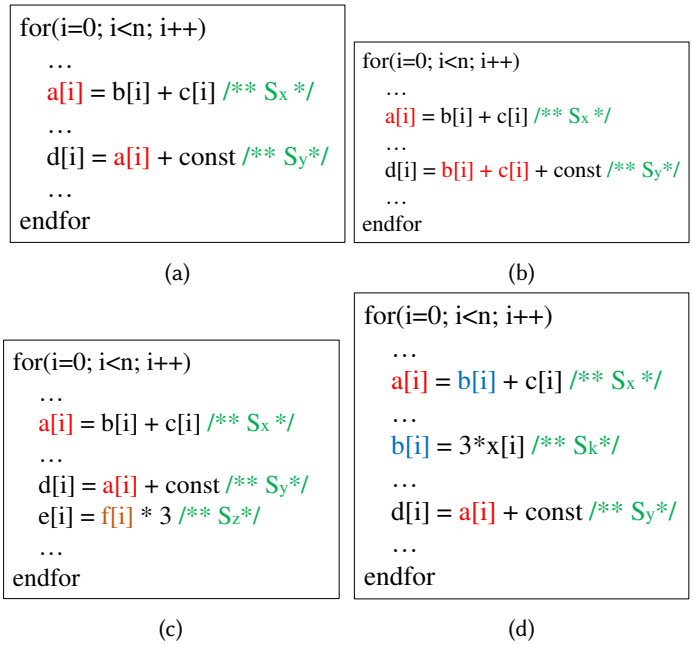

Fig. 2. Code examples.

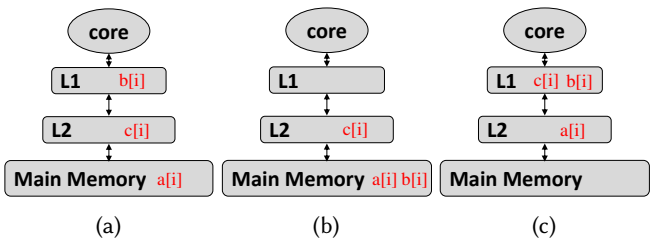

Fig. 3. Data locations of different data elements for the code fragments in Figure 2.

instance, the alternate locations for the same three data elements depicted in Figure 3b. In this case, recomputation would not work well as employing it would lead to a cost of 221 (=200+20+1) cycles.

Note that, for recomputation to be useful, the costly access (one to be replaced by recomputation) does not need to be in off-chip memory. Consider the same code snippet in Figure 2a. If $a[i]$ is present in L2 cache, and $b[i]$ and $c[i]$ are present in L1 cache (illustrated in Figure 3c), it can still be beneficial to recompute $a[i]$ in statement S_y using $b[i]$ and $c[i]$, as doing so replaces the costly L2 cache access with two less expensive L1 cache accesses and 1 addition operation.

3.2 Interaction with Data Locality

It is to be observed that CND is applicable even to codes that have already been heavily optimized for data locality, as even in such codes a large fraction of data would reside in costly storage locations. It is also important to note however that recomputation usually changes the locations of the data involved as well and, as a result, the locality of different data accesses, and this can lead to a conflict between employing recomputation and exploiting cache locality. Consider another code example (illustrated in Figure 2c) with an added statement S_z to the code sample in Figure 2a. Statement

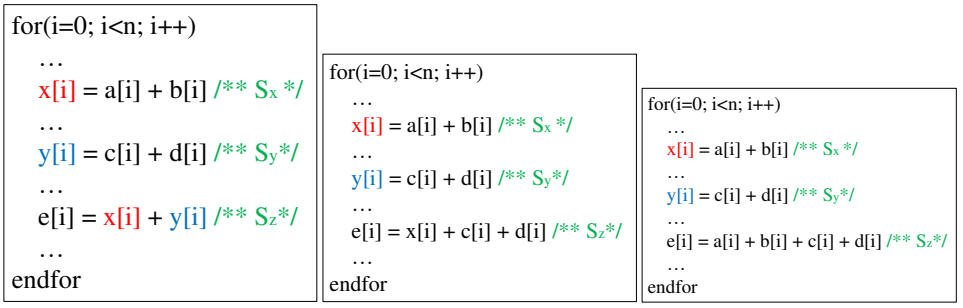

Fig. 4. Example illustrating different versions (paths) of recomputation.

S_z is executed immediately after statement S_y. Let us assume we choose to recompute $a[i]$ in S_y using $b[i]$ and $c[i]$ because the data locations are the same as in Figure 3a. However, doing so will fetch both $b[i]$ and $c[i]$ into the L1 cache while executing S_y. In other words, during recomputation, more data elements are fetched into the L1 cache compared to the original case (where only $a[i]$ is fetched into the L1 cache). Let us also assume that, before executing S_y, $f[i]$ is present in the L1 cache. As a result, recomputing $a[i]$, i.e., substituting $a[i]$ with $b[i] + c[i]$, can cause $f[i]$ to be evicted from the L1 cache due to a conflict miss. Therefore, although recomputation benefits $a[i]$ in the statement S_y, the potential reuse of $f[i]$ in the L1 cache can be lost when recomputation is used.

3.3 Overwritten Data

One other potential issue with recomputation is that the data that will be used for recomputing the value for a costly data access should *not* be overwritten by the time we want to perform recomputation. To illustrate an example scenario, we focus on Figure 2d. Let us assume that we want to recompute $a[i]$ in S_y using $b[i] + c[i]$. However, as can be observed, $b[i]$ is modified by statement S_k, which is after S_x. As a result, we cannot substitute $a[i]$ in S_y using $b[i] + c[i]$ as doing so would use a wrong value of $b[i]$ and consequently lead to a wrong execution result. Later in Section 8, we discuss an automatic code transformation strategy to handle such cases.

3.4 Different Code Versions Employing Recomputation

A given code segment can be rewritten in multiple ways, each corresponding to a *version* that exploits some subset of possible recomputation opportunities. Consider the code snippet shown in Figure 4a, where both $x[i]$ and $y[i]$ in statement S_z can be substituted using S_x and S_y for recomputation. Based on their data locations, one may choose only recomputing $y[i]$ (Figure 4b), recomputing both $x[i]$ and $y[i]$ (Figure 4c), or even not using recomputation at all (Figure 4a). As one can observe, the total number of possible recomputation choices for S_z is 4. We refer to each possible combination of recomputation (substitution) choices of all statements in a loop body as a *version*. Given a loop nest with n statements, assuming that each statement i ($1 \leq i \leq n$) has m_i possible substitution/recomputation choices, the total number of versions is $\prod_{i=1}^{n} m_i$.

The total number of possible recomputed versions can significantly increase if the code fragment under consideration contains "recursive" computations. For example, consider the loop body shown in Figure 5 where there exists a loop-carried reuse of $a[i]$ (i.e., $a[i]$ is reused across consecutive loop iterations). Figure 5b illustrates the unrolled version with a stride of 4 iterations. As one can see, each $a[i + k]$ can be recomputed using $a[i + k - 1]$. Further, $a[i + k - 1]$ can be recomputed

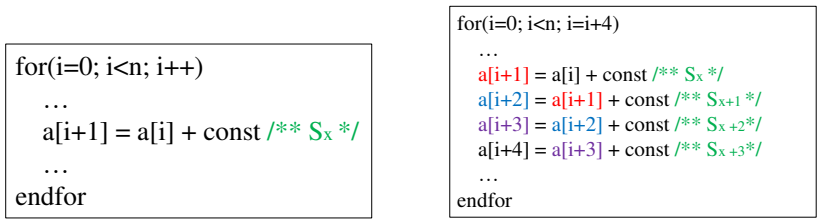

Fig. 5. Example illustrating recomputation with recursive substitution.

using $a[i + k - 2]$ and so forth, thereby exhibiting a recursive pattern. As a result, the number of different versions increases exponentially, leading to a very large recomputation "search space". However, the actual search space in practice is much smaller since many recomputation versions are simply *not* legal, i.e., they change the semantics of the original program. Furthermore, as we will discuss later in Section 7, most of the remaining versions can be quickly discarded using a branch-and-bound based strategy during the search process.

4 CND VERSUS NDC

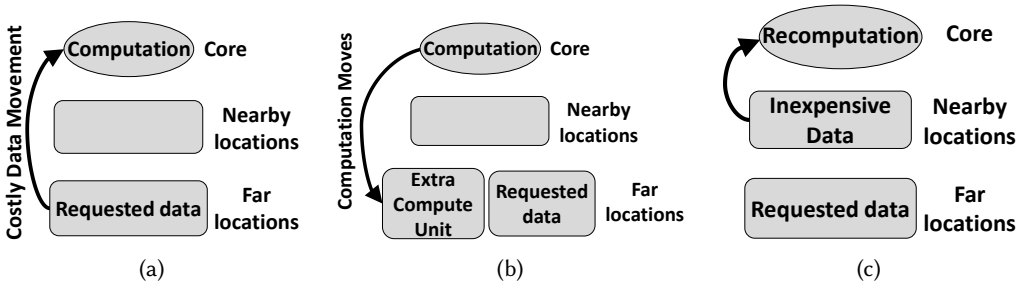

Fig. 6. (a) Traditional computing vs (b) near data computing (NDC) vs (c) computing with near data (CND).

In this section, we contrast CND with NDC and explain the similarities and differences between the two. We chose NDC for this comparison, as it is a relatively recent paradigm and, like CND, it also tries to reduce data access latencies and data movement costs. A conceptual comparison of the traditional mode of computation, NDC, and CND is given in Figure 6, assuming that a computation in the core shown originally wants to access data from far memory locations. In this figure, (a) illustrates the traditional computing paradigm, where a computation scheduled on a core that needs a data (say, with a long access latency) issues a request for it and the data is returned, from its far memory location, to the requesting core. This traditional option can be summarized as one in which we bring data to (the core that is to perform) computation. Clearly, in this option, we incur the full cost of the long-latency data access. NDC is illustrated in (b); it has several incarnations such as [18, 21, 24, 51], some of which will be further discussed in the related work section (Section 10). NDC works by bringing computation to data (usually through static scheduling), as opposed to bringing data to computation, which was the case in option in (a). Consequently, the distance between computation and data gets reduced, which in turn reduces data access latencies. In comparison, our proposed CND, illustrated in (c), eliminates the costly data access altogether and replaces it with recomputation, which involves a number of accesses to less costly data (in nearby memory locations) and some extra computation.

