Quantifying and Optimizing Data Access Parallelism on Manycores

Jihyun Ryoo
Penn State University
jihyun@cse.psu.edu

Orhan Kislal
Penn State University
kislal.orhan@gmail.com

Xulong Tang
Penn State University
xzt102@cse.psu.edu

Mahmut T. Kandemir
Penn State University
kandemir@cse.psu.edu

***Abstract*—Data access parallelism (DAP) indicates how well available hardware resources are utilized by data accesses. This paper investigates four complementary components of data access parallelism in detail: cache-level parallelism (CLP), bank-level parallelism (BLP), network-level parallelism (NLP), and memory controller-level parallelism (MLP). Specifically, we first quantify these four components for a set of 20 multi-threaded benchmark programs, and show that, when executed on a state-of-the-art manycore platform, their original values are quite low compared to the maximum possible values they could take. We next perform a limit study, which indicates that significant performance improvements are possible if the values of these four components of DAP could be maximized. Building upon our observations from this limit study, we then present two practical computation and network access scheduling schemes. Both these schemes make use of profile data, but, while the compiler-based strategy uses fixed priorities of CLP, BLP, NLP, and MLP, the machine learning-based one employs a predictive machine learning model. Our experiments indicate 30.8% and 36.9% performance improvements with the compiler-based and learning-based schemes, respectively. Our results also show that the proposed schemes consistently achieve significant improvements under different values of the major experimental parameters.**

I. Introduction

In today's manycores with large number of cores, network-on-chip (NoC), multiple memory controllers (MCs) and a large number of memory banks, maximizing data access parallelism (i.e., how well hardware resources are used by data accesses) can be as important as maximizing computation parallelism (i.e., how well computations are parallelized). One option along this direction is to simply define data access parallelism as "memory parallelism", the number of concurrent memory operations, and try to maximize that. For example, an early compiler work [52] on memory parallelism tuned iteration space tiling to cluster memory accesses. Works along similar directions include [56], [5], [15], [62], [59], [20], [36], [35].

While such efforts can be successful in certain applications and single-core architectures, given a large variety of emerging manycore systems, an approach that *exposes* architectural details to software can be a more promising option. In fact, instead of working with a high-level concept such as memory parallelism, one may want to dig further and identify its different "components" to better understand its behavior and reshape it for performance benefits, which is the underlying vision of this work.

In an NoC-based manycore with multiple MCs and memory banks, data access parallelism (DAP) can be divided into four components: *cache-level parallelism* (CLP), *bank-level parallelism* (BLP), *network-level parallelism* (NLP), and *memory controller-level parallelism* (MLP). CLP refers to the number of L2 banks[1] that are being accessed at a time when an L2 bank is being accessed. Similarly, BLP captures the number of memory banks being accessed when a bank is being accessed. Clearly, higher values for CLP and BLP indicate higher levels of cache-level and bank-level parallelism, respectively. NLP on the other hand indicates the number of NoC links being exercised when at least one of the NoC links is active. A higher NLP value means that the workload utilizes a larger fraction of NoC. Finally, MLP captures the number of MCs that are being used when one memory controller is active. Similar to the CLP and BLP cases, one would prefer high MLP and NLP values from a resource utilization perspective.

Note that *CLP, BLP, NLP and MLP capture different aspects of DAP*, and optimizing for only one of them does not necessarily lead to good results for the other three. For example, an execution can utilize a large number of LLCs (high CLP), but if all cache misses go to a small set of memory banks, its BLP would be quite low. While there exist prior studies that focus on each of these four components of DAP in isolation (most of the existing studies are hardware based [11], [16], [41], [50], [54], [55], [30], [63], with only a few software-based studies [19], [43]), one can potentially achieve the maximum performance by simultaneously exercising all of them. With this motivation, this paper makes the following **contributions**:

- It quantifies CLP, BLP, NLP and MLP for a set of 20 multi-threaded applications, and shows that their original values are quite low compared to the maximum possible values. In other words, these multi-threaded applications do *not* take advantage of DAP in their original forms.
- It performs a "limit study", where it measures the potential of maximizing CLP, BLP, NLP and MLP in isolation as well as optimizing them together. The results indicate that, when an entire application is considered, CLP and BLP play a bigger role, compared to NLP and MLP, in shaping the overall performance, and optimizing all four components can bring 40.9% performance improvement on average. However, the results also show that, for

[1]We assume an S-NUCA [32] based management of shared L2 as our last-level cache (LLC).

individual loop nests in an application, CLP, BLP, NLP or MLP may be the dominant component.

- It presents two practical computation and network access scheduling schemes that target CLP, BLP, NLP and MLP. Both these schemes make use of "profile data", but, while our compiler strategy uses fixed priorities of CLP, BLP, NLP and MLP, our machine learning (ML) based approach employs a predictive learning model.
- It presents results using these two schemes. The results indicate that the fixed priority based scheme improves, under the default values of our system parameters, CLP, BLP, NLP and MLP by 45.5%, 54.6%, 20.0% and 37.9%, respectively, on an average. These improvements in DAP collectively contribute to a 30.8% performance improvement, when averaged over all 20 programs. Further, the machine learning based scheme improves CLP, BLP, NLP, MLP and execution time by 55.4%, 75.3%, 36.3%, 50.9% and 36.9%, respectively.

The remainder of this paper is structured as follows. Section II introduces the target manycore architecture our study focuses on. Section III explains the computation parallelization strategy assumed by our work. The different components of data access parallelism are elaborated in Section IV. Sections V and VI present the evaluation platform/workloads and experimental results with the original applications. Section VII discusses our results from an ideal (but not implementable) data access parallelism strategy, and Section VIII presents and evaluates two practical data access parallelism optimization schemes: one is purely compiler based and one that employs machine learning. The related work is discussed in Section IX, and finally, the paper is concluded in Section X with a summary of our major findings and a brief discussion of the planned future work.

II. Target Manycore Architecture

Figure 1 shows the layout for an 8×8 network-on-chip (NoC) based manycore with static non-uniform cache architecture (S-NUCA). Each square represents a node that houses a core, a private L1 cache, a unified L2 bank (our last-level cache, LLC), and a router. All L2 banks in the system are "shared" and collectively constitute our LLC. The rectangles (marked with MC) are used to show memory controllers (MCs). These MCs control/schedule the off-chip memory accesses (LLC misses). The arrows represent how the routers are connected to each other and to the MCs. Even though various dynamic message routing solutions exist in the literature, in this work, we focus on an NoC with static routing (more specifically, XY-routing, in which the message is routed first in X or horizontal direction to the correct column, and then in Y or vertical direction to the receiver node/core), since dynamic routing introduces significant performance and energy overheads during execution.[2] Each MC manages a DRAM module, also referred to as DIMM, by issuing commands over

[2]Note that, our NLP optimization changes the underlying routing policy for data accesses. However, it is still a static routing and uses exactly the same number of network links as the XY-routing.

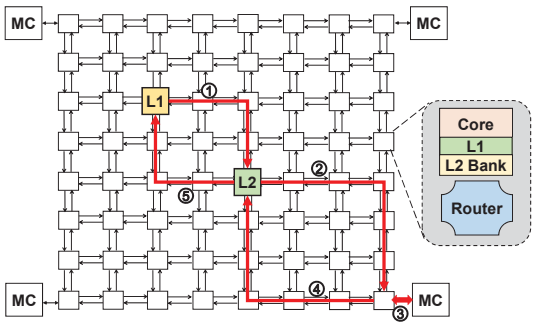


Fig. 1: Representation of an 8×8 NoC based manycore with S-NUCA and a sample memory access flow.

address/data buses (referred to as channel). Each DIMM is made up of multiple *ranks*; each rank consists of multiple *banks*; and, all the banks in a rank share the same timing circuitry. Each bank has a row-buffer, where the memory row is loaded before the data corresponding to the request is sent back over the channel.

As explained by Kim et al. [32], S-NUCA employs static mapping for data. Typically, certain bits of the physical address are used for assigning data to L2 banks and memory banks. There are multiple benefits this cache organization provides. Not enforcing uniformity means that the effective cache capacity will be significantly larger than a uniform cache structure. Since the cache banks are connected to each other, a miss on a local L1 might be mitigated by reading the data from another bank (remote L2), instead of requesting it from the memory controller. S-NUCA reduces the strain on the memory controllers and allows multiple data requests to work in parallel via the on-chip network. We provide the following example to discuss different scenarios that might occur with a typical S-NUCA system. Figure 1 shows an example of memory access flow in our architecture. When an L1 miss occurs, the request is forwarded to the node that accommodates the L2 bank that holds the requested data (①). If the requested data block is found in this home L2 bank of the data, it is read and sent back to the L1 cache in the requesting node (⑤). Otherwise, an L2 miss is said to occur, and a request is sent to the MC that controls that channel to which the bank that holds the requested data is connected (②). This target MC schedules the request and, after reading it from the off-chip memory (③), sends the data back to the L2 home bank (④), and then to the L1 cache (⑤). It is important to emphasize that, there are basically two time-consuming activities involved in a memory access: (i) time spent on traveling the NoC (which is a function of both the number of links between the source and destination as well as the degree of contention on the network) and (ii) time spent on accessing the off-chip memory bank. Clearly, maximizing DAP can reduce and/or hide the latencies of both these activities. Specifically, maximizing CLP and BLP allows more caches and banks to serve a given set of requests, and similarly, high NLP and MLP values mean that the observed network and memory queuing latencies will be reduced.

