Network Footprint Reduction through Data Access and Computation Placement in NoC-Based Manycores

Jun Liu, Jagadish Kotra, Wei Ding and Mahmut Kandemir
Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA - 16802, U.S.A
{jxl1036, jbk5155, wzd109, kandemir}@cse.psu.edu

ABSTRACT

Targeting network-on-chip based manycores, we propose a novel compiler framework to optimize the network latencies experienced by off-chip data accesses in reaching the target memory controllers. Our framework consists of two main components: data access placement and computation placement. In the data access placement, we separate the data access nodes from the computation nodes, with the goal of minimizing the number of links that need to be visited by the request messages. In the computation placement, we introduce computation decomposition and select appropriate computation nodes, to reduce the amount of data sent in the response messages and also to minimize the number of communication links visited. We performed an experimental evaluation of our proposed approach, and the results show an average execution time improvement of 21.1%, while reducing the network latency by 67.3%.

Categories and Subject Descriptors

H.4 [**NoC-Based Hardware**]: Data and Computation Placement; D.2.8 [**Computation Placement**]: Metrics—*performance measures, energy measures*

General Terms

Design, Experimentation, Performance, Energy

Keywords

NoC Based manycores, Data and Computation placement

1. INTRODUCTION

It is clear that processor performance is improving at a much faster pace than memory performance. This makes it very important to optimize memory system performance using both hardware and software techniques. Most of the compiler-based data access optimization schemes proposed in the literature exclusively target cache behavior, largely omitting off-chip memory accesses. While optimizing cache behavior is of utmost importance, off-chip data accesses can also play a critical role. This is particularly true for emerging manycore architectures where parallel threads from data-intensive applications can issue a lot of memory requests, creating contention not only on memory controllers, but also on the on-chip network. The cost of an off-chip data access in a manycore consists of two main components: (a) time spent by the access in the network on-chip (NoC) to reach the target memory controller and (b) time spent in accessing the memory itself including the queuing latency and the DRAM access latency. In a large manycore system, the first component can be very important and is the target of this paper. For example, Figure 1 plots the *execution time breakdown* for the multithreaded applications in our benchmark suite [12] into three parts (on a 4×8 multicore). The first, marked as "on-chip", captures the on-chip component of an off-chip memory access (i.e., the time the off-chip access spends in the on-chip network). The second, "off-chip", is the fraction of time spent in accessing the off-chip memory itself, including the queuing latency in the memory controller, and the DRAM access itself. Finally, the last part, called "other", is the fraction of time spent in computation as well as on-chip cache accesses. One can observe that the time spent in the on-chip network is quite significant (averaging on 31.4%).

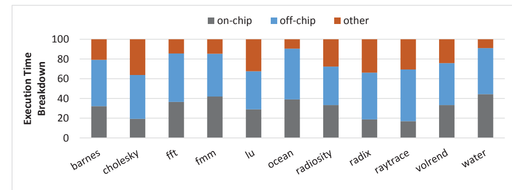

Figure 1: Execution time breakdown of applications running on a 4×8 on-chip network based multicore.

Motivated by this observation, this paper proposes and evaluates a novel "compiler-directed" *data access and computation placement* strategy targeting on-chip network based multicore architectures and programs composed using affine loop nests. Our specific contributions in this work include:

- We propose a data access placement strategy that minimizes the number of links visited by request messages. This strategy assigns a separate "data access node" for each data access to initiate the data request.
- We propose a computation placement strategy that minimizes the number of links visited by response messages. This strategy determines, for each computation, a computation node that has the shortest average distance to the related memory controllers.
- We propose a strategy that reduces the amount of data sent in response messages. The idea behind this is to break a computation into sub-computations (sub-operations), and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.Request permissions from Permissions@acm.org.
DAC '15, June 07-11, 2015, San Fransico, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06 ...$15.00
http://dx.doi.org/10.1145/2744769.2744876.

execute them directly on the data access nodes. Consequently, only the generated intermediate results from the sub-operations will be sent to the computation core, causing less traffic on the on-chip network.

- We present results collected on a 32-core system, indicating that our optimization framework cuts, on average, the network latency of off-chip data accesses by 67.3%, and generates an execution time improvement of 21.1%.

