Dynamic Thread and Data Mapping for NoC Based CMPs

Mahmut Kandemir
Pennsylvania State University
University Park, PA
kandemir@cse.psu.edu

Ozcan Ozturk
Bilkent University
Turkey
ozturk@cs.bilkent.edu.tr

Sai P. Muralidhara
Pennsylvania State University
University Park, PA
smuralid@cse.psu.edu

ABSTRACT

Thread mapping and data mapping are two important problems in the context of NoC (network-on-chip) based CMPs (chip multiprocessors). While a compiler can determine suitable mappings for data and threads, such static mappings may not work well for multithreaded applications that go through different execution phases during their execution, each phase with potentially different data access patterns than others. Instead, a dynamic mapping strategy, if its overheads can be kept low, may be a more promising option. In this work, we present dynamic (runtime) thread and data mappings for NoC based CMPs. The goal of these mappings is to reduce the distance between the location of the core that requests data and the core whose local memory contains that requested data. In our experiments, we evaluate our proposed thread mapping and data mapping in isolation as well as in an integrated manner.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors-compilers

General Terms

Algorithms, Performance

Keywords

Thread, data, dynamic, mapping, NoC, CMP

1. INTRODUCTION

It is clear today that large, complex uniprocessors are no longer scaling performance-wise. This is because there is only a limited amount of parallelism that can be extracted from a single thread of execution, and even extracting this limited parallelism requires sophisticated superscalar instruction fetch, dependence check and instruction execution mechanisms. Unfortunately, it is not possible to continuously increase clock frequency of these architectures as it is well known that, beyond a point, power dissipation becomes a major bottleneck. In addition to these constraints, with extremely large numbers of transistors available on today's microprocessor chips, it is too costly to design and debug ever-larger and ever-complex processors in every two or three years. All these trends clearly point that chip multiprocessors (CMPs) are in very good position to become the next generation mainstream computer architecture. The fact that there are many applications from diverse set of application domains that can take advantage of thread-level parallelism (e.g., embedded multi-media codes, data-intensive simulation programs) further motivates for CMPs.

Since future technologies offer the promise of being able to integrate billions of transistors on a chip, the prospects of having hundreds to thousands of cores on a single chip along with an underlying memory hierarchy and an interconnection system is entirely feasible. Once the number of cores on one chip passes some technology and architecture dependent threshold, point-to-point buses will no longer be a sufficient interconnect structure. Thus, it is reasonable to assume that future CMPs will employ an NoC (network-on-chip) in order to be able to handle the required communications between the cores in a scalable, flexible, programmable, and reliable fashion.

Clearly, assigning (mapping) threads and data manipulated by threads (of a multithreaded application) to CMP nodes is a challenging task for extracting maximum performance. One option is to assign threads and data to CMP nodes statically at compile-time and not to change this assignment throughout the execution of the application. While an optimizing compiler can certainly perform thread-data assignments, it is not possible to capture dynamic modulations in data access patterns when such static mappings are employed. Instead, a dynamic (runtime) mapping strategy, if its overheads can be kept low, may be more promising for data-intensive applications which go through multiple phases during execution.

Data and computation mapping for generic multiprocessors have been widely studied. From the data mapping angle, previously proposed approaches range from generic data partitioning to techniques specifically target NoC based systems. Balasundaram et al [3] propose a static performance estimator to guide data partitioning decisions for generic data partitioning. In [12], authors present a data layout selection tool that generates data layout specifications of distributed-memory machines automatically. Kim et al [13] introduce a dynamic and static algorithms to partition the cache among the threads. Panda et al [16] implement a data partitioning algorithm in embedded processor-based systems. In the multi-core domain, Guz et al [8] propose Nahalal, an architecture which partitions data according to its usage as shared versus private data, and locates the private data close to each processor. Kandemir [10] present an interprocessor data reuse vector based data locality optimization scheme for CMPs. Jin et al [9] implement a page-level data to L2 cache slice mapping for CMPs. Chishti et al [6] propose a data mapping scheme which utilizes replication and capacity allocation to neighboring processors within a CMP.

Pop and Kumar [17] use multi-threaded processors as resources in NoC based CMPs. They map the concurrent applications to multi-threaded processors of the CMP off-line. Chen et al [5] compare two thread scheduling techniques on CMPs, namely, Parallel Depth First and Work Stealing. In

[2], authors propose a scheduling method for real-time systems targeted on CMPs. Chou et al [7] propose a run-time strategy for allocating the application tasks to platform resources in an NoC. Nuzzo et al [20] present task allocation strategies based on bin-packing algorithms. Kempf et al [11] propose and evaluate a task mapping framework in the context of heterogeneous chip multiprocessors.