In this architecture, "address mapping" defines how the

physical address space is distributed across multiple shared components (e.g., L2 banks, memory channels, banks) in a system. Note that each component can have its own mapping strategy. Depending on the mapping scheme employed, a request (physical address) can result in an access to different components. There are various address mapping strategies, and the two widely-used are (i) cache line-level mapping, and (ii) page-level mapping. In the first one, the granularity of distribution is a cache line size, whereas in the second one, it is page size. In most of the experiments reported in this paper, we use a cache line-level distribution of physical addresses across L2 banks, MCs and memory banks.

III. Computation Parallelization

The primary focus of this paper is **data access parallelism** (DAP), and our proposed DAP maximization strategies (which target all four components of DAP) can work with any computation parallelization strategy, which can be "compiler-based" or "user-specified". In the specific computation parallelization strategy adopted in this work, given a loop nest, the compiler first extracts data and control dependencies and then *tiles* the loop nest.[3] The specific tiling strategy used is based on [29]. Note that this strategy is quite flexible and can choose non-rectangular tiles as well, if that strategy improves performance. For each loop nest in each application in our experimental suite, once the "tile shape" is decided using the approach in [29], we experimented with different "tile sizes" and selected the best tile size, i.e., the one that minimizes the overall execution time.[4]

Following iteration space tiling, the loop nest is parallelized. Note that, each "tile" represents a chunk of computations (iterations) assigned to a core for execution. The primary goal in this parallelization is to maximize the number of tiles that can be executed by different cores in parallel. Consequently, the tiles are *assigned* to cores such that inter-core tile dependencies are minimized as much as possible. It is also important to emphasize that, in general, the number of tiles is much larger than the number of cores, and as will be discussed later, this gives the compiler some "flexibility" in scheduling. Figure 2 shows a loop nest, its tiled version, and the assignment of tiles to cores. In the rest of this paper, $T_{c,j}$ indicates the jth tile assigned to core c.[5] The next step following the tile-to-core assignment is to schedule the tiles assigned to cores. We do this in a *DAP-oriented fashion*, as will be explained in the rest of this paper. A unique aspect of our scheduling is that the tiles assigned to a core are scheduled by considering the scheduling of the tiles assigned to other cores as well.

[3]Iteration space tiling [65], is a well-known loop restructuring technique that is typically employed to exploit data reuse at the cache level or coarse-grain computation parallelism. In this work, we focus on the latter goal. In tiling, a given iteration space is divided into chunks (where each chunk holds a set of loop iterations) and the iterations of each chunk are executed as a batch.

[4]We want to emphasize that this tiling strategy generates better results than the tiling strategy supported in compilers such as [28] and [1]. We believe this is due to the following two reasons: (i) the approach in [29] explores a larger set of tile shapes compared to [28] and [1], and (ii) our experimentation based tile size selection strategy generates better tile sizes than [28] and [1].

[5]When no confusion occurs, we will also use $T_{c,j}$ to indicate the tile that core c executes (at runtime) in step j.

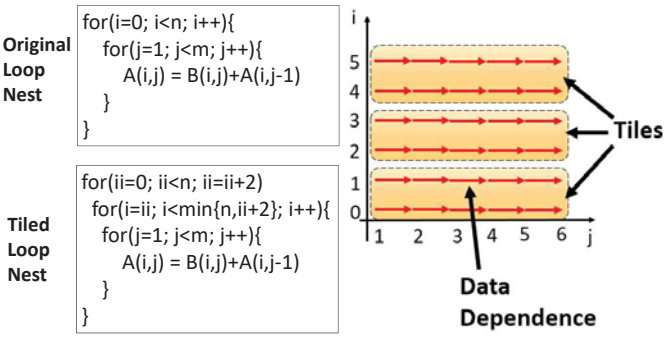


Fig. 2: An example loop nest, its tiled version (using rectangular tile shapes, in this case), and a pictorial representation of the tiles. In general, multiple tiles can be assigned to a core. Also, while in this case tiles are independent, in general one may also have inter-tile dependences that need to be enforced during scheduling.

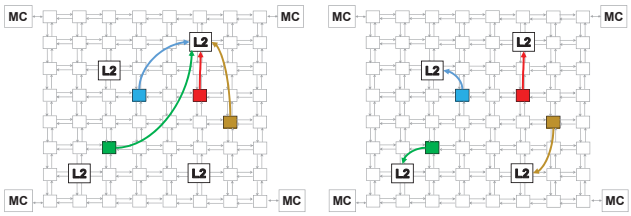


Fig. 3: (a) Original case with a CLP of 1. (b) Optimized case with a CLP of 4.

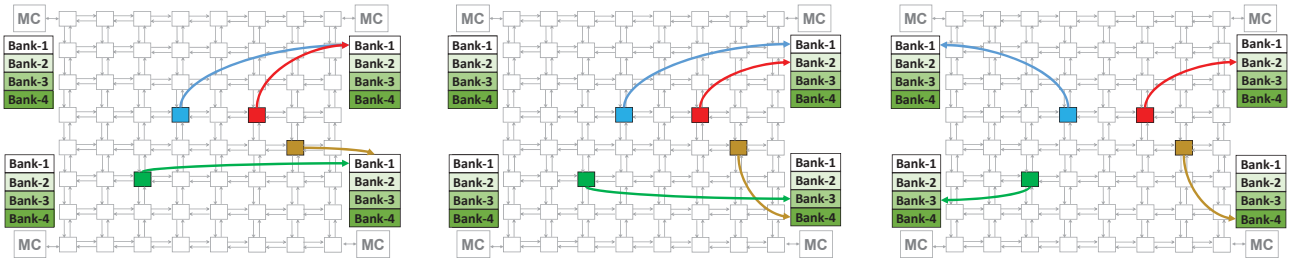
Note that, a program can exhibit high degrees of parallelism and processor utilization; however, if it does not have high values for different DAP components, its data request will experience significant delays on caches/MCs/network, eventually degrading the overall performance.

IV. Components of DAP

Below, we discuss in detail the four components/metrics of DAP, namely, CLP, BLP, MLP and NLP.

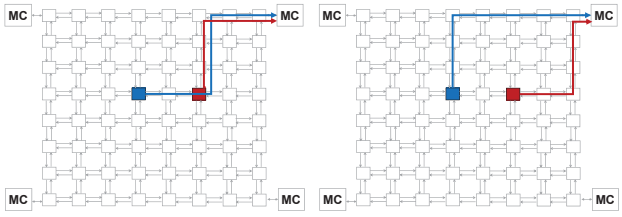
A. CLP

As stated earlier, physical addresses are distributed across the available LLC banks. While one can define CLP in various ways, the definition adopted in this work is based on cache access concurrency. More specifically, we define CLP as the number of LLC banks serving an L1 miss in an epoch of x cycles (where x can be calculated based on processor's ROB size). Clearly, a higher CLP value means better concurrent utilization of the LLC banks in the manycore system. While an application-conscious distribution of addresses to LLC banks can also improve CLP, in this work, we try to improve CLP via computation (tile) scheduling. In other words, our strategy is to schedule tiles in different cores such that CLP is maximized.



(a) (b) (c)

Fig. 4: BLP vs MLP comparison. (a) Original case with a BLP of 2 and an MLP of 2. (b) BLP-optimized case. MLP is still 2 but BLP is now 4. (c) MLP and BLP are optimized together (BLP = 4 and MLP = 4).



(a) (b)

Fig. 5: (a) Original case with an NLP of 7. (b) Optimized case with an NLP of 12.

Figure 3 illustrates, using an example, how CLP can be improved. In the original case shown in (a), we have a CLP of 1, whereas in the optimized case shown in (b), the CLP is 4. More specifically, in the original case, all four concurrent data accesses (L1 misses) originating from the shaded nodes access the same L2 bank, whereas in the optimized case, the same four requests go to different L2 banks.

To achieve CLP-based scheduling, we represent each tile $T_{c,j}$ using a *cache vector* $cv_{c,j} = < a_1, a_2, \cdots, a_m >$, where each bit a_k of which indicates whether $T_{c,j}$ accesses the kth LLC bank in the system. More specifically, bit a_k of $cv_{c,j}$ is set to 1 if any iteration in $T_{c,j}$ accesses the kth LLC bank; otherwise, it is set to 0. Our goal then is to maximize the value of the following expression at each and every scheduling step j:

$$cv_{1,j} \vee cv_{2,j} \vee \cdots \vee cv_{n,j},$$

where $\vee$ denotes bitwise OR operation and n is the total number of cores. In the ideal case, the result of this expression (which can be termed as the *cumulative cache vector* at step j) is $< 1, 1, 1, \cdots, 1, 1 >$, that is, it contains all 1s, indicating that all caches (LLC banks) in the system are accessed at scheduling step j (when accesses from all cores are considered). While the CLP component of data access parallelism is important, a high CLP does *not* guarantee high BLP, NLP or MLP. For example, an execution with a lot of concurrent L2 accesses to different L2 banks (high CLP) can generate L2 misses that mostly go to a small set of memory banks (low BLP).