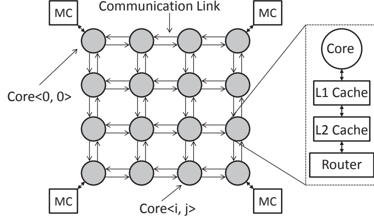

Figure 2: Target NoC-based manycore architecture (MC denotes a memory controller).

Note that our compiler-based work is complementary to related OS [7] and architectural based studies [6].

2. BACKGROUND AND MOTIVATION

2.1 Architecture

As shown in Figure 2, we target a network-on-chip (NoC) based manycore system with $(N \times M)$ nodes. Each node has a core and *private* L1 and L2 caches. To facilitate our discussion, each node in this architecture is labeled by a unique index $\langle i, j \rangle$ (e.g., the index of the upper leftmost core is $\langle 0, 0 \rangle$). Further, some nodes are attached memory controllers (MCs) that manage off-chip memory accesses. Each MC also has an index, which is the same of the index of the node to which the MC is attached. When no confusion occurs, we use the terms "core" and "node" interchangeably. Any missed data request in L1 cache is forwarded to L2 cache. If it is missed in the L2 cache too, an off-chip data request through a memory controller is initiated. Note that, in this architecture, the off-chip data accesses can be very expensive for two reasons: (a) overhead of traversing over the on-chip network and (b) overhead of accessing a memory bank. Our footprint reduction framework aims at optimizing the first overhead, while prior strategies to reduce the second overhead [9] are orthogonal to our work.

Assuming a static XY-routing in the network[1], we start with an important observation: for two cores that communicate with the same MC, the latencies of the messages can be different due to the different number of network hops (distance) that need to be traversed [8]. Specifically, shorter the distance that need to be traversed by a data access, better from both the performance and power perspectives. We define the distance $Dis(\langle i, j \rangle, \langle m, n \rangle)$ from core $\langle i, j \rangle$ to MC $\langle m, n \rangle$ as the Manhattan Distance:

$$Dis(\langle i, j \rangle, \langle m, n \rangle) = |i - m| + |j - n|, \quad (1)$$

which captures the number of hops that need to be traversed from core $\langle i, j \rangle$ to MC $\langle m, n \rangle$. We say that core $\langle i, j \rangle$ is **associated** with MC $\langle m, n \rangle$ if $Dis(\langle i, j \rangle, \langle m, n \rangle)$ is minimum among all distances between core $\langle i, j \rangle$ and any MC. For example, in Figure 2, core $\langle 0, 1 \rangle$ is associated with MC $\langle 0, 0 \rangle$. ~~since this reduces distance-to-~~data.

[1]There are two main reasons why we focus on XY-routing. First, since we want to expose routing to compiler, it needs to be static. Second, our preliminary experiments indicated that dynamic routing in NoC can incur significantly higher energy consumption than static routing.

2.2 Footprints

The execution of a statement in a program usually consists of two steps. The first step is to load the needed data elements from the main memory or on-chip caches into the registers. The second step is to perform the specified computation. In the first step, an off-chip access generates a **network footprint**, i.e., message traffic on the on-chip network. There are two types of footprints. The first type is caused by data *request messages* from the cores to the MCs. The second type is generated by the *data messages* (response messages) carrying the requested data from the MCs. For example, in Figure 3, core $\langle 2, 2 \rangle$ sends a request message to MC $\langle 0, 0 \rangle$ and causes footprint (1). This MC sends back a data message to core $\langle 2, 2 \rangle$ and generates footprint (2). We define the *footprint value* F as follows:

$$F = L \times W, \quad (2)$$

where L is the length of the footprint (the number of traversed links calculated using Eq. (1)), and W is the weight of the footprint (the number of data elements in the message). The network footprints directly affect data access latencies, network congestions and power consumption on the communication links/message buffers. Hence, *it is critical to reduce footprints on the on-chip network as much as possible.*