It is important to note that, in CND, as opposed to NDC, computation itself is *not* moved; instead, it is performed in its original location, albeit differently (i.e., via recomputation). As a result, CND does *not* require any extra computation units/circuitry typically needed in most NDC proposals (see Figure 6b). In fact, as we will show later in the paper, it is possible to reap a large fraction of the potential benefits of CND with a little help from the compiler. We also believe that CND can be expected to be more widely applicable than NDC. This is because, the additional hardware (e.g., computation units and associated circuitry), required in the case of NDC, cannot be placed into each and every chip location needed, as doing so would be extremely costly from both area and financial perspectives. As a result, a designer, based on her chip area and financial constraints, needs to favor some places (chip locations) over others, and this eventually leads to high data access latencies when accessing data that reside in not-favored locations. A quantitative comparison of CND against various NDC-based optimization strategies is later given in Section 9.

5 WORKLOADS AND EXPERIMENTAL METHODOLOGY

In this work, we evaluated CND using a set of single-threaded and multi-threaded workloads and a state-of-the-art manycore system, Intel Knight's Landing (KNL), described in detail in Section 2. We used the "cache mode" where the MCDRAM memory is used as an L3 (last-level cache).

Our single-threaded programs are from the floating-point suite of SPEC2006 [1] and our multi-threaded benchmarks are from SPLASH-2 suite [66]. The dataset sizes of our single-threaded benchmarks range between 4.1 GB and 17.3 GB, whereas those of our multi-threaded benchmarks range between 6.6 GB and 16.2 GB. Further, the L1, L2 and L3 (MCDRAM) miss rates of these benchmarks range, respectively, between 4.4%-39.1%, 3.1%-29.7%, and 1.9%-22.8%, indicating that, as far as cache/memory pressure is concerned, our benchmarks exhibit a great diversity. When performing experiments with a single-threaded application, we executed it on a randomly selected core, and no other application is executed on any other core. On the other hand, when experimenting with a multi-threaded application, we executed it on all cores available to us, in a one-thread-per-core fashion. Each experiment has been repeated 20 times, and the results reported below represent the average values.

Our compiler support, elaborated in Section 7, is implemented using LLVM [15], a state-of-the-art compiler tool-chain, as a source-to-source translator. We used the latest version of the LLVM (6.0.0), which includes the implementations of an extensive set of state-of-the-art code optimizations. The resulting optimized C/Fortran codes are then compiled using icc (Intel's compiler) with O3 flag, which activates well-known optimizations targeting both parallelism (e.g., SIMD optimizations) and data locality (e.g., loop permutation, iteration space tiling, and loop unrolling). In the presentation of our experimental evaluation below, we also use a concept called *location map*. Location map captures, at any given point in execution, the locations of all data elements. In the context of this work, the potential locations for a data element are *L1 cache*, *L2 cache*, *L3 cache* (MCDRAM configured as L3), *row-buffer*, and *main memory array (banks)*. It is to be noted that, for a given data access request, any of these locations is accessed/checked only after all the previous locations have been checked and the requested data could not be found in any of them (for example, L3 is checked, only if the data request misses in both L1 and L2).

We first present a qualitative and quantitative evaluation of CND and explain its limits (with the help of an optimal scheme) in Section 6. We then present a practical scheme (Section 7) that, while not bringing as much savings as the optimal scheme, improves performance significantly over the traditional mode of execution. Then, in Section 8, we present and experimentally evaluate an enhancement that improves the recomputability of a given program.

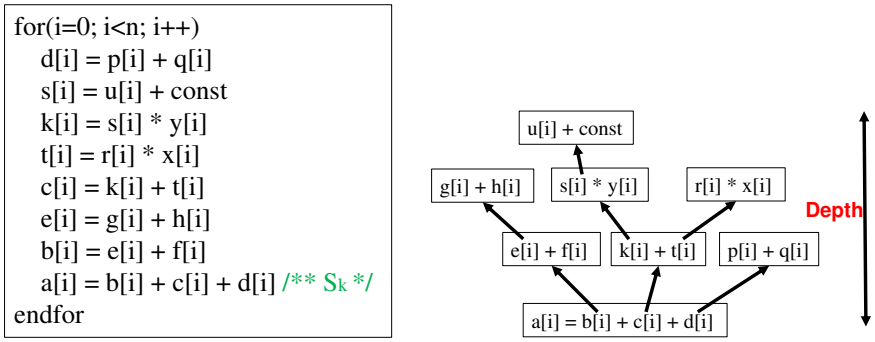

Fig. 7. An example illustrating recomputation depth.

6 RECOMPUTABILITY AND POTENTIAL OF CND

6.1 Relevant Metrics

We start our technical discussion of recomputability and CND by giving a series of metrics that are used in the remainder of this paper.

Definition 6.1. **Recomputability.** We say a data access is *recomputable* if the original values of all the variables needed to recompute it are still available (i.e., not overwritten); otherwise, it is called *unrecomputable*. Recall that Figure 2a and Figure 2d give examples of the recomputable and unrecomputable data accesses.

Definition 6.2. **Profitable Recomputability.** A *profitably recomputable* data access is a recomputable data access (per *Definition 6.1*) with the additional constraint that recomputing its value (using the values of all the variables needed to recompute it) is less expensive (in terms of total CPU cycles) than directly accessing it from its current storage location.

In mathematical terms, assuming that the value of x can be recomputed using the values of y_1, y_2, y_3, $\cdots$, y_n, an access to x is "profitably recomputable" if and only if:

$$cost(x) > \sum_{i=1}^{n} cost(y_i) + cost(\oplus),$$

where $cost(z)$ for a variable z refers to its access cost, and $cost(\oplus)$ represents the cumulative cost incurred by additional arithmetic/logic operations required for recomputation. For example, in statement S_y of Figure 2a, $a[i]$ would be profitably recomputable if $cost(a[i]) > cost(b[i]) + cost(c[i]) + cost(+)$, where $cost(+)$ is the cost of an addition operation.

Definition 6.3. **Benefit of Recomputation.** Benefit of recomputing x using the values of y_1, y_2, y_3, $\cdots$, y_n, can be expressed as

$$cost(x) - (\sum_{i=1}^{n} cost(y_i) + cost(\oplus)),$$

where the cost functions are as defined above.

Definition 6.4. **Recomputation Depth.** The depth of recomputation is the maximum number of successive substitutions made (for one of the variables need for recomputation) to recompute the value.

Consider, for example, the code fragment shown in Figure 7a. Figure 7b illustrates the concept of *recomputation depth*. In this case, the recomputation depth is 3, which corresponds to the length of the longest path. The depth of a recomputation is important, mainly because increasing the depth can sometimes make an otherwise unprofitable recomputation profitable. To be more concrete, using current example in Figure 7, let us focus on recomputation of $c[i]$ in Statement S_k. Suppose that $c[i]$, $k[i]$, $s[i]$ and $t[i]$ are present in the off-chip main memory before the execution of S_k. Let us further assume that $u[i]$, $r[i]$, $x[i]$ and $y[i]$ are present in the L1 cache. Obviously, recomputing $c[i]$ using $k[i] + t[i]$ is not beneficial as it introduces one extra memory access. However, if we further recompute $k[i]$ using $s[i] * y[i]$ and recompute $s[i]$ using $u[i] + const$, then we are able to obtain performance benefits as $u[i]$ is present in the L1 cache. Similarly, we recompute $t[i]$ using $r[i] * x[i]$. As a result, statement S_k can be rewritten into:

$$a[i] = b[i] + \underbrace{(u[i] + const) * y[i] + r[i] * x[i]}_{c[i]} + d[i],$$

where only L1 cache accesses and arithmetic operations are involved in recomputing $c[i]$. In this particular example, the beneficial depth of recomputation of $c[i]$ is 3.

Definition 6.5. **Location Map Summary (Distribution).** For a given recomputation, the location map summary gives the distributions of the locations of the variables involved.

Recall that data can be present in *L1 cache*, *L2 cache*, *L3 cache* (MCDRAM), *row-buffer*, and *main memory array (banks)*. As mentioned earlier in Section 5, a location map captures the location information of all the live variables before each statement. Further, a location map can help us identify which recomputations are profitable. Note that the location map can change after recomputation substitution for each and every statement, as the corresponding data accesses can change after substitution.

6.2 Algorithm for Generating All Recomputable Versions

We now give an algorithm that can be used to produce *all* recomputable versions of a given loop nest. Our proposal contains a backward scan and a forward scan. The backward scan is responsible for finding all the immediate substitutable choices for a particular variable, whereas the forward scan is used to eliminate those substitutable choices which are not recomputable. Note that, for each statement, we exhaustively generate all the recomputable versions without considering the location map and cost at this stage. Note also that the recursive recomputation is naturally captured in our algorithm, as the final versions of a particular statement include all the combinations of different versions of each variable involved. In other words, we only eliminate the non-recomputable versions (e.g., Figure 2d) from the search space. As this algorithm applies a backward scan as well as a forward scan on each variable, its asymptotic complexity is $O(S^2V)$, where S is the total number of statements in a loop nest, and V is the total number of variables in the loop nest.

6.3 Quantifying Recomputability

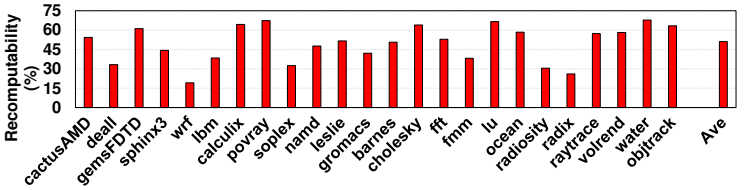

Fig. 8. Fraction of recomputable data accesses.

Algorithm 1 Generate recomputable versions.