B. BLP

We define bank-level parallelism (BLP) as the number of banks serving the last-level cache misses in a small epoch (e.g., 128 or 256 cycles). Clearly, a high BLP value means better (more balanced) utilization of available memory banks in the system, and can be expected to lead to higher application performance (compared to a lower BLP value). While there exist a number of prior works that targeted BLP, almost uniformly such works focused on BLP in isolation (Section X discusses them), without looking at it in a larger context, along with other components of data access parallelism. Similar to our definition of cache vector (cv), we define a *bank vector* $bv_{c,j}$ which is an s-bit vector, where s being the total number of banks in the system. The kth bit of this vector is set to 1 if $T_{c,j}$ accesses the kth bank in the system; otherwise, it is set to zero. Consequently, from a BLP viewpoint, one may want to maximize the following expression at each scheduling step j:

$$bv_{1,j} \vee bv_{2,j} \vee \cdots \vee bv_{n,j}.$$

C. MLP

Memory controller-level parallelism (MLP) is defined as the number of MCs serving the last-level cache misses in a small epoch. Since one can have multiple outstanding requests to the memory at a time which can potentially use different channels, a low MLP value may cause some of the controllers to be overwhelmed (and can also congest the NoC links around them), which can in turn degrade the overall application performance. From an optimization viewpoint, we want to maximize the value of the following expression at each scheduling step j:

$$mv_{1,j} \vee mv_{2,j} \vee \cdots \vee mv_{n,j},$$

where $mv_{c,j}$, *memory controller vector*, is an r-bit vector (r is the number of memory controllers), where kth bit is set to one if $T_{c,j}$ accesses the kth memory controller in the system; otherwise, it is set to zero.

Note that improving one of BLP or MLP may not necessarily guarantee an improvement for the other. Let us consider the three different cases depicted in Figure 4. (a) represents the original case with a BLP of 2 and an MLP of 2, i.e., only two memory controllers and two banks are accessed. In (b),

only BLP is optimized – BLP is now 4, whereas MLP is still 2. Finally, in (c), both of the metrics are optimized – BLP is 4 and MLP is 4.

D. NLP

As our last DAP metric, network-level parallelism (NLP) captures the number of NoC links that are simultaneously active in a given short period of time. Clearly, a higher value of NLP indicates a better use of NoC resources. We use $nv_{c,j}$ to denote the an l-bit vector, called *network vector*, where the kth bit ($1 \leq k \leq l$) is set to 1 if $T_{c,j}$ accesses the kth NoC link; otherwise, it is set to 0. As a result, the NLP maximization can be expressed as the problem of maximizing the value of the following expression at each scheduling step j:

$$nv_{1,j} \vee nv_{2,j} \vee \cdots \vee nv_{n,j},$$

In principle, NLP can be divided into two sub-components: NLP due to LLC hits and NLP due to LLC misses. However, in general one may not want to balance LLC hits and misses (as we want the latter to be as low as possible), and consequently, this division of NLP into two sub-components may not be very important. That is, in practice, we do not care about this division, as long as the overall NLP value is high. Figure 5 illustrates the impact of NLP optimization. In (a), which represents the original case, we have an NLP of 7, that is, only 7 links are used, whereas in (b) NLP is optimized to 12.

E. Discussion

While computation (tile) scheduling can be used for improving CLP, BLP and MLP, we need a different mechanism to improve NLP (though computation mapping has an impact on it as well). As stated earlier, by default, going from a source node (e.g., core/L1 cache) to a destination node (e.g., L2 bank) in our NoC is achieved using the XY-routing. However, between the same source-destination pair in our NoC, there can be *multiple routes*, even if we restrict our search space to the ones with the "minimum number of links" (as in the XY-routing). More specifically, consider our 2D mesh-based NoC where a message, say m, is to be sent from a source node, (xs, ys), to a destination node, (xd, yd). If $m = |xd - xs|$ and $n = |yd - ys|$, one can see that this message has C^m_{m+n} *unique* shortest paths. Thus, one can select, for each message, a path such that the number of links used by all messages is maximized, but none of the individual messages uses more links than the XY-routing would use. In this work, we adopted the strategy used in [42], which is a *deadlock-free* implementation.[6]

Also, while data access parallelism is important, it may not be the only factor that affects performance. Clearly, for an application that has not been parallelized well, the role the data access parallelism can play is limited. Further, "data access locality" can also be very important for some applications. Like data access parallelism, data access locality can also be defined in terms of its individual components. For example, everything else being equal, one may want to reduce the distance (in terms of NoC links) between a requesting core and the target L2 bank, so that average NoC latency per data access could be reduced. Where appropriate, we also report locality numbers to show how aggressively optimizing for DAP can affect the data access locality and overall application behavior.

[6]Note however that the goals of the two works are very different, as [42] tries to maximize link reuse to save NoC energy, while we are interested in maximizing the number of NoC links used.

TABLE I: System configuration.

Manycore Size, Frequency	64 (8×8), 1 GHz
L1 Cache	16 KB; 8-way; 32 bytes/line
L2 Cache	512 KB/core; 16-way; 64 bytes/line
Coherence Protocol	MOESI
Router Overhead	3 cycles
Page Size	2 KB
On-Chip Network Frequency	1 GHz
Routing Strategy	XY-routing
DRAM	DDR3-1333; 250 request buffer entries; 4 MCs 1 rank/channel; 16 banks/rank
Row-Buffer Size	2 KB
Address Distribution across LLCs	64 bytes
Address Distribution across banks	64 bytes
Epoch Length	256 cycles

Now that we have defined the four main components of DAP, we next evaluate them quantitatively for four different scenarios: original applications, ideal case with optimum DAP, a pure compiler-based heuristic, and a machine learning (ML) based approach.

V. Evaluation Platform and Workloads

To quantify DAP in multi-threaded applications as well as the impact of optimizing it, we used a simulation-based study. We want to emphasize that currently it is *not* possible to collect detailed statistics on different components of DAP (CLP, BLP, NLP and MLP) on an actual system, and this is why we conducted a simulation based study. Another reason is that we also want to measure the impact of maximizing DAP via an ideal scheme which cannot be implemented in real hardware.

All the experiments reported in this paper are performed using the GEM5 [7] simulation environment. GEM5 can model the system-level architecture as well as the processor microarchitecture. Table I gives the main architectural parameters (along with their default values) that define the manycore system simulated in this work. The values of some of these parameters are later modified to conduct a sensitivity study (Section VIII-C). In this work, each application is simulated for 1 billion instructions after the warm-up phase.

We used 20 multi-threaded applications extracted from three benchmark suites (mantevo [26], specomp [4] and splash-2 [9]). The dataset sizes used in these programs range between 751MB and 3.3GB. Their execution times, when running on the configuration given in Table I, vary from 57.2sec to 3.3min. To implement our compiler support, we used the LLVM compiler infrastructure [40]. In this work, we use LLVM as a source-to-source translator which takes a given (original) application program as input, and generates its DAP-optimized version as output. The optimized codes as well as the original ones are then compiled using the node compiler with the highest optimization flag (O3). The execution model

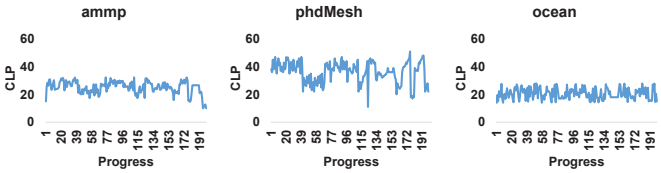


Fig. 6: CLP results over time for three representative applications in their original forms.

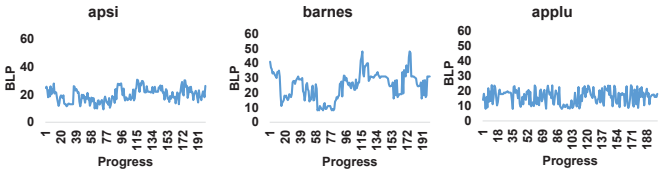


Fig. 7: BLP results over time for three representative applications in their original forms.

used in this work is similar to one that frequently appears in high-performance computing: each application is parallelized across *all* available cores in the system, and we run one (multithreaded) application at a time.

VI. Evaluation of Data Access Parallelism of the Original Applications

Now we quantify CLP, BLP, MLP and NLP of the original applications.When we say "original applications" we mean *no* DAP-specific optimization is employed. However, each application is compiled using *all* data locality and SIMD optimizations supported by the underlying node compiler (with the O3 option). More importantly, all versions compared in this paper (original and optimized) use the *same* (tile based) computation parallelization strategy explained earlier in Section III, and they only differ from each other in how they optimize for DAP. That is, the difference between the original and optimized versions come solely from DAP.

Figure 6 plots the variations in CLP over time for three of our applications, which we believe are "representatives" of the remaining applications as well. Each of these plots captures a small segment of execution (corresponding roughly to about 1/10th of the total application simulation time), and the x-axis indicates time (execution progress). *ammp* represents a case with average CLP; *phdMesh* represents a case with high CLP; and *ocean* represents a case with low CLP. The first bar for each benchmark in Figure 10 gives the average CLP value for the entire simulation period. One can see from these results that, in general, CLP values are *not* very high (considering that the maximum possible value is 64 since we have 64 L2 banks); in fact, in 12 of our 20 applications, the average CLP value is less than 32. These results mean that, our applications, in their original forms, do *not* utilize the available last-level caches (LLCs) well, even under the default cache line granularity distribution of addresses across the caches (although not presented in this paper in detail, using a page granularity distribution of physical addresses across the L2 banks led, on average, to 18% reduction in the CLP values reported in Figure 10). There are three reasons for these

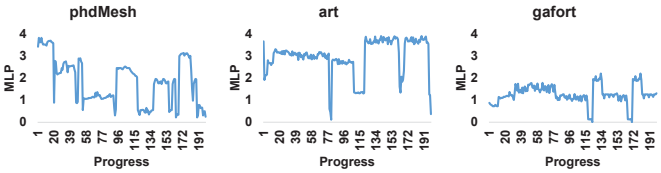


Fig. 8: MLP results over time for three representative applications in their original forms.