2.3 Data Access Placement

This section introduces our first optimization, called *Data Access Placement*, to reduce the "request message footprints". Everything else being equal, one would prefer the sender (core $\langle i, j \rangle$) of the request message to be as close as possible to the receiver (MC $\langle m, n \rangle$). In a conventional execution, for each data access, there is only one "data access node" (i.e., the sender of the request message), which is the same as the computation node (i.e., the node that will perform the computation). A data access node may need to send request messages to different MCs, in order to fetch multiple data elements needed by the computation. The problem is that, it is not possible to find a data access node that is close to all MCs. Our solution to this problem is to *separate/decouple* the data access and computation, and place them into different cores. Further, we can have "multiple" data access nodes to issue different request messages on behalf of the same computation node. The advantage of doing so is that we can decide at which node to perform a specific data access, so that the network footprint of a data request message can be reduced independently. The response data messages will be sent from the MCs to the computation node. Hence, the *data access node* actually performs a data access for the *computation node*, not for itself. We introduce a new **load** operation for our NoC-based manycore architecture ("*slw dest, addr*"). Specifically, this new load is executed on the data access node and sends the request message to the MC based on target address *addr*. The response message is then sent from the MC to the destination node *dest*. Figure 3 shows the footprint of request message (1) and data message (2) when not using an explicit data access node (the computation node is $\langle 2, 2 \rangle$) – this represents the conventional data access in the manycore. In comparison, Figure 4 illustrates the footprint when we employ $\langle 0, 1 \rangle$ as the data access node, whereas the computation node is still $\langle 2, 2 \rangle$. The new request footprint (1) is now sent from $\langle 0, 1 \rangle$ to the MC, and the data read will be forwarded to $\langle 2, 2 \rangle$ (using our new load operation). Therefore, compared to Figure 3, Figure 4 reduces the data request footprint F from 4 to

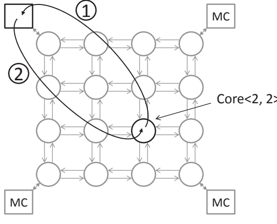

Figure 3: Footprint example without any data access node.

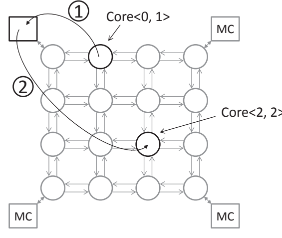

Figure 4: Footprint example with the data access node ($\langle 0, 1\rangle$).

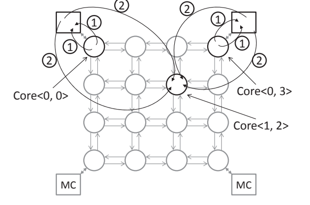

Figure 5: Optimal data access placement for a simple example.

1, assuming that the weight W in Eq. (2) is 1.

To explain the general principle of how to select the data access nodes, we consider the following program statement: $e = a + b + c + d$. Assume, for the sake of illustration, that a and b are mapped to MC $\langle 0, 0\rangle$, and c and d are mapped to MC $\langle 0, 3\rangle$. Assume further that this computation (statement) is to be performed on core $\langle 1, 2\rangle$, that is, this statement is scheduled to be executed on core $\langle 1, 2\rangle$. For this example, as shown in Figure 5, the optimal data access node for variables a and b would be $\langle 0, 0\rangle$, and for variables c and d $\langle 0, 3\rangle$. With this data access placement, the request footprints (1) need not to go over any links since the MCs are attached to nodes $\langle 0, 0\rangle$ and $\langle 0, 3\rangle$. We also want to point out that, our data access placement does not affect the data message footprints (2). Generally, one can identify for each data element (required by a computation) the MC to which that data is mapped, and choose a data access node that is close to that MC (reducing L in Eq. (2)).

2.4 Computation Placement

We now introduce another optimization, called *Computation Placement*, to reduce the "data message footprints". We use the same example statement ($e = a + b + c + d$) to explain our strategy. As in the previous section, we assume that a and b are accessed through MC $\langle 0, 0\rangle$, c and d are accessed through MC $\langle 0, 3\rangle$, and the computation is originally placed on core $\langle 1, 2\rangle$. We further assume that the data access placement optimization has already been performed and the data access nodes are determined to be $\langle 0, 0\rangle$ and $\langle 0, 3\rangle$. The original data footprints of this example are depicted in Figure 6. In this figure, both a and b (footprints marked using (1)) need to go over 3 hops from the MC (attached to core $\langle 0, 0\rangle$) to the computation node $\langle 1, 2\rangle$, and c and d (footprints (2)) need to traverse 2 hops.