INPUT: A loop nest
OUTPUT: All recomputable versions
for each statement S_i in the loop nest **do**
  $versions_{S_i} \leftarrow \emptyset$
  /**backward scan*/
  for each variable V_x in S_i **do**
    $set_{V_x} \leftarrow set_{V_x} \cup V_x$
    Backward scan the nearest assignment of V_x and find statement S_k
    for Each version P_k in $version_{S_k}$ **do**
      Forward scan from S_k to S_i
      if there exist an update to any variable in S_k **then**
        V_x is not recomputable using P_k
      else
        $set_{V_x} \leftarrow set_{V_x} \cup P_k$
      end if
    end for
  end for
  for all variables $V_x \cdots V_y$ in S_i **do**
    from variable sets set_{V_x} to set_{V_y}
    find all combinations and add each combination is to $versions_{S_i}$
  end for
end for
Generate all the recomputable versions
for all the statements $S_i \cdots S_j$ **do**
  from $versions_{S_i}$ to $versions_{S_j}$
  find all combinations where each combination is one output version of the loop nest.
end for

Having defined the important metrics related to recomputability and given an algorithm to generate *all* (recomputation-based) versions of a loop nest, we next quantify recomputability of our workloads using the target manycore. Figure 8 presents the fraction of recomputable data accesses (whether profitable or not) for our 24[1] benchmark programs. It can be observed that, on average, about 51.1% of all data accesses are recomputable. In each of the remaining data accesses, at least one of the required variables (for recomputation) is *not* available (i.e., its original value is already overwritten), making it unrecomputable. An example of this has already been given in Figure 2d, and we will later discuss in Section 8 how to make some of these unrecomputable data accesses recomputable.

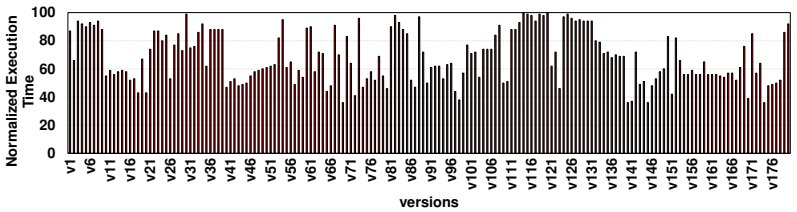

Fig. 9. The latencies (measured in CPU cycles) of different versions of the most time-consuming loop nest from *namd*, *normalized* with respect to the "slowest version" (whose execution time is set to 100).

Let us now focus on one of our benchmark programs, *namd* (a nanoscale molecular dynamics code), and explore it from a recomputability perspective in more detail. Figure 9 gives, for the most time-consuming loop nest of this benchmark, the latencies (measured in CPU cycles) of different versions (on the x-axis) of the loop body, *normalized* with respect to the "slowest version" (whose execution time is set to 100). In this plot, the execution time of the default (original) version is represented by the first bar (version $v1$), and the last bar corresponds to the version ($v180$) with all

[1]In graphs that present results for all benchmarks, the first 2 benchmarks on x-axis are single-threaded, whereas the remaining are multi-threaded.

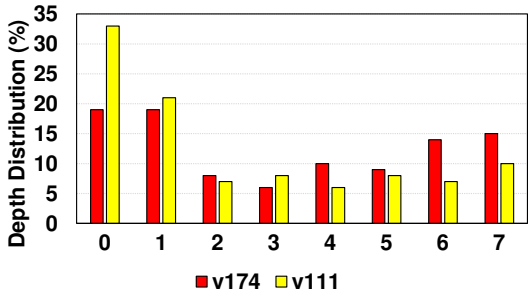

Fig. 10. The distribution of the depths of recomputation values of versions $v111$ and $v174$.

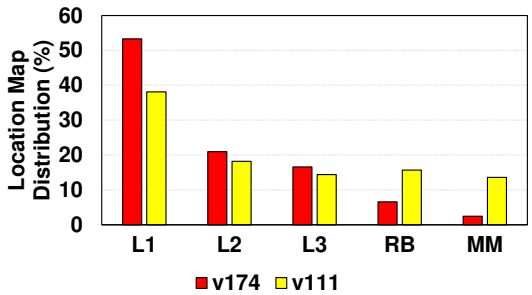

Fig. 11. The summary (distribution) of the location maps for versions $v111$ and $v174$.

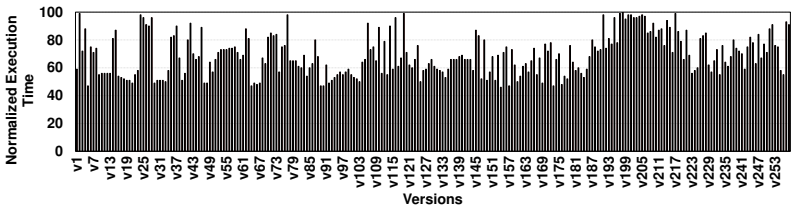

Fig. 12. The normalized execution times for the most time consuming loop nest from *raytrace*.

possible (legal) recomputations included. One can make the following observations from this graph. First, a large number of the recomputable versions generate better results than the default (original) version. Second, there is more than one version (3 in this case, to be more specific) that generate the best result. Third, the version with the most number of recomputations (corresponding to the one with the highest depth, whose result is captured by the last bar) does not lead to the best result. Together these observations mean that there is a great execution time variation (36%-100%) across different versions, some versions achieve significant execution time reduction over the original version (up to 58.6%), and there can be more than one version that generate the best improvement.

We now delve into more details focusing on two specific versions from Figure 9: one generating the best result ($v174$) and the other performing not so good ($v111$), and check their depth of recomputation and location maps. Figure 10 gives, for these two versions ($v111$ and $v174$), the distribution of the depths of recomputation values (note that each variable used in a given (original) computation can have a different depth, as already explained in Section 6.1, and the results reported in this figure are cumulative overall all variables in a version). We see that, although the maximum depth value is 7 in both the versions, the distribution of depth values are different from one another.

Figure 11 on the other hand shows the summary (distribution) of the location maps for these two versions. It can be observed from this figure that, most of the data accesses in $v174$ are satisfied from relatively fast storage locations (L1 and L2 caches to be specific), and very few data accesses

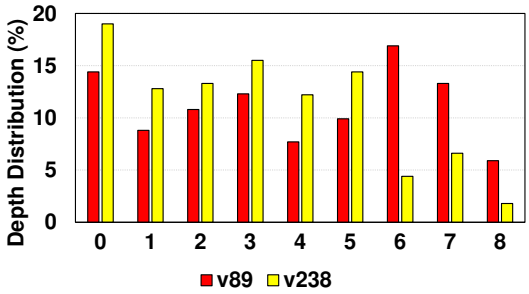

Fig. 13. The distribution of the depths of recomputation values of versions $v89$ and $v238$.

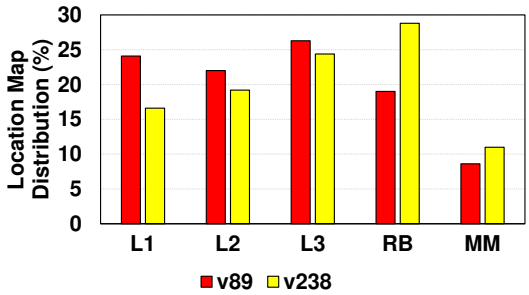

Fig. 14. The summary (distribution) of the location maps for versions $v89$ and $v238$.

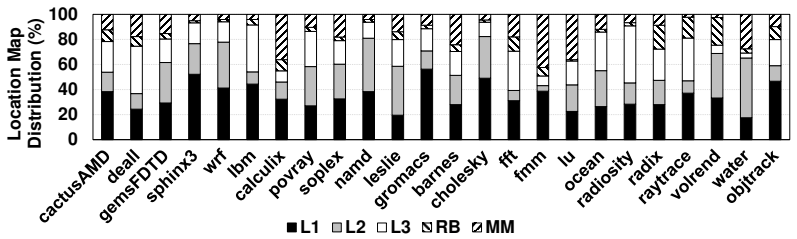

Fig. 15. The location map summary for the original versions of the applications.

are satisfied from row buffers (RB) and main memory (MM). In comparison, in the case of $v111$, data accesses are distributed more widely over the possible locations, where 15.7% and 13.6% are satisfied, respectively, from row buffers and memory banks. These results help us explain why version $v174$ performs much better than version $v111$. Figure 12 gives the execution times for the most time consuming loop nest from another application program (*raytrace*, a multi-threaded benchmark). Similar to Figure 9, the execution times of the different versions in Figure 12 also differ widely from one another, and there are 5 versions that generate the best result (as before, all results are normalized with respect to the "slowest version", and the first bar corresponds to the "original version"). Figures 13 and 14 give two different versions that employ recomputation ($v89$, one of the best performing versions, and $v238$) the depth distribution and location map distribution, respectively. And, again, these location map results help us understand the performance difference between the two versions.

We now present the location maps for the three different versions of the *entire applications*. Figure 15 gives the location map summary for the original versions of the applications. It can be observed that, on average, about 34.3%, 23.3%, 21.4%, 6.6% and 14.4% of data requests are satisfied from L1, L2, L3 (MCDRAM), row buffers and main memory, respectively. As can be expected, depending on the data locality they exhibit, different applications have quite different location

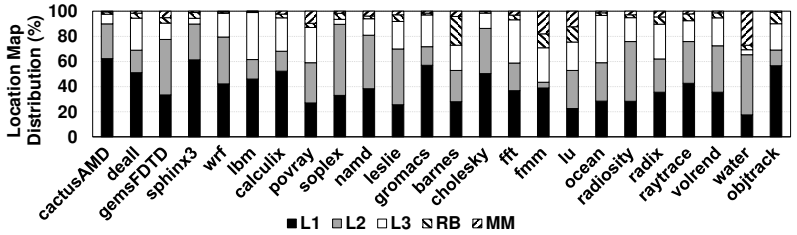

Fig. 16. The location map for the "best version" for each application.

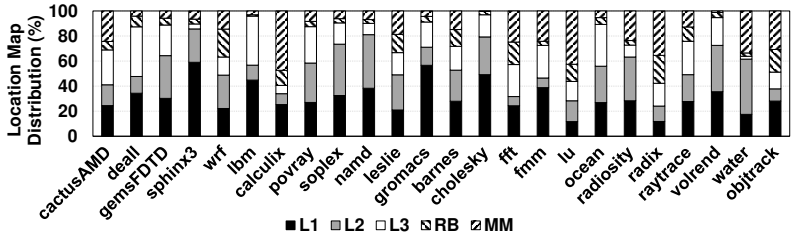

Fig. 17. The location maps for the most aggressively recomputed versions of all applications.