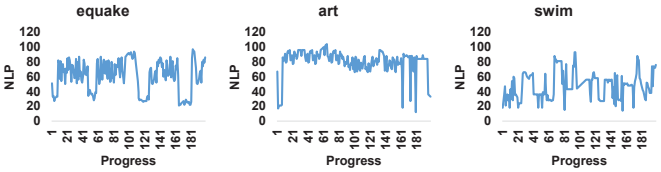


Fig. 9: NLP results over time for three representative applications in their original forms.

less-than-expected CLP values. First, due to temporal locality of data accesses, only a subset of the available L2 caches is actively used at any given execution period. Second, due to the existence of an NoC, it takes some time for a data access to reach its target L2 bank, during which the cache remains idle, if there are no other request being served by the same cache bank. Third, most of these applications have some "computation-intensive" periods as well, where there are not many data accesses, which also contribute to low CLP numbers.

Figure 7 gives the BLP variations of three representative applications (*apsi*, *barnes* and *applu*) over time. As in the case of the CLP values, these BLP values are *not* very high (considering that the maximum possible BLP value is $4 \times 16 = 64$), and one can see from the first bar for each application in Figure 11 that, the average BLP values across 20 applications is around 23.4. The reason for these low BLP values may change from one application to another. In most of the applications with low BLP, the reason is the lack of sufficient LLC misses to fill the available banks; and in the remaining ones, it is the locality of LLC misses, that is, the LLC misses (just like LLC accesses) also exhibit locality and tend to concentrate on a small number of memory banks at a given period of time. Again, adopting a page granularity distribution of physical addresses across the banks (as opposed to the cache line granularity used in our default setting) led to significant reductions in the BLP values plotted in Figure 11 (24% reduction on average).

We next consider the MLP variations for three of our applications (*phdMesh*, *art*, and *gafort*) in Figure 8. Keeping in mind that the maximum possible value for MLP in our default architecture is 4 (Table I), these values are quite low, giving an average of 2.1 (see the first bar for each application in Figure 12), due to the skewing of the last-level cache misses towards 1 or 2 of the memory controllers in a given execution window.

Finally, the results for the three sample NLP traces presented in Figure 9 (for applications *equake*, *art*, and *swim*), and the overall NLP values shown as the first bar for each application

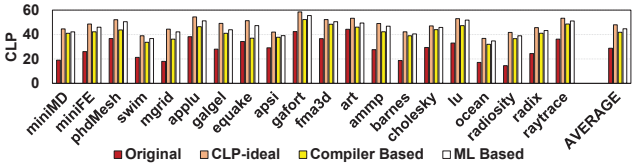


Fig. 10: CLP results under different schemes.

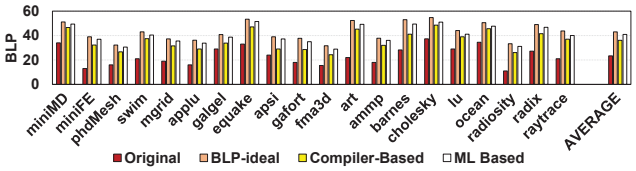


Fig. 11: BLP results under different schemes.

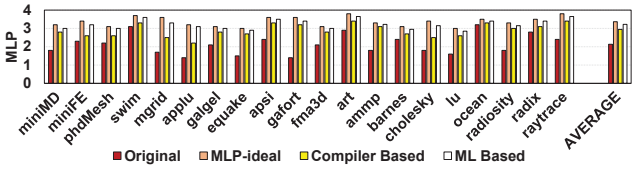


Fig. 12: MLP results under different schemes.

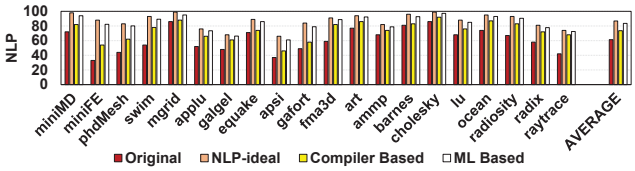


Fig. 13: NLP results under different schemes.

in Figure 13 indicate that the execution does *not* utilize NoC links well (the average NLP value being 61.3). This low utilization has also an important consequence: the concurrent accesses in a period of execution (whether they are cache hits or cache misses destined for a memory controller) compete for a small set of available on-chip communication resources (links and router buffers), which in turn increases contention on the NoC, further reducing the performance.

Summary: It is clear that these multi-threaded applications do *not* take full advantage of the DAP offered by the underlying manycore architecture. We want to emphasize that *these applications have been parallelized very well.* That is, their "computation parallelism" is quite high (and in fact only 5% of the loops in these codes could not be parallelized due to the existence of data dependencies). It is the lack of the sufficient data access parallelism (DAP) that prevents these applications from reaching their maximum potential (as can be observed from low CLP, BLP, MLP and NLP). The next section tries to answer the question of what potential benefits one would obtain if one could somehow maximize the different components of DAP in isolation as well as in combination, without hurting computation parallelism in any way.

VII. Results from the Ideal Data Access Parallelism

This section evaluates five different versions of each application: **CLP-ideal**, **BLP-ideal**, **MLP-ideal**, **NLP-ideal**, and **ALL-ideal**. It is important to emphasize that none of these versions is implementable in practice (but they all can be simulated using GEM5) and they represent in a sense what could be achieved when a certain component of DAP could be fully optimized, without negatively impacting any other aspect of the execution. CLP-ideal corresponds to an execution where the CLP is maximized to the extent allowed by data accesses. For example, if there are, say, 50 cache accesses in flight concurrently, CLP-ideal assumes that they are destined to 50 *different* LLC banks in the system. That is, it maximizes the number of bits in the resulting cumulative cache vector (after each scheduling step, as discussed in Section IV-A). CLP-ideal further assumes that, to the extent possible, each data access goes to the nearest (available) LLC bank, in an attempt to reduce "distance-to-data". BLP-ideal and MLP-ideal make similar assumptions (as CLP-ideal) for the bank and memory controller accesses, respectively. It is important to note that, even under, say, CLP-ideal, one may not be able to reach the ideal value of 64, since there may not be 64 LLC bank accesses at all times. In other words, in a given period of execution, CLP-ideal maximizes the number of concurrent accesses to different LLC banks under a given number of concurrent LLC accesses (in that period). In comparison, NLP-ideal maximizes the number of links exercised at any given time, under the constraint that each data access travels over the minimum number of links to reach its destination (as in the XY-routing). Finally, ALL-ideal combines CLP-ideal, BLP-ideal, MLP-ideal and NLP-ideal.

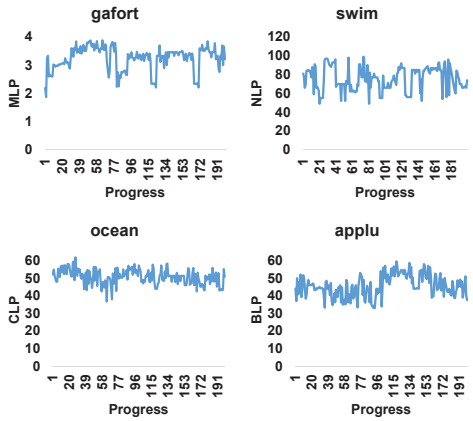


Fig. 14: Variations in four different components of DAP in four originally under-performing applications, when using the ideal scheduling scenario.

Figure 14 gives the CLP, BLP, MLP and NLP behavior of four applications that performed worst in Figures 6, 7, 8 and 9 (*ocean*, *applu*, *gafort*, and *swim*, respectively). It can be clearly seen that these applications, whose original versions performed very poorly, now perform very well from the perspective of these four DAP metrics. The second bars for each application in graphs in Figures 10, 11, 12 and 13 give the average CLP, BLP, MLP and NLP values under CLP-ideal, BLP-ideal, MLP-

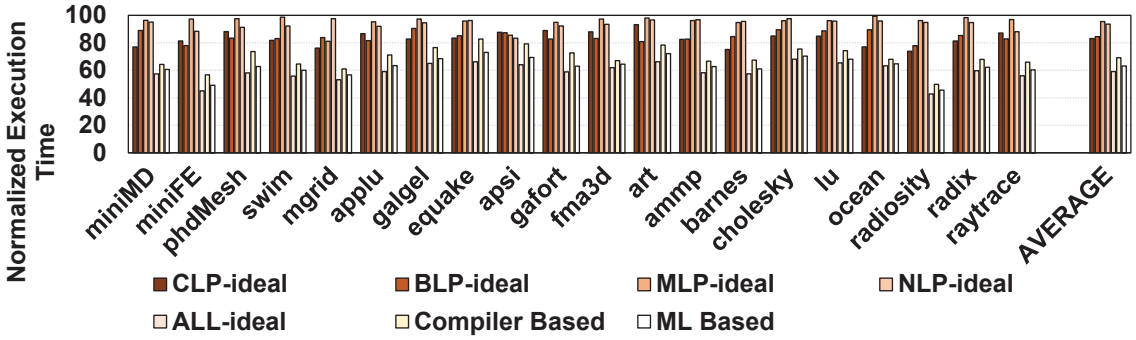


Fig. 15: Normalized execution times with different schemes.

ideal and NLP-ideal, respectively, for all 20 applications we have, for the entire simulation duration. Comparing these plots with those given in the first bar for each application indicate significant improvements due to ideal DAP. In fact, CLP-ideal, BLP-ideal, MLP-ideal and NLP-ideal achieve, on average, CLP, BLP, MLP and NLP values of 47.9, 43.0, 3.4 and 86.8, respectively, indicating improvements of 66.7%, 81.1%, 57.6% and 41.6%, in that order, over the original applications.