Our initial optimization is straightforward. The basic idea is to select a computation node that has the smallest average distance to the MCs. For the example above, we can choose the computation node as $\langle 0, 2\rangle$ instead of $\langle 1, 2\rangle$. The data footprints with this computation node placement are shown in Figure 7. Compared to Figure 6, a and b (footprints (1)) in Figure 7 need to traverse only 2 hops, and c and d (footprints (2)) need only 1 hop. The above optimization for computation placement actually reduces the lengths of the data footprints (L in Eq. (2)). Our next optimization, on the other hand, aims at reducing the "weights" of the data footprints (W in Eq. (2)). For example, $e = a + b + c + d$ can be rewritten as $e = f + g$, where $f = a + b$ and $g = c + d$. Observing that both a and b are requested by core $\langle 0, 0\rangle$, we can choose to perform the sub-computation ($f = a + b$) directly on core $\langle 0, 0\rangle$ as depicted in Figure 8. As a result, we only need to send data element f from $\langle 0, 0\rangle$ to $\langle 0, 2\rangle$ (footprint (2)). Even though the number of hops that need to be traversed by the data is still the same, the number of data elements to be sent has reduced from 2 (a and b) to 1 (f), thereby reducing the footprint weight W. Similarly, we can perform the sub-computation ($g = c + d$) directly in core $\langle 0, 3\rangle$ and send only one data element (g) from $\langle 0, 3\rangle$ to core $\langle 0, 2\rangle$ (footprint (3)). In general, the basic idea of reducing the "weights" of the data footprints is to first identify the "sub-operations" (sub-computations) of a computation that access data from the same MC and decompose the original computation accordingly (into these sub-operations). We then execute these sub-operations directly on the data access nodes, and send only the generated intermediate results (which usually contain less data) from the data access nodes to the computation node.

3. OVERVIEW OF OUR FRAMEWORK

The input to our framework is a parallelized code (currently, we employ shared memory programming model for the multicore architecture and can handle pthreads and OpenMP codes). Our optimization scheme consists of two phases: *data access placement* and *computation placement*, with the goal of reducing request footprints and data footprints, respectively. The computation placement phase can be further divided into two sub-steps, *computation decomposition* and *computation (node) placement*, in order to reduce the amount of data in the response messages as well as the number of links visited. The output of our framework is an optimized parallel code where footprints of the request and data messages are minimized. In this work, we target at optimizing "affine loop nests"[2], and employ the *polyhedral model* [5] for "pre-process" these loops. In the polyhedral model model, each iteration (in an n-dimensional loop nest) can be represented by an iteration vector $\vec{i} = (i_1, i_2, ..., i_n)^T$. The set of computations (computation domain) can be represented as a polytope $\mathcal{D}_c$ of n dimensions as follows:

$$\mathcal{D}_c : \; D_c \begin{bmatrix} \vec{i} & \vec{n} & 1 \end{bmatrix}^T \geq \vec{0}, \tag{3}$$

where D_c is the inequality matrix of the domain, and $\vec{n}$ is the vector of loop-independent parameters. To increase parallelism and improve data locality, the iteration space can be divided into smaller blocks (*tiles*) [4, 11, 10], each of which contains a subset of the iterations and is an "atomic" execution unit. All the necessary data needed for a tile must be *ready* before its execution starts. Also, all the output data will be available at the end of the tile execution. One important constraint in tiling is that, after partitioning the iteration space, the dependencies among the resulting tiles

[2]In these types of loops, the loop bounds and array references are affine functions of enclosing loop indices and loop-independent variables.

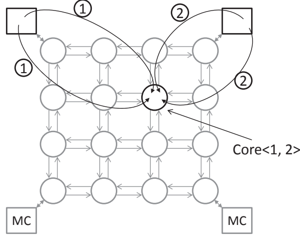

Figure 6: Data footprints before the computation placement optimization.