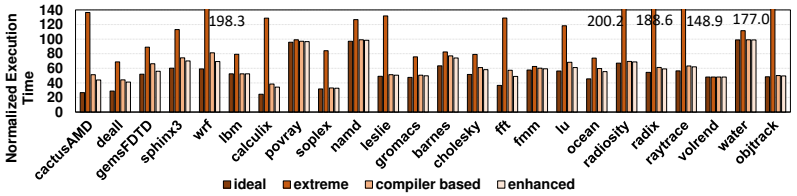

Fig. 18. The normalized execution time of the best version of each application program (lower, better).

maps. Figure 16 gives the location map for the "best version" for each application. For a given application, the best version is the one that employs the "best performing version" for each loop nest.[2] Comparing the results in Figures 15 and 16 clearly indicates that, the applications in the latter exhibit much better locality than the corresponding applications in Figure 15. In fact, in this case, 90.5% of total data requests are satisfied from either L1 or L2 or L3. For comparison purposes, we also present, in Figure 17, the location maps for the "most aggressively recomputed" versions of our applications, that is, the versions where each (legal) recomputation opportunity has been taken. It can be observed that, such aggressive recomputation does *not* generate good locality for a large majority of our application programs.

Finally, the first bar in Figure 18 plots the execution time of the best version (as defined above) of each application program tested. For each application, the execution time is *normalized* with respect to the execution time of the "original version" of that application (the latter is set to 100). Note that these results represent the *benefit* of *recomputation*. We see that, as can be expected, using the best recomputation option for each application generates the best overall execution time. In the same graph, the second bar for each application gives the result of the most aggressively recomputed version (whose location map is given in Figure 17). As can be observed, using aggressive recomputation can lead to disastrous results for many benchmarks.

While the results presented in Figure 18 for the best performing versions are very encouraging and clearly demonstrate the potential of recomputation (as an alternative to directly performing

[2]While the inter-nest data reuse can also play a role here, its contribution to the overall reuse is much lower compared to intra-nest data reuse.

costly/remote data accesses), recall that these results have been obtained via "exhaustive search" (i.e., for each loop nest in each application, the best performing version is identified and used). Clearly, such an approach would *not* be viable in a practical setting. Encouraged by the results in Figure 18 however, in the next section, we propose and experimentally evaluate a practical (compiler-based) approach that can take advantage of inherent recomputability exhibited by our single-threaded and multi-threaded application programs.

7 A PRACTICAL APPROACH TO EXPLOITING RECOMPUTABILITY

This section presents a *compiler-driven* strategy to take advantage of recomputability. More specifically, we present a compiler algorithm that transforms a given input code to an optimized version that carefully employs recomputation. In doing so, our algorithm tries to *balance* the opportunities enabled by recomputation and the opportunities revealed data reuse, with the goal of approaching the maximum improvements presented in the previous section.

7.1 Compiler Algorithm

At a high level, our algorithm scans the entire search space of code versions (for a given code fragment to optimize) and prunes that space using CME (explained below), which estimates the number of cache misses for each version. Among all the versions scanned, our approach selects the one with the "minimum cost", i.e., the minimum number of cache misses. As a result, any potential negative performance impact of our approach is taken into account.

For a given loop nest, we first partition the loop nest into n sub-loops. A *sub-loop* is a portion of loop iterations of a given loop nest. The reason we partition a loop nest into sub-loops is because different loop iterations might have different profitable recomputation versions. For example in Figure 19a, the entire loop nest with N iterations is partitioned in to n sub-loops. Each sub-loop has N/n iterations. The code snippet in Figure 19a gives the kth sub-loop. Note that, in an extreme scenario, it is possible that each loop iteration has a particular profitable recomputation version and there is no common profitable version between any two iterations. To capture such scenarios, one needs to unroll the entire loop and perform recomputation analysis on each iteration separately, which can incur significant space and compilation time overheads. On the other hand, one can choose to generate a single (common) version (based on the output of Algorithm 1) for all iterations of the loop nest. Although, in the latter case, the overheads would be less, that option may not be able to take full advantage of recomputation. To strike a balance between the overheads and recomputation benefits, we partition the loop nest into n sub-loops. The partitioning (unrolling) factor n is an input to Algorithm 2 along with the target loop nest. A larger value of n indicates a finer granular application of our approach. The corresponding costs, both in terms of code analysis and code generation, are also high when using a larger value for n. Later, we provide experimental results with different values for n. However, independent of the choice of n, our algorithm merges all sub-loops (line 26 in Algorithm 2) if they prefer the same profitable version after applying the version identification and dependences allow such a merge. As a result, all the sub-loops with the same profitable version generate the optimized code only once, effectively reducing the code generation overheads.

For each sub-loop, we apply cache miss equations (CME) [23] to predict whether a particular reference hits in the cache or not. Specifically, CME iterates over loop statements in a sub-loop one by one. For each reuse vector[3] originating from each data reference in a statement, CME generates

[3]Reuse vectors capture the repeated memory access patterns in loop-based code segments [64]. Specifically, given two loop iterations $\vec{i}$ and $\vec{j}$, where $\vec{i} \neq \vec{j}$ and $\vec{i}$ is executed before $\vec{j}$. If a memory address is accessed by both iterations, the reuse vector is $\vec{r} = \vec{i} - \vec{j}$.

Algorithm 2 Identifying the most profitable recomputation based version of a given input loop nest. Note that this algorithm is invoked for each loop nest of an application separately.

```
INPUT: A loop nest L, A partition factor n
OUTPUT: Most profitable version
 1: /**Step 0: call Algorithm 1 and get all versions*/
 2: all_versions ← generate_recomputable_versions(L)
 3: min_cost ← ∞
 4: /** Step 1: participation the loop nest into n sub-loops */
 5: sub-loops ← partition (L, n)
 6: /** Step 2: find the profitable version of a sub-loop */
 7: for each sub-loop do
 8:     for each version V_k in all_versions of a sub-loop do
 9:         cost_{V_k} ← 0
10:         /**call cache miss equations on V_k*/
11:         for each statement S_i in V_k do
12:             for each reuse vector of each reference in S_i do
13:                 build and solve cache miss equations
14:                 cost_{V_k} += cost_{S_i}
15:                 if cost_{V_k} > min_cost then
16:                     /** partial cost larger than previous versions,
17:                     abandon this version and move to next*/
18:                     goto line 8
19:                 end if
20:             end for
21:         end for
22:         min_cost ← cost_{V_k}
23:     end for
24: end for
25: /** Step 3: merge sub-loops with same profitable version */
26: merge(sub-loops)
```

several linear Diophantine equations and predicts the hit/miss status of the data reference based on the solutions of the Diophantine equations. We integrate our profitable version identification algorithm with the CME process. The pseudo-code for our compiler algorithm is provided in Algorithm 2. We first generate all the versions using Algorithm 1, and partition the loop nest to be optimized into sub-loops. We then apply CME analysis to each version of each sub-loop. During the CME process of a sub-loop, for each statement, we log the cost after CME provides the hit/miss status. If the total cost is greater than the minimum cost across all the previously-searched versions, we stop further exploring this current version and move to the next version (lines 15 to 18). This is due to the fact that the cost of statements in a version of a given sub-loop body monotonically increases as its statements are being processed. This particular characteristic allows us apply the branch and bound concept to significantly reduce the search space and associated search overheads.

The complexity of our compiler algorithm is $O(SVR)$, where S is the total number of statements in the loop nest, V is the total number of variables in the loop nest, and R is the total number of versions of all sub-loops. We observed that, for our benchmark programs, our approach increased compilation time between 5% and 55%, over the compilation of the original codes without our optimization.

To better explain Algorithm 2, let us consider the example shown in Figure 19. Figure 19a shows the code of a sub-loop where $a[i]$ is modified in statement S_x and reused in statements S_y and S_z. As one can see, $a[i]$ in both S_y and S_z can be recomputed using S_x, resulting in a total of four versions. Figure 19b shows all four versions of the code, where each path along the arrow constitutes one version of the sub-loop. Note that, version ① and version ② share the same path between S_x and S_y, indicating that they have the same versions for S_x and S_y but different versions for statement S_z. Algorithm 2 first generates all four versions (using Algorithm 1) and then applies CME analysis to each version separately. To be more concrete, let us assume that applying CME to version ① gives us the cost of $2L2 + 4L1 + 3\oplus$. We next apply CME to version ②, and the resulting cost is $4L2 + 3L1 + 4\oplus$, due to $b[i]$ and $c[i]$ being present in the L2 cache. As a result, version ① has a

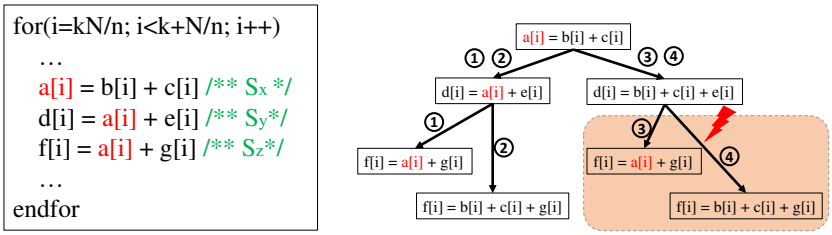

Fig. 19. Branch and bound employed for identifying profitable versions.

lower cost and we temporally choose version ① as our preferred version. We next move to explore version ③. When CME processes statement S_y in version ③, the cost after processing S_y ends up being $4L2 + L1 + 3\oplus$, which is already larger than that of version ①. Therefore, we do not need to further process the branches of versions ③ and ④ (annotated using a shaded box in Figure 19b). In other words, we "prune" the search for the versions whose intermediate costs are already larger compared to the version with the current minimum cost (i.e., most profitable version).