The first five bars in Figure 15 plot the execution times under CLP-ideal, BLP-ideal, MLP-ideal, NLP-ideal and ALL-ideal, *normalized* with respect to the original applications. When averaged across all 20 applications we have, CLP-ideal, BLP-ideal, MLP-ideal, NLP-ideal and ALL-ideal bring execution time improvements of 17%, 15.5%, 4.5%, 6.3% and 40.9%, respectively. As stated earlier, *none* of these five ideal schemes is implementable, and one needs a practical scheme if we are to take advantage of DAP. Furthermore, any practical scheme that tries to improve CLP, BLP, MLP or NLP is also likely to have an impact on cache behavior (e.g., cache hit/miss statistics), on-chip network behavior, and off-chip memory behavior (e.g., row-buffer hits/misses), which also need to be quantified. In the rest of this paper, we propose and experimentally evaluate two different approaches to optimize DAP in practice. One of these approaches is pure compiler-based, whereas the other relies on the compiler as well as a machine learning based model.

VIII. Practical DAP Optimization

In this section, we discuss two practical approaches for optimizing DAP. Note that none of these approaches affects the computation parallelism in the application code being optimized – they only change DAP.

A. Compiler-Based Approach

Our compiler-based scheme is based upon the observation that, considering the results captured by the first four bars in Figure 15, our four metrics can be ordered – from the most effective one to the least effective one – as CLP, BLP, NLP and MLP. Therefore, our approach tries to optimize them in that order. More specifically, the compiler first attempts to optimize CLP, and BLP is optimized only if doing so does not hurt CLP. Similarly, NLP is optimized only if doing so does not hurt CLP or BLP, and finally, MLP is optimized only if doing so does not adversely affect the other three metrics. As will be explained below, this scheme works by exploiting the multiple options we have (in general) for maximizing any of the metrics. For example, when two different ways of assigning computations (tile) to cores generate exactly the same CLP, we may prefer the one that generates a better BLP. And similarly, if two alternate tile scheduling generate exactly the same CLP an BLP values, our preference is for the one that results in a higher NLP value than the other. And, finally, if two alternate tile scheduling have exactly the same CLP, BLP and NLP values, our compiler picks the one with the better MLP value.

Algorithm 1 gives the high level description of our DAP-oriented computation (tile) scheduling algorithm. As stated earlier, *the algorithm ranks its priorities as CLP, BLP, NLP and MLP*, based on the results plotted in Figure 15. Specifically, each iteration of the outermost loop (line 1) schedules a group of tiles on different cores with respect to our optimization priorities. The inner loop (lines 5 to 25) iterates on each core to find an appropriate tile for it to schedule next. Since the priority order is fixed (i.e, CLP, BLP, NLP, and MLP), we first select tile(s) which give the maximum CLP (max_clp) value (lines 6 - 8). Second, we choose tile(s) which give the maximum BLP (max_blp) value (lines 9 - 11). Note that, the candidate tiles being used to look for the maximum BLP value are the tiles which already give the maximum CLP value in our first step (lines 6 - 8). Similarly, the candidate tiles for searching maximum NLP (max_nlp) value are the tiles already give the maximum values for both CLP and BLP. Finally, we select one tile which gives us maximum MLP (max_mlp) value (as this tile already satisfies maximum CLP, BLP, and NLP). It is important to emphasize that, while NLP is primarily determined by the route taken by the messages, computation mapping still plays a role in shaping it. This is because two alternate scheduling can lead to different constraints on the on-chip network depending on the data accesses issued by the tiles in them. In identifying the best tile to schedule for a core c at a scheduling step j, our compiler also considers all C^m_{m+n} paths (see Section IV-E) for each potential/schedulable tile, and determines the one that maximizes the value of $(nv_{1,j} \vee nv_{2,j} \vee \cdots \vee nv_{n,j})$. These details are omitted from Algorithm 1 for clarity.

Since our compiler approach needs to distinguish between cache hits and misses, it also employs a cache miss prediction

Algorithm 1 DAP-oriented tile scheduling.

INPUT: number of cores (n); number of tiles per core (m); window size (w);
while $there\ are\ tiles\ waiting\ to\ schedule$ **do**
  $\overrightarrow{clp},\ \overrightarrow{blp},\ \overrightarrow{blp},\ \overrightarrow{mlp} \leftarrow \vec{0}$
  $max\_clp,\ max\_blp,\ max\_nlp,\ max\_mlp \leftarrow \varnothing$
  $schedule \leftarrow \varnothing$
  for $core_i\ from\ core_1\ to\ core_n$ **do**
    for $tile_{i,j}\ from\ tile_{i,1}\ to\ tile_{i,w}$ **do**
      $max\_clp \leftarrow max\_clp \cup\ tile_{i,j}\ max\{\overrightarrow{clp} \bigvee tile_{i,j} \rightarrow cv_{i,j}\}$
    end for
    for $tile_{i,j}\ in\ max\_clp$ **do**
      $max\_blp \leftarrow max\_blp \cup\ tile_{i,j}\ max\{\overrightarrow{blp} \bigvee tile_{i,j} \rightarrow bv_{i,j}\}$
    end for
    for $tile_{i,j}\ in\ max\_blp$ **do**
      $max\_nlp \leftarrow max\_nlp \cup\ tile_{i,j}\ max\{\overrightarrow{nlp} \bigvee tile_{i,j} \rightarrow nv_{i,j}\}$
    end for
    for $tile_{i,j}\ in\ max\_nlp$ **do**
      $max\_mlp \leftarrow max\_mlp \cup\ tile_{i,j}\ max\{\overrightarrow{mlp} \bigvee tile_{i,j} \rightarrow mv_{i,j}\}$
    end for
    $choose\ tile_{i,j}\ from\ max\_mlp$
    $delete\ tile_{i,j}\ from\ core_i$
    $schedule \leftarrow schedule \cup tile_{i,j}$
    $\overrightarrow{clp} \leftarrow \overrightarrow{clp} \cup tile_{i,j} \rightarrow cv_{i,j}$
    $\overrightarrow{blp} \leftarrow \overrightarrow{blp} \cup tile_{i,j} \rightarrow bv_{i,j}$
    $\overrightarrow{nlp} \leftarrow \overrightarrow{nlp} \cup tile_{i,j} \rightarrow nv_{i,j}$
    $\overrightarrow{mlp} \leftarrow \overrightarrow{mlp} \cup tile_{i,j} \rightarrow mv_{i,j}$
  end for
  move slide window
end while

scheme [23] to predict where the LLC miss will happen in each tile, and identify the memory controllers and banks that will be accessed in each tile.

Our approach also needs some help from the operating system (OS). Most OS support APIs that allocate a physical page for a given virtual address using a page-coloring algorithm. There is also an API called page-createva(.) in Solaris (and similar calls in other OS such as Linux) that can accept hints from the applications. For the purposes of this work, we modified this OS call to allocate physical addresses such that the OS uses the same cache bits, rank bits and bank bits from the virtual address for the physical address. Consequently, cache/rank/bank bits are *not* modified during the virtual-to-physical address translation, to allow the compiler reason about DAP metrics. In our experiments, we observed *no* extra page faults due to this constraint in address mapping.

1) Results: The third bar for each application in Figures 10, 11, 12 and 13 give the average CLP, BLP, MLP and NLP values under this compiler scheme. We see that, when averaged across the 20 applications we have, the compiler scheme achieves average CLP, BLP, MLP and NLP values of 41.8, 36.1, 2.9 and 73.6, respectively. When compared to the average values achieved by CLP-ideal (47.9), BLP-ideal (43.0), MLP-ideal (3.4) and NLP-ideal (86.8), respectively, these DAP improvements brought by the compiler can be considered very good (especially, considering that the compiler is limited in its ability to analyze the code and extract data access patterns).

To get the full picture of the impact of our compiler-based strategy however, one needs to consider its effects on *other aspects of execution* as well. In other words, improving DAP can have other types of impacts on cache, NoC and off-chip memory behaviors, which also need to be quantified, for a fair evaluation. For this purpose, we evaluate the impact of our compiler scheme on three metrics: "cache hit rate", "NoC latency", "row-buffer hit rate",[7] to have an idea about its impact of cache performance, NoC performance and main memory performance, respectively. The first bar for each application in Figure 16 gives the percentage variation in the L2 cache rates of the original applications when using the compiler-based scheme. We see that this variation is very small, that is, the compiler optimization does not affect the cache hit rates much (reducing them by only 0.54% on an average). This is mainly because the granularity we use for scheduling is a *tile*, and once a *tile* is scheduled (in both the original applications and optimized ones), all its iterations are executed one after another. Consequently, as far as a tile is concerned, one can expect similar cache performances in both the cases. Variations between the original codes and optimized codes occur across the boundaries of the tiles (as the two versions schedule tiles differently), but the impact of these tile transitions is limited, and consequently, the cache performances of both the schemes are quite similar. A similar observation can be made for the row-buffer hit rates as well. The second bar for each application in Figure 16 plots the percentage variation in row-buffer hit rates, where the average difference between the original applications and DAP-optimized ones is only about 1%. Finally, the last bar gives the variations in NoC latency, when the compiler-optimized codes are used. We see that, our compiler-based DAP optimization increases NoC latency by about 3.1%, on average. These increases in the NoC latencies are not very high. Overall, the impact of our compiler approach on metrics not related to parallelism is quite limited.