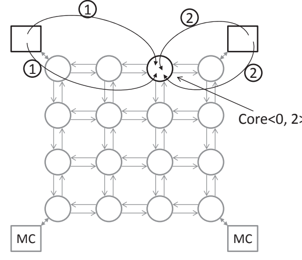

Figure 7: Data footprints after the computation (node) placement optimization.

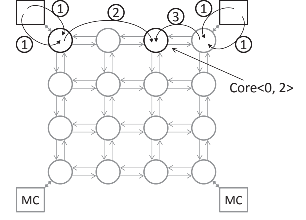

Figure 8: Data footprints after the computation decomposition optimization.

should form a "partial order". In other words, a legal tiling requires that no two tiles should be mutually dependent on each other. Note that while an application-specific data access pattern can in theory create hard-to-predict bottlenecks on the network, we did not observe that to happen in our data-parallel applications i.e, network congestion was more or less evenly distributed across the links.

4. TECHNICAL DETAILS

Recall that the execution of a statement in our approach is divided into two phases: *data access* and *computation.* Similarly, the execution of a tile can also be divided into such two phases, which gives us the flexibility to map the data access and computation phases of a given tile to different cores. where each vertex represents a tile, and each edge indicates a dependency relationship between two tiles. Our compiler fetches the ready tiles from the *Data Dependency Graph* (DDG) and maps the data accesses and computations in these tiles to cores. Due to the dependency constraints and resource limitations, it may not be possible to schedule all the tiles at the same time step (scheduling slot). Instead, we typically need multiple rounds to schedule the tiles. At each round, the number of tiles to be handled equals to either the number of currently ready tiles or the number of cores in the target machine, whichever is smaller.

4.1 Data Access Placement

We need to consider the following factors when performing our data access placement optimization for a tiled code. First, the data access nodes should be as close as possible to the MCs, in order to reduce L in Eq. (2). Second, we would like to *balance* the "data access workload" across all cores. If this is not done, the nodes that are closest to the MCs can be overloaded (become bottlenecks). Third, we need to ensure that the different data requests originating from a given tile will arrive at the MCs at similar time steps, i., the lengths of different request footprints should be similar. Otherwise, some data requests could be excessively delayed, which could in turn delay the computations in the tile.

We first determine how all the data elements referred in a tile are accessed through the MCs. To achieve this, we first need to locate the data block accessed by a given iteration tile. As discussed in Section 3, in the polyhedral model, each iteration can be represented by an iteration vector $\vec{i} = (i_1, i_2, ..., i_n)^T$ in an n-dimensional iteration space. Similarly, each data element in an array can also be denoted using a data vector $\vec{d} = (d_1, d_2, ..., d_m)^T$ in an m-dimensional space. Each data reference to an array in a loop nest can be written as: $\vec{d} = \mathbf{A}\vec{i} + \vec{o}$, where $\mathbf{A}$ is the access matrix (of $m \times n$) and $\vec{o}$ is the offset vector (of $m \times 1$). Therefore, given any iteration of a tile, we can determine which parts of an

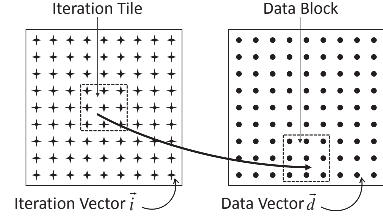

Figure 9: Example mapping from an iteration tile to a data block.

array are accessed based on the access matrix $\mathbf{A}$ and the offset vector $\vec{o}$. Specifically, let us assume that the polytope $\mathcal{D}_t$ for a tile is as follows:

$$\mathcal{D}_t : D_t \begin{bmatrix} \vec{i} & \vec{n} & 1 \end{bmatrix}^T \geq \vec{0}. \quad (4)$$

We next obtain the data points accessed by the iteration polytope $\mathcal{D}_t$. For each data reference $\vec{d}$, we have $\vec{d} = \mathbf{A}\vec{i} + \vec{o}$. We first concatenate $\vec{d}$ to the iteration vector $\vec{i}$ in Eq. (4). We then add columns of 0s as the coefficients corresponding to $\vec{d}$ in D_t to form D'_t, and obtain the following expression:

$$D'_t \begin{bmatrix} \vec{i} & \vec{d} & \vec{n} & 1 \end{bmatrix}^T \geq \vec{0}, \quad (5)$$