Let us further elaborate on the role CME plays in our approach. At a high level, considering (one by one) the reuse vectors representing the difference (in vector form) between the iterations that reuse data in a lexicographically increasing order, CME determines for each reuse vector the "hits" and "misses" it incurs. More specifically, starting with the most conservative estimate (all accesses are misses), it uses reuse vectors one by one to identify the hits (due to reuses) and reduces the set of misses. Consider, as an example, the simple code fragment below:

$$\text{for}(i = 2; i <= N; i++)\ b[i] = a[i] + a[i-1]\ \text{endfor}$$

In this case, each reference has spatial reuse and in addition $a[i]$ and $a[i-1]$ have temporal reuse between them (i.e., a data element accessed by the former at iteration $i = k$ is accessed by the latter in iteration $i = k + 1$). Conservatively, we can start by assuming $3(N-1)$ misses as we have 3 references accessed in a total of $(N-1)$ iterations. However, CME realizes that, given a cache line size of L, $b[i]$ will miss only once per cache line resulting a total of $(N-1)/L$ misses (instead of our conservative estimation, $(N-1)$). A similar argument goes for $a[i]$ as well, resulting only in $(N-1)/L$ additional misses. Finally, $a[i-1]$ will "reuse" (in the current iteration) any data brought to the cache by $a[i]$ in the previous iteration, incurring potentially only one additional miss (when accessing $a[0]$). As a result, CME would estimate the total number of misses in this case as $\{2(N-1)/L\} + 1$, a much lower value than our conservative bound $(3(N-1))$.

Note that, in a compiler implementation, the reuse opportunities mentioned above are represented as reuse vectors and the cache misses are represented as Diophantine equations. While in the most general case finding solutions to Diophantine equations can be costly, there exist many techniques for manipulating them (see [23] for a discussion on that) and such techniques allow us reduce the complexity and/or number of possible solutions significantly in most practical cases.

Note that CME cannot be fully accurate as it (i) conservatively assumes that any reuse that that cannot be fully analyzed statically (e.g., due to indexed-array subscripts or pointer accesses) will not cause any reduction in the number of misses and (ii) has limitations in capturing the conflict and coherence misses, especially in caches shared by multiple cores. As a result, its accuracy (in predicting hits and misses) is less than 100%. Consider for example the code fragment below:

$$\text{for}(i = 2; i <= N; i++)\ c[i] = d[f[i]] + a[i] * b[i]\ \text{endfor}$$

As an example of the reason (i) for inaccuracy mentioned above, we have an indirect array access: $d[f[i]]$, that is, the result of array access $f[i]$ is used to index another array (d). In most of the cases where such accesses appear, contents of f are not know until runtime. As a result, while CME

will be able to estimate the misses for $f[i]$ (as i itself is compile-time analyzable), it will not be able to estimate the misses for $d[f[i]]$ (as $f[i]$ is not compile-time analyzable). Now, we turn our attention to an example for reason (ii) mentioned above as a potential source of inaccuracy. In the code fragment shown above, while one would expect the misses that would be incurred by $a[i]$ and $b[i]$ to be easily estimatable, it is possible that accesses via these two references can either fully or partially conflict in cache (for example, $b[i]$ can displace $a[i]$ from the cache, thereby destroying the potential spatial reuse of the other elements that reside in the same cache line as $a[i]$ and would be accessed soon).[4] While our compiler also uses array padding [52] to minimize the possibility of conflict misses as much as possible, it is still possible that some hard-to-predict conflict misses will escape our analysis.

Finally, it also needs to be emphasized that the only role CME play in our framework is to "estimate" the number of cache misses for a given version, and if desired, CME can be replaced with any other cache miss estimation strategy.

7.2 Interaction with Spatial Reuse

The interaction of our approach with spatial reuse demands further discussion. In particular, recall that, when we decide to recompute the value of a data element a in a statement, we do not access it from its original location (we instead access the elements that are needed to recompute its value). However, doing so also raises a concern regarding spatial reuse. More specifically, the data elements that reside in the same cache line/memory block as a are not accessed either. And, it is possible that such elements would be used (when they are brought into L1, for example) in the following computations. As a result, theoretically speaking, recomputing a can lead to some loss in data locality originating from spatial reuses. This can be problematic, particularly in pure streaming applications, and as a result, we expect our approach not to be the best optimization strategy for such applications.

However, we also want to emphasize that, our approach explained above is built upon the high level idea of generating different versions of the code fragment being optimized and pruning them based on the feedback from the cache miss equations (CME). As a result, any "loss" in spatial locality incurred as a result of applying recomputation will be caught by CME (i.e., we will observe increased misses) and the code versions that lead to excessive losses in locality will be automatically discarded by our approach. It also needs to be mentioned that all the results presented below for our compiler approach already include all impacts (both positive and negative) of recomputation.

7.3 Results

We now experimentally analyze the behavior of this compiler-based approach, and compare its results to the best results (presented earlier in Section 6). Clearly, since the results presented earlier in the first bar in Figure 18 are obtained after evaluating all possible versions for all loop nests in our application programs, the compiler-determined version for a given loop nest must definitely be among them. The important question is then whether, for a given loop nest, the compiler-determined version and the best version detected via exhaustive search are the same, and if not, how close they are to each other. In the experiments below, unless stated otherwise, the n value is 10, that is, each loop nest is divided into 10 sub-loop nests. Later, in a separate experiment, we vary the value of n and perform a sensitivity study.

For each application, the third bar in Figure 18 gives the normalized execution time for the *compiler-optimized* version. It can be seen that, the compiler version generates performance improvements, ranging between 0.9% and 66.9%, averaging on 36.9%. To better explain these results, we

[4] In fact, in the worst case, all of the accesses to $a[i]$ and $b[i]$ can result in misses, as a result of the ping-pong effect in cache.

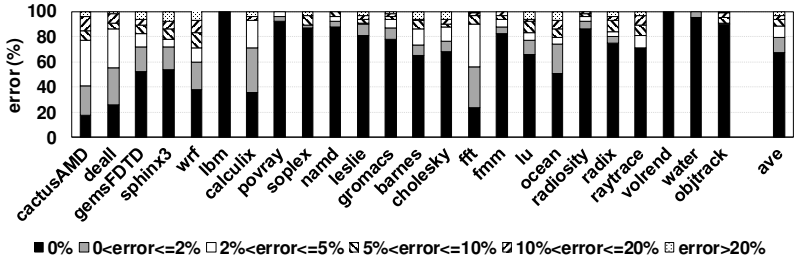

Fig. 20. The error of the compiler based approach experiences in identifying the best version of a loop nest.

give in Figure 20 the errors the compiler based approach experiences in identifying the best version of a loop nest in a given application program. More specifically, in this plot, for an application, the portion marked using "0%" represents the fraction of loop nests for which the compiler identified version and the (experimentally) best version are exactly the same – the compiler identifies the best version with 100% accuracy (i.e., 0% error). On the other hand, "x% < error ≤ y%" captures the fraction of cases (loop nests) where the compiler-generated version is not the same as the actual best version, and the performance difference between them is greater than x% and less than (or equal to) y%. Finally, "error > 20%" represents the fraction of the cases where the compiler version is more than 20% worse (performance-wise) than the optimal (ideal) version. As can be seen from the last bar, on an average, our compiler analysis catches the best version (of a loop nest) in 67.6% of the cases. And, in 79.3% of the cases, the difference between the compiler version and best version is less than or equal to 2%, indicating that, overall, the compiler is quite successful in identifying the most profitable recomputation opportunities automatically.

Let us now discuss why our compiler approach identifies the best version(s) in some benchmarks quite well, while it is relatively less successful in some other benchmarks. It is important to re-emphasize that our compiler approach mainly relies on CME to identify the final code version to use. Consequently, in cases (loop nests) where our compiler approach is not very successful, the main reason is the inaccuracy in cache miss estimation. We present in Table 2 the CME accuracies for our benchmark programs. It can be observed that the CME accuracy values vary significantly across the benchmarks, being as high as 93.3% in *namd* and as low as 69.8% in *deall*.

To better understand why CME generates different accuracies for different benchmarks, we performed another set of experiments. Specifically, for each benchmark in our experimental suite, we checked whether the hit/miss estimation made by the compiler matched the reality, that is, whether the compiler correctly guessed whether a given access would be a hit or a miss. After recording this information, we went over the accesses for which we have misprediction and mapped them to the static data accesses in the code. Also, for the references CME predicted hit but they turned out to be miss in reality, we also recorded the type of the miss (cold, capacity, conflict and coherence). We then tried to identify the reason why such a data access might have been mispredicted.

Recall that we discussed two main reasons in Section 7.1 as potential sources of inaccuracy in miss estimation. Hence, we divided our mispredicted data accesses into three categories: (i) misprediction do to the difficulty in compile-time analyzability of a data reference (e.g., an indexed array access such as $c[f[i]]$, or a pointer-based access such as $*g$ where g is a pointer); (ii) misprediction due to potential conflict or coherence misses; and (iii) some other reason we could not figure out. In the last category are the mispredictions that we could not identify any particular reason why CME mispredicted. The breakdown of CME mispredictions (errors) into these three broad categories is given in Figure 21.

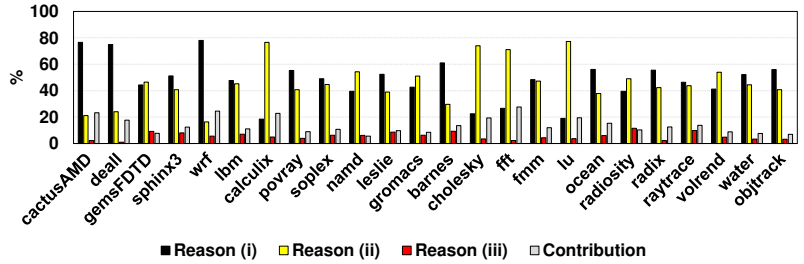

Fig. 21. The breakdown of CME mispredictions (errors) into these three broad categories.