Taking into account its impact on DAP (Figures 10, 11, 12, 13) and other metrics of interest (Figure 16), the sixth bar for each application program in Figure 15 gives the "normalized execution time" achieved by the compiler scheme (with respect to the original execution), as a result of improving DAP. We can see an average execution time improvement of 30.8%, which compares quite well to the 40.9% average improvement brought by ALL-ideal. This result clearly shows that *the compiler can be quite successful in improving DAP and reducing execution times* for these multithreaded applications.

Note that, the compiler algorithm explained above tries to exploit CLP, BLP, NLP and MLP in that order. We also performed experiments with a modified version of our compiler algorithm which targets only one component of DAP at a time. The results, plotted in Figure 17 indicate that, while doing so certainly improves, as expected, the target DAP component (for instance, BLP Only generates the best BLP value and CLP Only generates the best CLP value), none of these single component centric versions generates a better performance than the version that considers all four DAP metrics in ordered.

B. Machine Learning-Based Approach

As explained earlier in Section VII, in general, CLP and BLP dominate data access parallelism exhibited by our multi-

[7]Each bank in the off-chip memory has one buffer (called row-buffer), which provides fast access to the page which is accessed last from that bank. A high row-buffer hit rate usually translates to good performance.

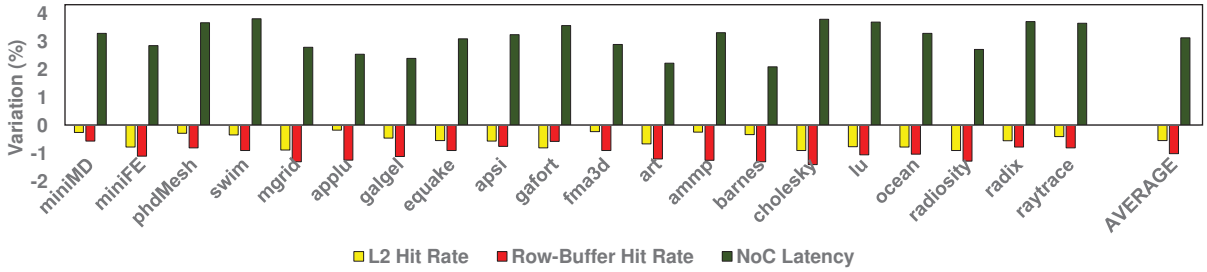


Fig. 16: Percentage variations in L2 hit rates, row-buffer hit rates, and NoC latency, as a result of applying our compiler-based approach.

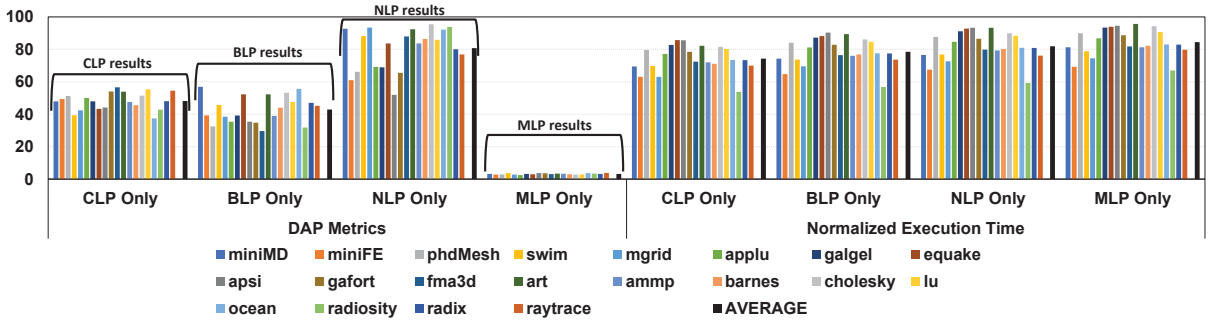


Fig. 17: The values of DAP components and normalized execution times when the compiler targets only one DAP component. Note that, CLP Only refers to the CLP value when the compiler targets only CLP, and BLP Only refers to the BLP value when the compiler targets only BLP (and similarly for the other two metrics). For the DAP components, the y-axis represents the absolute value. For the execution time results, the y-axis represents the normalized time (with respect to the original execution).

threaded benchmarks. However, different loop nests in a given application can have different preferences. Figure 18 gives the breakdown of preferable priority order among CLP, BLP, NLP and MLP across different loop nests in an application. For example, in *phdMesh*, 56.6% of loop nests prefer the optimization order CLP, BLP, NLP and MLP, 22.5% of them prefer BLP, CLP, NLP and MLP, 11% of them prefer CLP, NLP, BLP and MLP, and 7.1% of them prefer BLP, NLP, CLP and MLP. Considering all benchmarks, there is a variety among the preferable orders across different nests.[8]

Motivated by this observation, we propose a **machine learning** (ML) based strategy that selects the best ordering to optimize based on the application and hardware characteristics. While different learning strategies can be used for this purpose, this work employs Support Vector Machines (SVM).[9] In the training phase, we use sample loop nests, to determine the corresponding orderings of DAP components and construct a "predictive model", and in the inference phase, we predict the ordering to use for a loop nest not seen before in that architecture.

[8]Note that the graph in Figure 18 gives the breakdown of the preferences of different loop nests. It does not account for the individual improtance (e.g., iteration counts) of different loop nests.

[9]A Support Vector Machine (SVM) is a classifier that can be defined by a separating hyperplane. More specifically, given labeled training data (supervised learning), SVM outputs an optimal hyperplane which categorizes new examples.

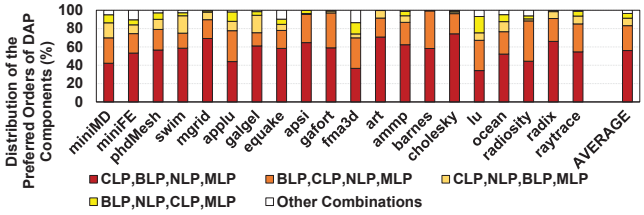


Fig. 18: Breakdown of preferable priority orders among CLP, BLP, NLP and MLP across different loop nests in our applications. Note that the last component (Other Combinations) represents the cumulative contribution of all remaining twenty orders.

TABLE II: Features used in our machine learning-based approach. Note that the static features are the ones extracted from the program code by the compiler (at compile-time), whereas the dynamic features are the ones extracted by the profiler (at run-time).

Static Features	Static Instruction Count, Number of Program Statements, Number of Data References, Branch Count, Number of Arrays Accesses
Dynamic Features	Dynamic Instruction Count, Dynamic Data Access Count, Cache Miss Statistics, Number of Concurrent Main Memory Accesses, Average Distance (on NoC) per Data Request

1) Training: In this part, we employ an offline supervised training module to which code (loop nest) features of interest are presented. The static (compiler-extracted) and dynamic (profile-extracted) code features used in this work are listed in Table II. This module executes the input code (loop nest) in the

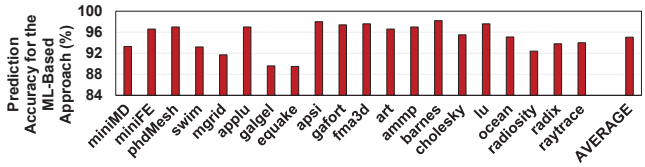


Fig. 19: Prediction accuracy of machine learning-based approach.

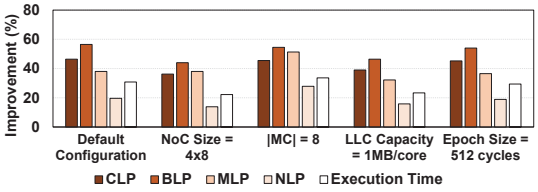


Fig. 20: Sensitivity results with the compiler-based scheme.

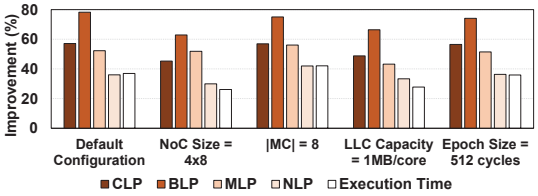


Fig. 21: Sensitivity results with the ML-based scheme.

target architecture, targeting all 24 possible values of vector (x_1, x_2, x_3, x_4), where each x_i can be one of CLP, BLP, NLP and MLP. For example, when testing for (BLP, CLP, NLP, MLP), the compiler first tries to optimize for BLP, and in cases where there are multiple optimized code candidates (tile scheduling) with exactly the same BLP values, it favors the one with higher CLP; and in cases where there are options with the same CLP and BLP values, it favors the one with the higher NLP value, and so on. At the end of the training of the loop nest, we identify the (x_1, x_2, x_3, x_4) vector that generates the "best performance" for that nest. Using such profile data from representative loop nests, we build an SVM model to be used later in the inference phase.

In our work, all the loop nests that we used as "representatives" in training are from outside the benchmarks used in our evaluations (in the inference phase). Specifically, we used more than 350 representative loop nests from the specfp2006 [2], specomp[4] and parsec [6] suites as well as 16 popular linear algebra kernels. Also, whenever possible, we used all the different datasets available to us for a given benchmark (as many as 8 for some benchmarks) to better train our model.