Using this expression and the equality defining the data access ($\vec{d} = \mathbf{A}\vec{i} + \vec{o}$), we can obtain the polytope $\mathcal{D}_d$ of the data block accessed by the tile. Figure 9 gives an example mapping from iteration space to data space, indicated by an arrow starting at the iteration tile and ending at the data block. Each point on the left box represents an iteration vector $\vec{i}$, while each point on the right space denotes a data vector $\vec{d}$. A loop nest can be generated to enumerate all the data points in $\mathcal{D}_d$ [2]. In the loop nest enumerating the data elements of a data block, we introduce two loops to replace the original innermost loop. The new second innermost loop determines the starting address of a contiguous data chunk that is mapped to a specific MC. The new innermost loop iterates over all the data elements in the contiguous chunk one by one and issues data requests to the MC. As can be observed, by obtaining the data block accessed by a tile, we only need to send one request for a specific data element, even though it may be used multiple times in the tile.

Once we have determined the MCs for all the data elements of a tile, we merge the data accesses to the same MC and *schedule them together* on the same data access node. In addition, on a data access node, if several data elements share the same cache line, we coalesce them into a single access and issue only one off-chip request message. As mentioned earlier, we may *not* simply choose the core that is closest to the MC, since we also want to maintain "data access load balance" across the cores. For this reason, we assign weights A_i to the cores based on their distances (the number of hops) to the associated MCs. Specifically, we choose

A_i to be inversely proportional to the number of hops. *This strategy turns out to be the best tradeoff between load balance and overall traffic reduction.* Thus, the shorter the distance from the core to the MC is, the larger is the weight of the core. If two cores have the same number of hops to a given (associated) MC, their weights will be the same. When we select which core to issue the data request messages, we need to check both a core's weight and its current workload R_i(i.e., the number of bytes to be accessed). Specifically, assume that a core with weight A_i currently has been assigned workload R_i. For a data request, we check all the cores associated with the MC, and keep those cores as candidate cores which satisfy the following condition: $\frac{R_i}{A_i} \leq \sigma$, where σ is a "workload balancing threshold". A delay in any data access can lead to a delay of the entire computation of the tile, which can in turn degrade the overall application performance. Therefore, the lengths (L) of the footprints generated by different data accesses in a tile should be similar. Our strategy to achieve this is as follows. For each set of data accesses A_i to an MC, we first choose its best available core D_i. Among all such cores, we identify the one with longest request footprint $Max(L)$. Then, the data requests to be issued by this core can be considered as the "bottleneck". As a result, for the data accesses to other MCs, we can choose any available core other than the current best available one, as long as the length of the newly-generated footprint is smaller than or equal to $Max(L)$. for scheduling other tiles, while keeping similar footprints for different data requests coming from the current tile. We perform our data placement for the ready tiles one-by-one based on the above steps, until all the ready tiles are processed.

4.2 Computation Placement

To reduce *data footprints*, our computation placement step should take into account the following factors. First, we would like to reduce the distances between MCs and computation nodes, which directly affects the data footprint lengths (L in Eq. (2)). Second, we want to reduce the amount of data (W in Eq. (2)) transferred from the MCs to the computation node, which can be achieved through computation decomposition. performed on the data access nodes and only the generated intermediate results are sent. Third, we want to balance the distances from different MCs to the computation node (in order to balance the corresponding data arrival times). Finally, we want to balance the computation workload among all the cores.

4.2.1 Computation Decomposition

The idea behind computation decomposition is to extract some sub-operations of a computation and execute them directly on the data access nodes. Such a sub-operation should not depend on the rest of the computation, and all the data it needs should be requested by the *same* data access node. We refer to a sub-operation that satisfies these two constraints as *independent operation*. Note that an independent operation may contain multiple operators.