Table 2. Cache miss estimation accuracies.

cactusAMD	74.5%	deall	69.8%	gemsFDTD	83.4%	sphinx3	85.6%	wrf	79.4%
lbm	87%	calculix	80.6%	povray	92.2%	soplex	89.1%	namd	93.3%
leslie	91.7%	gromacs	92%	barnes	85.5%	cholesky	83.3%	fft	76.3%
fmm	90.8%	lu	86%	ocean	83.5%	radiosity	91.5%	radix	84.1%
raytrace	85.8%	volrend	90.6%	water	92.4%	objtrack	93.1%		

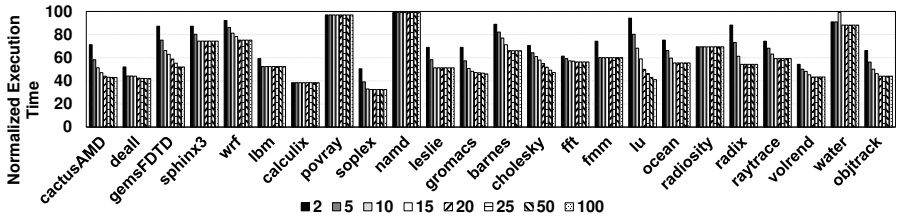

Fig. 22. Sensitivity of savings to the value of n.

It can be observed from this plot that, on average, the contributions of reasons (i), (ii) and (iii) to inaccuracy (mispredictions) are 48.1%, 46.3% and 5.6%, respectively. More specifically, not being able to fully analyze a reference (data access) in the program is the primary reason why the overall error the compiler generates in identifying the best version is relatively high in benchmarks such as *wrf*, *cactusAMD* and *deall*. In comparison, it seems that benchmarks such as *calculix*, *cholesky*, *lu*, and *fft* suffer largely from inaccuracies stemming from conflict and coherence misses. The last bar in Figure 21 gives the contribution of the such problematic references (data accesses that fall into (i), (ii) and (iii)) to the total number of references, and we see that, in most of the benchmarks, this value is less than 15% (averaging on 13.7%).

Recall from Section 5 that, in executing our multi-threaded applications, we executed one application at a time in *all* cores (i.e., one thread per core). Since our recomputation-based approach can replace, in general, a given costly data access with multiple (less costly) data accesses, it can potentially increase the traffic on the *on-chip network*. While unfortunately we do not have a direct way of measuring this overhead accurately on Intel KNL, we do not expect it to be a major issue, due to the high bandwidth provided by the on-chip network. In any case, this potential impact on on-chip network (like all other overheads our approach could bring) is already included in the experimental results presented earlier in Figure 18.

We now study the sensitivity of our savings to the value of n. The results are presented in Figure 22 for n values ranging between 2 and 100 (recall that the default value used so far in our experiments is 10). It can be observed that, beyond a certain value of n, the improvements achieved do not change much. This value, while application dependent is generally between 10 and 20. The reason for this behavior, is that, beyond a certain level of unrolling, there is not much additional recomputation opportunity left. To show how some recomputation opportunities may disappear

as n is increased, let us consider the following sample loop nest that iterates only 4 times (for illustrative purposes):

for($i = 2; i <= 5; i++$)
  $a[i] = b[i] + c[i]$
  $... = a[i-1] + ...$
  $... = a[i-2] + ...$
endfor

Let us first assume that n is 2. In this case (after unrolling), the program statements assigned to the first sub-loop are as follows:

$a[2] = b[2] + c[2]$
$... = a[1] + ...$
$... = a[0] + ...$
$a[3] = b[3] + c[3]$
$... = a[2] + ...$
$... = a[1] + ...$

It is to be observed that $a[2]$ in the fifth statement above presents a potential recomputation opportunity as can be replaced, if it is beneficial to do so, by $b[2] + c[2]$ from the first program statement. Similarly, the program statements assigned to the second sub-loop are as follows:

$a[4] = b[4] + c[4]$
$... = a[3] + ...$
$... = a[2] + ...$
$a[5] = b[5] + c[5]$
$... = a[4] + ...$
$... = a[3] + ...$

Similar to the previous sub-loop, here we have a potential recomputation opportunity for $a[4]$, i.e., it can be replaced by $b[4] + c[4]$.

Let us now increase n from 2 to 4. In this case the first sub-loop will have the following statements:

$a[2] = b[2] + c[2]$
$... = a[1] + ...$
$... = a[0] + ...$

Unfortunately, this version does not present any recomputation opportunity, and the program statements assigned to other sub-loops also do not present any recomputation opportunities. In general, beyond a certain value of n, there may not be additional recomputation opportunities.

Overall, while the results in Figure 22 suggest using large n values (increased unrolling) as already stated, doing so increases both the code size and compilation time. For example, moving from 10 to 100, increased the code size by around 7x and compilation time by around 1.7x. Based on these results, we recommend using large n values if neither compilation time nor code size is a prime concern.

Now, we focus on four of our applications (lbm, calculix, barnes, and cholesky) where we could confidently change the input size, and measure how the compiler-based approach performs as the dataset sizes increase. The results plotted in Figure 23[5] indicate that the effectiveness of our compiler based approach significantly increases as the dataset sizes increase. This is mainly because,

[5]On x-axis of this plot, as we go from one point to another, the input size is nearly doubled. Note also that, each result in this graph is *normalized* to the original version with the corresponding input size.

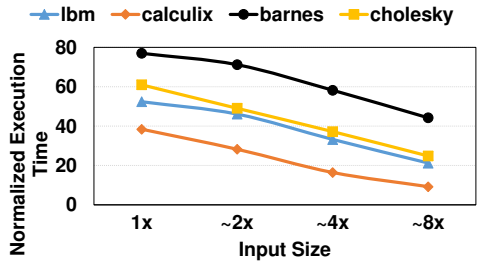

Fig. 23. Normalized execution time with different input sizes.

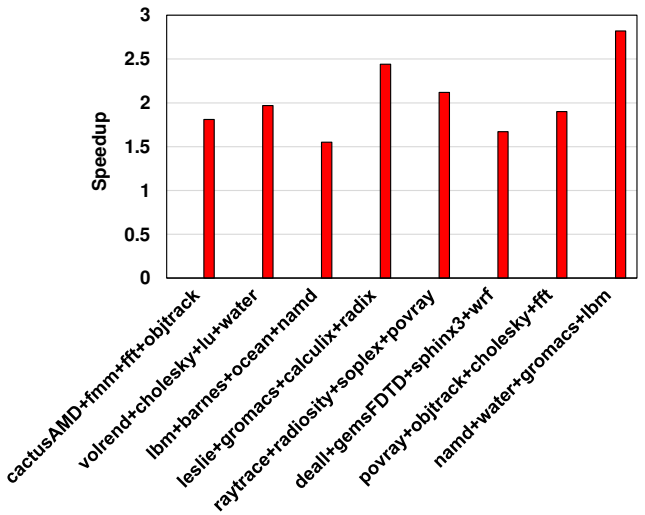

Fig. 24. Results when running multiple multithreaded applications.

with larger dataset sizes, the application makes more costly data accesses which increases the opportunities for employing recomputation.

We also performed experiments with multiple applications running at the same time. More specifically, from our benchmark programs, we formed different workloads, each having 4 applications. Each application is assigned to 9 nodes of our 36 node manycore system. We first executed the workload without our recomputation optimization, and then with our optimization. We then calculated the geometric mean of the speedups of the four applications, with respect to the first execution. The results (geometric mean of individual application speedups) are plotted in Figure 24 for 8 different workloads. It can be observed that, our approach brings improvements ranging between 1.55 and 2.82, that is, it is also effective in execution environments that run multiprogrammed workloads of multithreaded applications.

8 IMPROVING RECOMPUTATION OPPORTUNITIES

So far in our discussion we focused on the original application programs and performed two types of studies: one measuring the recomputability potential of the programs and one that proposes a practical (compiler-based) scheme to take advantage of recomputability. In this section, we take a different approach and explore whether it is possible to modify the application programs to *improve* their recomputability.

Algorithm 3 Improve recomputability.

INPUT: A loop nest L
OUTPUT: Loop nest with improved recomputability.
for each statement S_i in L **do**
  $versions_{S_i} \leftarrow \emptyset$
  /** backward scan */
  for each variable V_x in S_i **do**
    $set_{V_x} \leftarrow V_x$
    Backward scan the nearest assignment of V_x and find statement S_k
    for Each version P_k in $versions_{S_k}$ **do**
      Forward scan from S_k to S_i
      if there exist a statement S_j updates variable V_y in S_k **then**
        insert statement $V_{temp} \leftarrow V_y$ before S_j
        $P_k^{'} \leftarrow$ replace V_y in P_k using V_{temp}
        $set_{v_x} \cup P_k^{'}$
      end if
    end for
  end for
  for all variables $V_x \cdots V_y$ in S_i **do**
    from variable sets set_{v_x} to set_{v_y}
    find and add all combinations to $versions_{S_i}$
  end for
end for
Generate all the recomputable versions
for all the statements $S_i \cdots S_j$ **do**
  from $versions_{S_i}$ to $versions_{S_j}$
  find all combinations where each combination is one output version of the loop nest.
end for

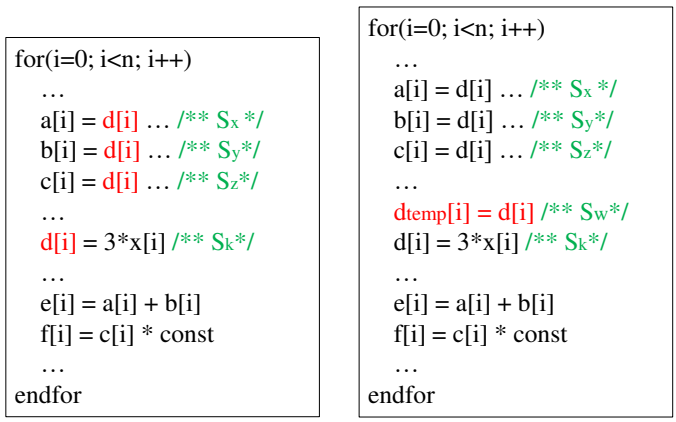

Fig. 25. Improving recomputability using variable storage/duplication.

8.1 Motivation and Algorithm

Clearly, improving recomputability metric is one of the possible approaches that can improve performance based on the concept of recomputability. Another approach would be, for example, identifying critical variables (i.e., variables that can significantly contribute to the recomputation of many costly data accesses) and pinning them in the first layers of cache (L1 and L2). Instead, in this paper, we focus on code level modifications to improve chances for recomputation, and postpone the cache pinning idea to a future work.