2) Inference: In this phase, given a loop nest (*not* seen before) and a target architecture, our approach first extracts features listed in Table II. These features are then given to our SVM module, which returns the vector (x_1, x_2, x_3, x_4) that contains the preferable ordering (from left to right in x_i positions) among CLP, BLP, NLP and MLP. Following this, the compiler applies the corresponding code transformation and accompanying message routing policy for the loop nest and generates code. This process is repeated for each loop nest in the application code being optimized. Clearly, while the pure compiler-based approach discussed earlier in Section VIII-A uses the same order of DAP components for *all* the loop nests in an application (namely, CLP, BLP, NLP, MLP), this ML-based approach *can* choose different orders for different loop nests, depending on the characteristics of those nests.

3) Results: We start by presenting the "accuracy" of our SVM based prediction, that is, what is the fraction of loop nests, for which our learning based approach predicted the correct (ideal) order of DAP components? The results, plotted in Figure 19, indicate that our approach achieves an average prediction accuracy of 95.5%.

The fourth bar for each application in Figures 10, 11, 12 and 13 give the average CLP, BLP, MLP and NLP values under the machine learning-based approach. It can be observed from these results that, this approach achieves average CLP, BLP, MLP and NLP values of 44.8, 40.9, 3.2 and 83.5, respectively. These are clearly better than the corresponding values obtained when using the pure compiler approach (41.8, 36.1, 2.9 and 73.6 in that order), and are quite close to the performance of CLP-ideal (47.9), BLP-ideal (43.0), MLP-ideal (3.4) and NLP-ideal (86.8), respectively.

The variations we observed with this machine learning based approach on L2 hit rate, row-buffer hit rate and average NoC latency were similar to those observed with the compiler based approach (Figure 16); so, we do not show them in the interest of space. The last bar for each application program in Figure 15 gives the "normalized execution time" achieved by this ML-based scheme (with respect to the original execution), as a result of improving DAP. We observe an average performance improvement of 36.9%, which is much better than the average improvement brought by the pure compiler approach (30.8%), and is also very close to ALL-ideal (40.9%). That is, instead of using the same order (of DAP components) for all loop nests in a program, customizing the order based on the characteristics (features) of the individual loop nests in the program can bring significant additional benefits.

C. Sensitivity Experiments

Due to space concerns, we only give average results (geometric means) across our applications. It is important to mention that, in each of the experiments presented below, *we change the value of only one parameter at a time*; all the other parameters maintain their default values listed in Table I. Also, to minimize the number of plots, the results are presented as "percentage improvements" over the default (original) execution. The first parameter we varied is the mesh size (number of cores). Recall that the NoC size we used so far in our experiments is 8×8, which is the largest size we could simulate in our simulator. We also performed a set of experiments with a smaller configuration (4×8), and the resulting CLP, BLP, NLP, MLP, and execution time improvements for the compiler-based approach are presented in Figure 20 as the second group of bars (the first group of bars give the improvements with the default parameters, for ease of comparison). In general, the benefits coming from the compiler

based scheme are higher with the larger configuration (8×8). This is mainly because in a large configuration the compiler has more room in redirecting accesses to parallel resources. We next change the number of MCs from the original value of 4 to 8. While both the original applications and their compiler-optimized counterparts benefit from the increased MC count, the improvements brought by our approach over the original increase with the increased number of MCs (see the third group of bars in Figure 20). This is due to the fact that a larger number of MCs does not only give the compiler scheme a better chance to improve MLP, it also better spreads the off-chip memory accesses (LLC misses) over the NoC space, thereby improving NLP as well. The third parameter whose impact we quantify is the LLC capacity. Recall that our default configuration has a 512 KB L2 cache per core. The results, shown as the fourth group of bars in Figure 20, indicate that using a larger LLC capacity per core (1 MB) reduces the savings brought by the compiler based scheme in both CLP and BLP. The last parameter we evaluate is the epoch length (whose default value was 256 cycles). As shown in the last group of bars in Figure 20, the effectiveness of the compiler-based approach is not very sensitive to the epoch size. To summarize, these sensitivity experiments demonstrate that the compiler-based scheme performs reasonably well under the different values of our major simulation parameters. Figure 21 plots the results from the same set of sensitivity experiments when our ML-based approach is used. The trends observed in this graph are similar to those observed in Figure 20.

We also ran experiments with multiple applications executing concurrently on the same manycore system. Our experiments with various 4-application workloads running under the default values of our major simulation parameters produced an average 22% improvement in the weighted speedup metric [21]. Further, the average improvements jumped to 27% when running 8-application workloads, indicating that our approaches can be very effective when hosting a multi-programmed workload of multithreaded applications.

IX. Related Work

In this section, we go over the related works in five categories: application mapping, memory/bank level parallelism, NoC optimization, cache-oriented optimization, and resource utilization.

Application mapping: Muralidhara et al. approached the application mapping from the angle of optimizing the performance of caches [49]. Kim et al. proposed memory scheduling methods to improve system throughput and fairness [33], [34]. Das et al. proposed a set of mapping policies that improve performance via reducing the inter-application interference [18]. To our knowledge, none of these works deals specifically with DAP.

Memory/bank level parallelism: Lee et al. focused on improving BLP when prefetching is used [41]. Mutlu et al. developed a parallelism-aware batch scheduler that reduces memory overheads [50]. Ding et al. proposed a tile scheduling scheme that predicts the LLC misses [19]. Prior works also include [17], [53], [60], [24], [37], [10] and [66]. While most of these works focus on memory performance and BLP, we consider all four components of DAP.

NoC optimizations: Hu et al. proposed a message routing technique that minimizes the energy spent on data communication [27]. Monchiero et al. explored distributed shared memory architectures and developed an on-chip hardware memory management unit [47]. Lankes et al. proposed optimizations for skewed access patterns on data IO interfaces [39]. Compared to our work, all these works are NoC centric.

Cache-oriented optimizations: Kowarschik et al. studied various cache optimization techniques focusing on numerical linear algebra [38]. Cho et al. developed a caching policy that can adapt to the environment by dynamically controlling data placement [13]. Chaudhuri proposed a data migration mechanism that reduces the overhead of cache management [11]. The main focus of these studies is LLC, whereas we focus on LLCs, NoC, MCs, and memory banks.

Resource utilization: Finally, there are prior works that target quantifying and/or optimizing resource utilization in various domains [25], [8], [61], [14], [51], [12], [45], [58], [22], [46], [3], [48], [44], [57], [64], [31]. However, most of these works deal with exclusively either CPU/GPU core utilization, or off-chip network utilization, or off-chip memory/storage utilization. In contrast, we focus on emerging manycores and consider different aspects of data access parallelism.

X. Concluding Remarks and Future Work

This is the first comprehensive study of data access parallel parallelism (DAP) on manycores considering its four components (CLP, BLP, NLP and MLP). Our evaluation results demonstrate that the original values of these four metrics are far from optimal. On the other hand, our limit study clearly indicates the huge potential one has if these metrics could be improved, in terms of both data access parallelism and execution cycles. Finally, our compiler-driven approach represents one possible strategy to harness some of this potential. In particular, the proposed compiler strategy improves CLP, BLP, NLP and MLP, on average, by 45.5%, 54.6%, 20.0% and 37.9%, respectively, resulting in an average execution time improvement of 30.8% across all the workloads tested. We also developed a machine learning-based approach which improves CLP, BLP, NLP and MLP, on average, by 55.4%, 75.3%, 36.3% and 50.9%, respectively, resulting in an average execution time improvement of 36.9%.

Our future work will focus on integrating the DAP-centric optimizations presented here with those targeting data access locality (e.g., classical cache centric ones as well as those reducing the number of NoC links traversed by data requests). We will also work on quantifying the benefits of our approach on other types of applications, e.g., pointer-intensive ones.

XI. Acknowledgement

This work is supported in part by NSF grants 1526750, 1763681, 1439057, 1439021, 1629129, 1409095, 1626251, 1629915, and a grant from Intel.

REFERENCES

[1] GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[2] SPECfp 2006. https://www.spec.org/cpu2006/CFP2006/.

[3] AL FARUQUE, M. A., KRIST, R., AND HENKEL, J. ADAM: Run-time agent-based distributed application mapping for on-chip communication. In *Proceedings of the 45th Annual Design Automation Conference* (2008), DAC.

[4] ASLOT, V., DOMEIKA, M., EIGENMANN, R., GAERTNER, G., JONES, W. B., AND PARADY, B. *SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance*. 2001.

[5] BARUA, R., LEE, W., AMARASINGHE, S., AND AGARAWAL, A. Compiler support for scalable and efficient memory systems. *IEEE Transactions on Computers 50*, 11 (2001).

[6] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (2008).

[7] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The Gem5 simulator. *ACM SIGARCH Computer Architecture News 39*, 2 (Aug. 2011).

[8] BOYER, M. *Improving Resource Utilization in Heterogeneous CPU-GPU Systems*. PhD thesis, University of Virginia, 2013.

[9] BUELL, D. A., ARNOLD, J. M., AND KLEINFELDER, W. J. *Splash 2: FPGAs in a custom computing machine*, vol. 9. Wiley-IEEE Computer Society Press, 1996.

[10] CASTRO, D., ROMANO, P., DIDONA, D., AND ZWAENEPOEL, W. An analytical model of hardware transactional memory. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2017).

[11] CHAUDHURI, M. PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *IEEE 15th International Symposium on High Performance Computer Architecture* (2009).