When applying computation decomposition to a tile, the decomposition of the statements in the loop body may not always be the same. One reason is that the same reference to an array can access different MCs at different iterations. Consequently, for each iteration of a tile, we need to identify the set of independent operations $\mathcal{O}$ as much as possible by analyzing the relative priorities (precedences) of the operations, as well as the data access placement decisions. We first locate all the sub-operations that have the highest priority, and for each such sub-operation, we then check whether its data are requested by the same data access node or not. If so, we label this sub-operation as an independent operation and replace it with an intermediate variable in the final computation. We then schedule this sub-operation to be performed directly on the data access node, and send only the generated result to the computation node.

Table 1: Our default configuration.

Cores/ Caches	Processor: two-issue L1: 16 KB (per node), 64 byte lines, 2 ways L2: 256KB (per node), 64 byte lines, 16 ways
Memory	Number of Memory Controllers: 4 Total Capacity: 4GB
On-chip Networks	Size: 4×8 two-dimensional mesh Delays: 16B links, 2-cycle pipeline Routing: XY-routing

4.2.2 Computation Node Placement

As we have already pointed out, the computation node should have "balanced distances" to the *data sources*. Note that the data source here can be either a MC, or a data access node that executes a sub-operation. To this end, we take into account all the data sources of the current tile. For each data source $\langle D_{ix}, D_{iy}\rangle$, we assign it a weight G_i which is the amount of data to send to the computation node. When the data source is a data access node that executes a sub-operation, the generated intermediate data from the sub-operation will be counted towards the weight.

Once the weight G_i is available for each data source $\langle D_{ix}, D_{iy}\rangle$, we find the center $\langle C_x, C_y\rangle$ of all the data sources using the following formula:

$$\langle C_x, C_y\rangle = \langle \frac{\sum_{i=0}^{N-1} D_{ix}G_i}{N}, \frac{\sum_{i=0}^{N-1} D_{iy}G_i}{N}\rangle, \qquad (6)$$

where N is the number of data sources. If a data source has more data to send compared to other ones, the selected center node $\langle C_x, C_y\rangle$ will be closer to it. Consequently, the center node will have balanced (similar) distances to all the data sources. In case the center node $\langle C_x, C_y\rangle$ has already been assigned to another tile, we check this center node's available neighbors (i.e., those with one-hop distance). Among these cores $\langle C'_x, C'_y\rangle$, the one with the smallest value of $\sum_{i=0}^{N} G_i(|C'_x - D_{ix}| + |C'_y - D_{iy}|)$ is selected as the computation node. If none of the current neighboring cores is available, the distant neighbors will be checked until the computation node is decided. Therefore, in both data access and computation placement, we have mechanisms to maintain load balance (to prevent the nodes that are close to MCs from being overloaded).

5. EXPERIMENTAL ANALYSIS

The compiler component of our approach is implemented using the LLVM compiler infrastructure [1] and all the experiments presented below are carried out using the GEM5 tool-set [3]. We used a set of multi-threaded applications [12]. The "*baseline*" against which we compare our placement schemes is the original parallel applications with default data access/computation behavior (without our footprint reduction optimizations).

Figure 10 plots the "percentage reductions" in off-chip data access latency and total execution time, when both the components of our approach (data access placement and computation placement) are applied. The average latency on the network for off-chip data accesses has been reduced by 67.3% with our footprint reduction optimization. As a

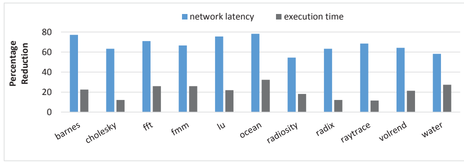

Figure 10: Percentage reductions of network latency and total execution time brought by our approach.

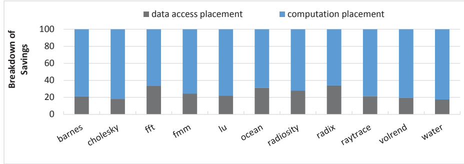

Figure 11: Breakdown of savings in execution time

result, it achieves an average of 21.1% reduction in total execution time. We observe that, even though the percentage reductions in network latency for applications *fmm* and *radix* are similar (66.6% vs 63.3%), the performance improvement for *fmm* is much higher than that for *radix* (26% vs 12.2%). This is because *fmm* is a highly memory-intensive application, whereas *radix* is not. Figure 11 quantifies the "individual contributions" of data access placement and computation placement components of our approach. One can observe that, the computation placement component is responsible for about 75.4% of our execution time improvements. The reason is that, the size of a request message is generally much smaller than the size of response message, which contains the actual data to be accessed. As a result, the response footprints contribute more to the network latencies for off-chip data accesses than the request footprints.