Recall that Figure 8 gives recomputability for our benchmark programs. Our goal in this section is to improve these numbers by ensuring that "critical values" are stored long enough until all the recomputations that need them are completed. In this context, a critical value is a value that

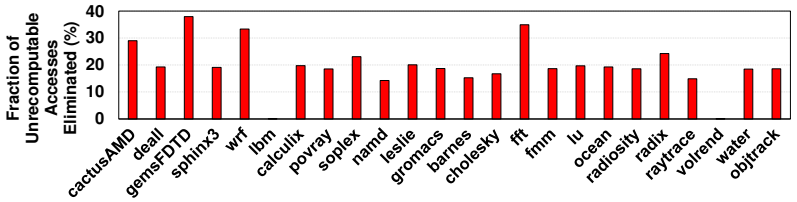

Fig. 26. The increased fraction of data accesses that are recomputable in the optimized applications.

would contribute to the recomputation of several other values. Let us consider as an example the code fragment shown in Figure 25. In this code fragment (taken from one of our benchmarks), we cannot recompute the values of $a[i]$, $b[i]$ or $c[i]$ simply, because one of the values needed for recomputation ($d[i]$) has been overwritten. Our solution is to store the value of $d[i]$ in a temporary location before it is overwritten and use that value later (when needed) for the recomputations of $a[i]$, $b[i]$ and $c[i]$. We want to emphasize that, for this optimization to be successful, there are two conditions: i) copy should be done before the variable being overwritten, and ii) overwritten variable should have some reuse in successive statements.

Algorithm 3 gives the pseudo-code for our compiler algorithm that improves recomputability using variable duplication.[6] The overall process is similar to Algorithm 1 and it also consists of a backward scan and a forward scan. The fundamental difference between Algorithm 3 and Algorithm 1 is that, if there is an update to a critical variable, Algorithm 3 tries to insert a copy of an originally unrecomputable variable (lines 9 to 12), instead of simply abandoning the recomputation which is the decision made in Algorithm 1. In other words, by storing the variables, this approach increases the number of possible recomputation versions, thereby improving the value of the recomputability metric. It is important to emphasize that, Algorithm 3 only improves recomputability and does *not* guarantee the additional recomputation based versions are indeed profitable. It is possible that the additional versions obtained from Algorithm 3 are costlier due to different location maps as well as overheads of extra statements added. All the versions generated from Algorithm 3 and Algorithm 1 are input to Algorithm 2, and Algorithm 2 automatically identifies the most profitable recomputation version (i.e., the version with minimum cost).

8.2 Results

We implemented this optimization in our compiler and evaluated its impact on recomputability and performance for our benchmark programs.[7] The results given as the last bar for each benchmark in Figure 18 indicate that, this optimization improves performance by 40.1% on average, and as much as 67.4% and 65.8% in benchmarks *soplex* and *calculix*. To explain these results better, we present in Figure 26 the fraction of data accesses that were unrecomputable in the original applications but are now recomputable in the optimized applications. It can be seen from these results that our optimization approach eliminates 19.7% of the unrecomputable references in the original applications, which is why it generates further improvements over the compiler support presented in Section 7.

9 EMPIRICAL COMPARISON AGAINST NDC

In this section, we present a quantitative comparison of our CND against NDC (recall that a qualitative comparison is given in Section 4). Since NDC cannot be mimicked in a real hardware, we pursued a *simulation-based* study instead. More specifically, using gem5 [14], we modeled an

[6]The asymptotic complexity of this algorithm is $O(S^2V)$, which is the same as Algorithm 1.
[7]We want to make it clear that this enhancement is applied before our compiler approach discussed in Section 7. In other words, the scheme evaluated in this section also makes use of the compiler approach discussed earlier.

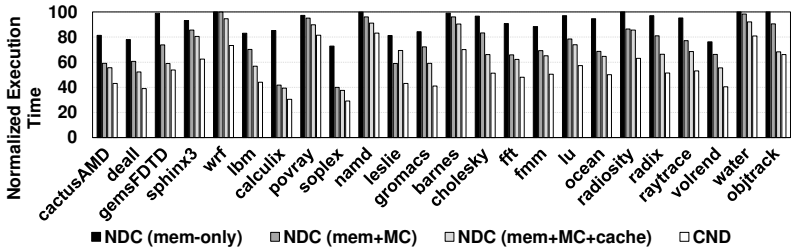

Fig. 27. The normalized execution times of our applications with four different schemes: NDC (mem-only), NDC (mem+MC), NDC (mem+MC+cache), and our CND (with n=10).

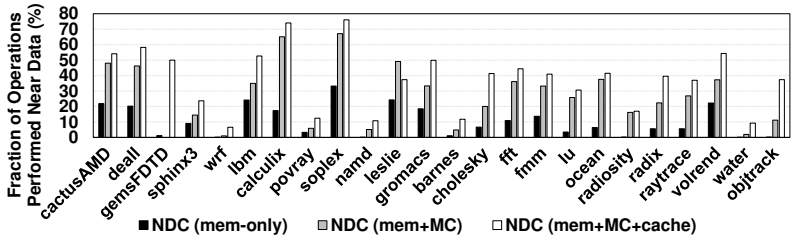

Fig. 28. The opportunities caught by three different NDC-based schemes.

architecture, which is very similar to Intel KNL. gem5 is a highly configurable simulation framework, which supports multiple ISAs, a large set of CPU models, a detailed cache/memory subsystem, including support for various types of cache coherence protocols and on-chip interconnects.

Clearly, there exist many different approaches to NDC in the literature. In this work however, we implemented three different NDC approaches with increasing strengths. The first of these, called "NDC (mem-only)", performs an arithmetic or logic computation in a memory bank if the most updated values of all the operands required by the computation already reside in that bank. The second approach, referred to as called "NDC (mem+MC)", includes NDC (mem-only), but in addition it also performs an arithmetic or logic computation in a memory controller (MC) if all the involved operands reside in the banks controlled by that MC. In the third approach, called "NDC (mem+MC+cache)", we also capture the cases (opportunities) where all the required operands are in the same L2 or they are all in the same L3 (in addition to those opportunities caught by NDC (mem+MC)). Here, the difference between NDC (mem-only) and NDC (mem+MC) should be noted. If two operands are needed for an operation, reside in two different banks controlled by the *same* memory controller, the latter scheme would take advantage of that, whereas the former scheme could not.

We want to emphasize three critical points. First, our evaluation of these schemes does *not* include any overheads. In reality, performing computations in memory banks, memory controllers and (L2/L3) caches would involve some extra latency; so, the results reported below overestimate the benefits of NDC. Second, we assume that all types of arithmetic and logic operations can be performed in an NDC-manner. Again, in reality, one would restrict the types of operations. Third, modifying hardware to include units to perform computations (e.g., enhancing an MC with an ALU) can have significant area costs as well, which is in fact one of the factors that limit the widespread adoption of the NDC technology, in our opinion. In a sense, the results from these three NDC schemes described above represent the "upper bounds" that can be achieved by an *ideal* implementation and execution scenario.

Figure 27 plots the execution times of our application programs with four different schemes: NDC (mem-only), NDC (mem+MC), NDC (mem+MC+cache), and our CND (with n=10).[8] These execution times represent *normalized* values with respect to the original application codes. Clearly, as the strength of an NDC mechanism increased, i.e., as we move from NDC (mem-only) to NDC (mem+MC) to NDC (mem+MC+cache), the improvements achieved also increase. More specifically, compared to the original executions, NDC (mem-only), NDC (mem+MC), NDC (mem+MC+cache) bring average execution time improvements of 8.7%, 24.4% and 31.5%, respectively.[9] More importantly, even these ideal NDC implementations cannot outperform our CND, primarily because our approach catches more opportunities to reduce data movement costs.[10] To elaborate more on this last point, we present in Figure 28 the opportunities caught by the three NDC based schemes explained above. More specifically, the y-axis in this figure represents the fraction of arithmetic or logic operations that could be performed near data (in a memory bank, memory controller or cache, depending on the specific scheme in question). We see that, on average, only 10.4%, 27.9% and 37.9% of all operations could be performed near data, when using, respectively, NDC (mem-only), NDC (mem+MC), NDC (mem+MC+cache). This result helps us explain the not-so-good execution time results of these three schemes plotted in Figure 27.

10 DISCUSSION OF RELATED WORK

In this section, we discuss the prior research efforts that focus on reducing data access and movement latencies. We classify the relevant research works into three broad categories: i) data locality optimizations, ii) near data computing (NDC) based studies, and iii) works related to recomputation.

Data Locality Optimizations: Optimizing data accesses has generated in the past a lot of different optimization strategies, but most of these strategies employ either code (access pattern) restructuring [11, 17, 26, 32, 34, 35, 41, 56, 62, 63, 65], data layout reorganization [20, 22, 31, 39, 45, 49], or a combination of both. CND is clearly different from these past approaches as it employs an entirely different strategy based on the concept of *recomputation*. However, it can co-exist with such data access optimization schemes, or can be integrated with them, if desired.

Near Data Computing Based Optimizations: The concept of NDC is not new and can be traced back to 1970s [24, 57]. NDC organizes execution differently from the traditional approaches mentioned in the previous paragraph. More specifically, it brings computation to data, instead of the other way around. Recently, due to emerging new technologies such as 3D-stacked memory [44] and hybrid memory cube [30], NDC has become more popular, and now there exist a large number of proposals aimed at exploiting NDC in various ways [7–9, 12, 18, 29, 42, 48, 68]. Ahn et al. [8] proposed a processing-in-memory architecture where data locality is considered while executing PIM instructions. Pattnaik et al. [50] proposed a computation offloading model on GPUs where the GPUs include PIM cores. Hsieh et al. [29] proposed a scheme to dynamically offload beneficial code segments to PIMs without involving programmer efforts. Xu et al. [67] used PIM architectures to improve the execution of deep learning algorithms. As discussed in Section 4 and quantified in Section 9, CND is more practical than NDC, and generates better results in general.

[8]Note that, for this comparison, we use the CND version without our enhancement discussed in Section 8. We chose the version that does not use our enhancement, because, in principle, one can also imagine compiler techniques that can enhance the effectiveness of NDC, though this itself is a new research direction.