[12] CHEN, D., HENIS, E., KAT, R. I., SOTNIKOV, D., CAPPIELLO, C., FERREIRA, A. M., PERNICI, B., VITALI, M., JIANG, T., LIU, J., ET AL. Usage centric green performance indicators. *ACM SIGMETRICS Performance Evaluation Review 39*, 3 (2011), 92–96.

[13] CHO, S., AND JIN, L. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006), MICRO 39.

[14] CHOU, C., JALEEL, A., AND QURESHI, M. BATMAN: Maximizing bandwidth utilization of hybrid memory systems. *Georgia Institute of Technology, Tech. Rep. TR-CARET-2015-01* (2015).

[15] CHOU, Y., FAHS, B., AND ABRAHAM, S. Microarchitecture optimizations for exploiting memory-level parallelism. In *ACM SIGARCH Computer Architecture News* (2004), vol. 32, p. 76.

[16] CHOU, Y., SPRACKLEN, L., AND ABRAHAM, S. G. Store memory-level parallelism optimizations for commercial applications. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (2005), MICRO 38.

[17] CONG, J., JIANG, W., LIU, B., AND ZOU, Y. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst. 16*, 2 (Apr. 2011).

[18] DAS, R., AUSAVARUNGNIRUN, R., MUTLU, O., KUMAR, A., AND AZIMI, M. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *IEEE 19th International Symposium on High Performance Computer Architecture. HPCA* (2013), pp. 107–118.

[19] DING, W., GUTTMAN, D., AND KANDEMIR, M. Compiler support for optimizing memory bank-level parallelism. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), MICRO-47.

[20] DING, W., TANG, X., KANDEMIR, M., ZHANG, Y., AND KULTURSAY, E. Optimizing Off-chip Accesses in Multicores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015).

[21] EYERMAN, S., AND EECKHOUT, L. System-level performance metrics for multiprogram workloads. *IEEE Micro* (2008).

[22] FERRANTE, A., MEDARDONI, S., AND BERTOZZI, D. Network interface sharing techniques for area optimized NoC architectures. In *Digital System Design Architectures, Methods and Tools, 11th EUROMICRO Conference* (2008).

[23] GHOSH, S., MARTONOSI, M., AND MALIK, S. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.* (1999).

[24] GUO, Y., LIU, Q., XIAO, W., HUANG, P., PODHORSZKI, N., KLASKY, S., AND HE, X. Self: A high performance and bandwidth efficient approach to exploiting die-stacked dram as part of memory. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2017).

[25] GUO, Z., FOX, G., ZHOU, M., AND RUAN, Y. Improving resource utilization in MapReduce. In *Proceedings of the IEEE International Conference on Cluster Computing* (2012), CLUSTER '12.

[26] HEROUX, M. A., DOERFLER, D. W., CROZIER, P. S., WILLENBRING, J. M., EDWARDS, H. C., WILLIAMS, A., RAJAN, M., KEITER, E. R., THORNQUIST, H. K., AND NUMRICH, R. W. Improving Performance via Mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratories, 2009.

[27] HU, J., AND MARCULESCU, R. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2005).

[28] INTEL. Intel Compilers. https://software.intel.com/en-us/intel-compilers.

[29] IRIGOIN, F., AND TRIOLET, R. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1988).

[30] KANDEMIR, M., ZHAO, H., TANG, X., AND KARAKOY, M. 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* (2015).

[31] KAYIRAN, O., JOG, A., PATTNAIK, A., AUSAVARUNGNIRUN, R., TANG, X., KANDEMIR, M. T., LOH, G. H., MUTLU, O., AND DAS, C. R. uC-States: Fine-grained GPU Datapath Power Management. In *PACT* (2016).

[32] KIM, C., BURGER, D., AND KECKLER, S. W. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGARCH Comput. Archit. News 30*, 5 (Oct. 2002).

[33] KIM, Y., HAN, D., MUTLU, O., AND HARCHOL-BALTER, M. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16 The Sixteenth International Symposium on High-Performance Computer Architecture* (2010), pp. 1–12.

[34] KIM, Y., PAPAMICHAEL, M., MUTLU, O., AND HARCHOL-BALTER, M. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010), pp. 65–76.

[35] KISLAL, O., KOTRA, J., TANG, X., KANDEMIR, M. T., AND JUNG, M. Enhancing computation-to-core assignment with physical location information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018).

[36] KISLAL, O., KOTRA, J., TANG, X., TAYLAN KANDEMIR, M., AND JUNG, M. POSTER: Location-Aware Computation Mapping for Many-core Processors. In *Proceedings of the 2017 International Conference on Parallel Architectures and Compilation* (2017).

[37] KOTRA, J. B., GUTTMAN, D., N., N. C., KANDEMIR, M. T., AND DAS, C. R. Quantifying the potential benefits of on-chip near-data computing in manycore processors. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2017).

[38] KOWARSCHIK, M., AND WEISS, C. *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*. 2003.

[39] LANKES, A., WILD, T., AND HERKERSDORF, A. Hierarchical NoCs for optimized access to shared memory and io resources. In *Digital System Design, Architectures, Methods and Tools. 12th Euromicro Conference* (2009).

[40] LATTNER, C., AND ADVE, V. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization. CGO.* (2004).

[41] LEE, C. J., NARASIMAN, V., MUTLU, O., AND PATT, Y. N. Improving memory bank-level parallelism in the presence of prefetching. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), MICRO 42.

[42] LI, F., CHEN, G., KANDEMIR, M., AND KOLCU, I. Profile-driven energy reduction in network-on-chips. In *Proceedings of the 28th*

ACM SIGPLAN Conference on Programming Language Design and Implementation (2007), PLDI.

[43] LIN, J., LU, Q., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), SC.

[44] LIU, G., AN, H., HAN, W., LI, X., SUN, T., ZHOU, W., WEI, X., AND TANG, X. FlexBFS: A Parallelism-aware Implementation of Breadth-first Search on GPU. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2012).

[45] MESNIER, M. P., WACHS, M., SAMBASIVAN, R. R., ZHENG, A. X., AND GANGER, G. R. Modeling the relative fitness of storage. 37–48.

[46] MOHANTY, S., AND PRASANNA, V. K. Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures. In *ASIC/SOC Conference. 15th Annual IEEE International* (2002).

[47] MONCHIERO, M., PALERMO, G., SILVANO, C., AND VILLA, O. Exploration of distributed shared memory architectures for NoC-based multiprocessors. *Journal of Systems Architecture 53*, 10 (2007), 719 – 732.

[48] MUDALIGE, G. R., GILES, M. B., REGULY, I., BERTOLLI, C., AND KELLY, P. H. J. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar)* (2012), pp. 1–12.

[49] MURALIDHARA, S. P., KANDEMIR, M., AND KISLAL, O. Reuse distance based performance modeling and workload mapping. In *Proceedings of the 9th Conference on Computing Frontiers* (2012), CF '12.

[50] MUTLU, O., AND MOSCIBRODA, T. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *35th International Symposium on Computer Architecture* (2008), ISCA.

[51] OZTURK, O., KANDEMIR, M., AND IRWIN, M. J. Using data compression for increasing memory system utilization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28*, 6 (2009), 901–914.

[52] PAI, V. S., AND ADVE, S. Code transformations to improve memory parallelism. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture* (1999), pp. 147–155.

[53] PARK, J. J. K., PARK, Y., AND MAHLKE, S. ELF: Maximizing memory-level parallelism for GPUs with coordinated warp and fetch scheduling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), ACM, p. 18.

[54] PATSILARAS, G., CHOUDHARY, N. K., AND TUCK, J. Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era. *ACM Transactions on Architecture and Code Optimization. TACO 8*, 4 (2012), 28.

[55] PATTNAIK, A., TANG, X., JOG, A., KAYIRAN, O., MISHRA, A. K., KANDEMIR, M. T., MUTLU, O., AND DAS, C. R. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *PACT* (2016).

[56] PHADKE, S., AND NARAYANASAMY, S. MLP aware heterogeneous memory system. In *2011 Design, Automation & Test in Europe* (2011), pp. 1–6.

[57] PUTHOOR, S., TANG, X., GROSS, J., AND BECKMANN, B. M. Oversubscribed command queues in gpus. In *Proceedings of the 11th Workshop on General Purpose GPUs* (2018).

[58] QIU, D., AND SHROFF, N. B. A new predictive flow control scheme for efficient network utilization and QoS. 143–153.

[59] SHRIFI, A., DING, W., GUTTMAN, D., ZHAO, H., TANG, X., KANDEMIR, M., AND DAS, C. 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* (2017).

[60] SUNG, I.-J., STRATTON, J. A., AND HWU, W.-M. W. Data layout transformation exploiting memory-level parallelism in structured grid manycore applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (2010), ACM, pp. 513–522.

[61] TAKESHIMA, R., AND TSUMURA, T. Automatic code tuning for improving GPU resource utilization. In *2014 Second International Symposium on Computing and Networking* (2014), pp. 419–425.

[62] TANG, X., KANDEMIR, M., YEDLAPALLI, P., AND KOTRA, J. Improving Bank-Level Parallelism for Irregular Applications. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016).

[63] TANG, X., KISLAL, O., KANDEMIR, M., AND KARAKOY, M. Data movement aware computation partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017).

[64] TANG, X., PATTNAIK, A., JIANG, H., KAYIRAN, O., JOG, A., PAI, S., IBRAHIM, M., KANDEMIR, M., AND DAS, C. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Computer Architecture* (2017).

[65] WOLFE, M. More iteration space tiling. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (1989), pp. 655–664.

[66] ZHOU, H., AND CONTE, T. M. Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Transactions on Computers* (2005).