In each experiment presented in Figure 12 we only changed one parameter and remaining parameters are kept at their default values given in Table 1. From Figure 12, one can see that, when the issue width of the cores in the architecture is increased from 2 to 4, we observe a reduction in our improvements, since the latter configuration exhibits more data access parallelism. We also see that, our approach generates better savings with a larger (8×8) machine configuration. This is mainly because a larger manycore makes average distance-to-MC larger, rendering data access and computation placement more critical. Finally, our approach generates better savings with small number of MCs, as this configuration increases the pressure on individual MCs, demanding better data access and computation placement. We conducted eight experiments, and in each of them, our 4 MCs (default number) are placed differently. We observed an average execution time improvement of 21.8% with these MC placements.To understand how best our compiler-based approach performs, we conducted various experiments where the placement of data accesses and computation accesses are performed in an ideal fashion. "ideal data access placement" is that the data access node (for a given data access) is the one that is closest to the target memory controller and no load balancing problem is experienced."Ideal computation placement" is the one that uses the best location (node) to perform each sub-computation. The second bar for each benchmark in Figure 13 gives the percentage reduction in execution time with the ideal placements. We see that the ideal placement generates an average improvement of 27.2%, indicating that there is scope to further optimize data accesses in NoC based manycore system.

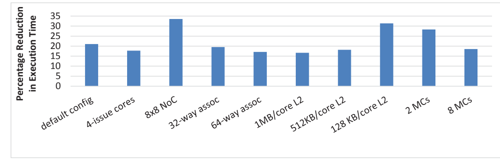

Figure 12: Sensitivity experiments with one parameter changed with others kept at their default values given in Table 1.

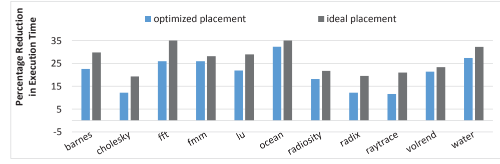

Figure 13: Results with ideal placement.

6. CONCLUDING REMARKS

We proposed a novel compiler-directed framework to optimize the data latencies for NoC-based manycores. Our proposed framework consists of two main components: data access placement and computation placement. In the data placement, the number of links visited were minimized while in computation placement, computation decomposition was proposed to reduce the amount of data sent in the response messages. This optimization framework cuts, on average, network latency of off-chip data accesses by 67.3% and application execution time by 21.1%, on a 4×8 manycore system.

7. ACKNOWLEDGMENTS

This work is supported in part by NSF grants 1213052, 1205618, 1439021, 0963839, 1017882, and a grant from Intel.

8. REFERENCES

[1] LLVM. *http://llvm.org/*.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. *PACT*, 2004.

[3] N. Binkert et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[4] U. Bondhugula et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. *CC*, 2008.

[5] U. Bondhugula et al. A practical automatic polyhedral parallelizer and locality optimizer. *PLDI*, 2008.

[6] R. Das et al. Application-to-core mapping policies to reduce memory system interference in multi-core systems. *HPCA*, 2013.

[7] M. Dashti et al. Traffic management: A holistic approach to memory placement on numa systems. *ASPLOS*, 2013.

[8] E. Kim et al. Energy optimization techniques in cluster interconnects. *ISLPED*, 2003.

[9] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. *MICRO*, 2010.

[10] A. Lim et al. An affine partitioning algorithm to maximize parallelism and minimize communication. *ICS*, 1999.

[11] A. W. Lim et al. Maximizing parallelism and minimizing synchronization with affine transforms. *POPL*, 1997.

[12] S. Woo et al. The splash-2 programs: characterization and methodological considerations. *ISCA*, 1995.