[9]Note that this is partly because our implementations of these three schemes do not assume any latency overhead due to i) the added hardware units or ii) the time it takes to determine that all of the required operands are indeed locally available. If these overheads are added, the relative ranking of these three schemes can potentially change.

[10]Note that the savings reported in this graph for CND are different from those reported earlier in Figure 18, as those results were from the KNL system and these results are from the gem5 simulation.

Works Related to Recomputation: We now discuss two prior works related to recomputation. Akturk and Karpuzcu [10] investigated the effectiveness of recomputing data values in minimizing the energy overhead of expensive off-chip memory accesses. Based on their observations, they proposed an architecture-supported recomputation strategy called amnesic. Our approach is different compared to their approach from two main aspects. First, their approach is simulation based and requires extra hardware support such as recomputation registers and additional buffers. In contrast, our proposed CND is a pure software approach and is evaluated on a real manycore system (KNL). Second, our proposed enhancement (Section 8) can further improve the recomputability in applications, thereby improving the benefits one can achieve from recomputation. Another related work to recomputation is [6]. Targeting cloud platforms, the authors observed that storing a result and regenerating a result is not simply a cost-benefit tradeoff. It is also related to information security, data feasibility, etc, due to the unique characteristics of cloud platforms. To this end, the authors gave a detailed analysis of recomputation in cloud platforms. Our approach in this paper focuses on manycore systems and our primary target is workload performance.

11 CONCLUSIONS AND FUTURE WORK

This work makes four main contributions. First, targeting emerging manycore systems, it proposes a novel computation paradigm called CND (which is based on recomputing the values of costly data elements), and quantifies its potential using a diverse set of sequential and parallel application programs. Second, it presents a practical compiler-driven strategy to take advantage of CND. Third, it proposes a strategy to increase the chances for employing CND. And finally, it compares CND with another recent paradigm, near data computing (NDC), in both qualitative and quantitative terms. Our experimental evaluations on a state-of-the-art manycore system reveal that CND can be very effective in reducing data access/movement costs, and improves, on average, the execution time by 40.1% across our programs including both single-threaded and multi-threaded applications. Further, our simulation based experiments indicate that CND outperforms various (idealistic) implementations of NDC.

While in this work, we evaluated the effectiveness of our approach in multicore case as well, one can further enhance our approach by considering unique opportunities a multithreaded execution can present. For example, while one of the cores can be closer to a subset of data elements (required for recomputation), the other core can be closer to another subset of data elements. In such scenarios, determining the ideal location (core) to perform recomputations can be quite challenging, but also very rewarding, if done carefully. In our future work, we will investigate such additional multicore/multithreaded execution specific opportunities for recomputation and also experiment with other types of benchmarks (e.g., database and cloud workloads). Finally, we will also explore the idea of pinning critical values (i.e., values that can be used in future recomputations) in fast cache memories (e.g., L1 and L2 caches).

ACKNOWLEDGMENT

We thank Evgenia Smirni for shepherding our paper. We also thank the anonymous reviewers for their constructive feedback. Myoungsoo Jung is in part supported by NRF 2016R1C1B2015312, DOE DEAC02-05CH11231, IITP-2018-2017-0-01015, NRF 2015M3C4A7065645, Yonsei Future Research Grant (2017-22-0105) and Samsung grant (2018). This research is supported in part by NSF grants #1526750, #1763681, #1439057, #1439021, #1629129, #1409095, #1626251, #1629915, and a grant from Intel.

REFERENCES

[1] 2006. SPEC CPU 2006. https://www.spec.org/cpu2006/.

[2] 2014. 7-Zip LZMA Benchmark. Intel Bradwell. https://www.7-cpu.com/cpu/Broadwell.html.
[3] 2015. 7-Zip LZMA Benchmark. Intel Skylake X. https://www.7-cpu.com/cpu/Skylake_X.html.
[4] 2017. 7-Zip LZMA Benchmark. AMD Zen. https://www.7-cpu.com/cpu/Zen.html.
[5] 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. https://arxiv.org/pdf/1804.06826.pdf.
[6] Ian F. Adams, Darrell D. E. Long, Ethan L. Miller, Shankar Pasupathy, and Mark W. Storer. 2009. Maximizing Efficiency by Trading Storage for Computation. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*.
[7] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*.
[8] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*.
[9] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data Reorganization in Memory Using 3D-stacked DRAM. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*.
[10] Ismail Akturk and Ulya R. Karpuzcu. 2017. AMNESIAC: Amnesic Automatic Computer. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.
[11] Jennifer M. Anderson and Monica S. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*.
[12] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 Workshop. *Micro, IEEE* (2014).
[13] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling. In *Proceedings of the 21st international symposium on High Performance Computer Architecture (HPCA)*.
[14] 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).
[15] Uday Bondhugula, J. Ramanujam, and et al. 2008. PLuTo: A practical and fully automatic polyhedral program optimization system. In *Proceedings of Programming Language Design And Implementation (PLDI)*.
[16] Amirali Boroumand, Saugata Ghose, Brandon Lucia, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. 2016. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *IEEE Computer Architecture Letters* (2016).
[17] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[18] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. 1999. Impulse: Building a Smarter Memory Controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (ISCA)*.
[19] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*.
[20] MichałCierniak and Wei Li. 1995. Unifying Data and Control Transformations for Distributed Shared-memory Machines. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*.
[21] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. 2012. Application-to-core Mapping Policies to Reduce Memory Interference in Multi-core Systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*.
[22] Wei Ding, Xulong Tang, Mahmut Kandemir, Yuanrui Zhang, and Emre Kultursay. 2015. Optimizing Off-chip Accesses in Multicores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[23] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1998. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, New York, NY, USA, 228–239. https://doi.org/10.1145/291069.291051
[24] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: the Terasys Massively Parallel PIM Array. *IEEE Computer* (1995).
[25] Boris Grot, Stephen W. Keckler, and Onur Mutlu. 2009. Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-chip. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on*

Microarchitecture.

[26] 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*.

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

[28] Hasan Hassan, Gennady Pekhimenko, Nandita Vijaykumar, Vivek Seshadri, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. 2016. ChargeCache: Reducing DRAM latency by exploiting row access locality. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[29] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*.

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

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

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

[33] Mahmut Kandemir, Hui Zhao, Xulong Tang, and Mustafa Karakoy. 2015. Memory Row Reuse Distance and Its Role in Optimizing Application Performance. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

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

[35] Orhan Kislal, Jagadish Kotra, Xulong Tang, Mahmut Taylan Kandemir, and Myoungsoo Jung. 2017. POSTER: Location-Aware Computation Mapping for Manycore Processors.. In *Proceedings of the 2017 International Conference on Parallel Architectures and Compilation*.

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

[37] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[38] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N. Patt. 2009. Improving Memory Bank-level Parallelism in the Presence of Prefetching. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*.

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

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

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

[42] G.J. Lipovski and C. Yu. 1999. The Dynamic Associative Access Memory Chip and Its Application to SIMD Processing and Full-Text Database Retrieval. In *MTDT*.

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

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

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

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

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

[48] R Nair, SF Antao, C Bertolli, P Bose, JR Brunheroto, T Chen, C-Y Cher, CHA Costa, C Evangelinos, and BM Fleischer. 2015. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development* (2015).

[49] M.F.P. O'Boyle and P.M.W. Knijnenburg. 2002. Integrating Loop and Data Transformations for Global Optimization. *J. Parallel Distribute Computer* (2002).
[50] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)*.
[51] Erik Riedel, Christos Faloutsos, and David Nagle. 2000. *Active disk architecture for databases.* Technical Report. DTIC Document.
[52] Gabriel Rivera and Chau-Wen Tseng. 1998. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*.
[53] Jihyun Ryoo, Orhan Kislal, Xulong Tang, and Mahmut Taylan Kandemir. 2018. Quantifying and Optimizing Data Access Parallelism on Manycores. In *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.
[54] Akbar Shrifi, Wei Ding, Diana Guttman, Hui Zhao, Xulong Tang, Mahmut Kandemir, and Chita Das. 2017. DEMM: a Dynamic Energy-saving mechanism for Multicore Memories. In *Proceedings of the 25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.
[55] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* (2016).
[56] Yonghong Song and Zhiyuan Li. 1999. New Tiling Techniques to Improve Cache Temporal Locality. In *PLDI*.
[57] Harold S. Stone. 1970. *IEEE Transaction Computing* (1970).
[58] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. 2016. Improving Bank-Level Parallelism for Irregular Applications. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[59] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[60] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut Kandemir, and Chita Das. 2017. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Computer Architecture (HPCA)*.
[61] Xulong Tang, Ashutosh Pattnaik, Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita R. Das. 2019. Quantifying Data Locality in Dynamic Parallelism in GPUs. In *Proceedings of the 2019 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.
[62] S. Verdoolaege, M. Bruynooghe, G. Janssens, and P. Catthoor. 2003. Multi-dimensional incremental loop fusion for data locality. In *ASAP*.
[63] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *ASPLOS*.
[64] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *PLDI*.
[65] 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).
[66] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of International Symposium on Computer Architecture (ISCA)*.
[67] Lifan Xu, Dong Ping Zhang, and Nuwan Jayasena. 2015. Scaling Deep Learning on Multiple In-Memory Processors. In *Proceedings of the 3rd Workshop on Near-Data Processing*.
[68] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *HPDC*.
[69] Haibo Zhang, Prasanna Venkatesh Rengasamy, Nachiappan Chidambaram Nachiappan, Shulin Zhao, Anand Sivasubramaniam, Mahmut Kandemir, and Chita R. Das. 2018. FLOSS: FLOw Sensitive Scheduling on Mobile Platforms. In *In Proceedings of The Design Automation Conference (DAC)*.
[70] Haibo Zhang, Prasanna Venkatesh Rengasamy, Shulin Zhao, Nachiappan Chidambaram Nachiappan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. 2017. Race-to-sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.
[71] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2002. Breaking Address Mapping Symmetry at Multi-levels of Memory Hierarchy to Reduce DRAM Row-buffer Conflicts. In *The Journal of Instruction-Level Parallelism (JILP)*.

Received August 2018; revised October 2018; accepted December 2018