In this work, our goal is to reduce data access latency by reducing the distance between the core that requests the data and the memory that holds the requested data. For this purpose, we first present an application-specific, dynamic thread assignment strategy for NoC based CMP systems. In this strategy, the job of thread-to-core assignment is given to a helper thread which carries out this task on behalf of the application and in parallel with the execution of application threads. Consequently, the resulting thread mappings are customized for an application and can change at runtime to adapt dynamic data sharing patterns exhibited by the threads of the application. We implemented this thread mapping strategy using a simulation platform and compared it to a static mapping scheme. The results from our experimental evaluation show that application-specific dynamic thread mapping can bring significant improvements in performance for all CMP sizes and applications we ran. More specifically, we achieve an average execution latency saving of 16.3% over static mapping.

Secondly, we present a dynamic data-to-core mapping scheme, also implemented using a helper thread. The results collected show that dynamic data mapping alone brings an improvement of 8.2% on average. More importantly, our integrated thread-data mapping scheme, which alternates between thread mapping and data mapping in successive mapping periods (epochs), takes these savings (over the static mapping) to 29.1% when all the applications are considered. Our results clearly demonstrate that to achieve the best performance both thread mapping and data mapping should be considered together. The results also indicate that the integrated scheme achieves consistent savings under different thread counts and CMP sizes.

2. EXECUTION MODEL

We assume an NoC based CMP architecture in which each node has a processor core, a private L1 cache (instruction and data) and a portion of the shared on-chip main memory space. Note that, while the on-chip memory space is distributed across CMP nodes, the entire memory space is logically shared, i.e., a core can access the memory attached to any node. It should be emphasized however the distance between the node of the requesting core and the node that holds the requested data element in its memory can make significant difference in performance (in data access latency), and in fact, our main goal in this work is to reduce this distance through dynamic thread mapping and data mapping across CMP nodes.

We assume that, when requested by an application, operating system (OS) gives a set of nodes (called *allocation* in this paper) to the requesting application and the application has full control in assigning its threads and data anywhere within its allocation. We further assume that the allocation for an application does not change during its execution, until it finishes. We also assume the existence of two types of migration support in this architecture: *thread migration* and *data migration.* The former moves a thread from one location in the allocation to another, whereas the latter moves a data block from one location to another within the allocation. While it is conceptually possible to modify an application code (i.e., thread bodies) to insert explicit "thread move" and "data move" instructions, this can have significant degradation in performance of the application. Therefore, in this work, we explore a different option which employs *helper threads* to migrate threads and data blocks across CMP nodes. Specifically, we use two helper threads, which are created at the time the application has been initiated. One of these threads moves application threads across the CMP nodes at runtime and the other moves data blocks. Note that a helper thread can execute in parallel with application threads. The overall goal of these helper threads is to reduce the distance between the core that requests a data block and the memory that holds the requested data block. While we focus on a two-dimensional (2D) mesh-based NoC (i.e., CMP nodes form a 2D mesh), our approach is applicable to other types of on-chip communication network topologies as well, as long as the target topology is exposed to our scheme. In the rest of this paper, we use the terms "mapping" and "migration" interchangeably. Basically, what we mean by "migrating a thread/data" is to "re-map" it from one location to another.

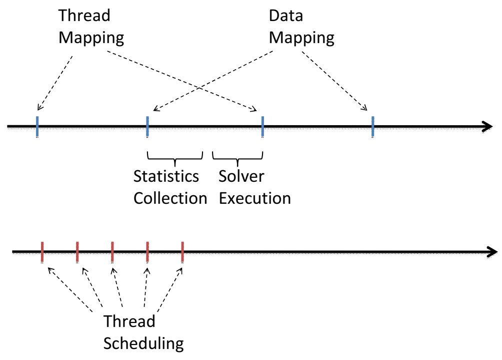

Figure 1: Thread migrations, data migrations and thread schedulings. Time flows from left to right.

Figure 1 depicts how thread migrations, data migrations and thread schedulings take place on the timeline. Typically, thread scheduling (performed by the process scheduler in that node) is invoked much more frequently; in comparison, data and thread migrations (carried out by helper threads as explained above) are less frequent and they alternate (in our integrated scheme)—i.e., a set of thread migrations followed by a set of data migrations, which in turn are followed by a set of thread migrations, and so on. In each thread (or data) migration interval, the first half of the interval is used to collect runtime statistics, which indicate the frequency of accesses to each data block from each core. In the second half, an ILP (integer linear program) solver is invoked and the new thread (or data) mapping is determined and implemented. Note that the new mappings take place only after the ILP solver returns a solution, and at each step, the number of threads (or data blocks) to be migrated is dictated by the changes in the data access patterns of the threads between the previous interval and current interval.

In the following two sections, we describe the thread migration and data migration components of our approach. As mentioned above, in our integrated scheme, thread migrations and data migrations interleave at an (mapping) interval granularity. That is, at the end of the first interval (see Figure 1), threads are migrated; at the end of the second interval, data blocks are migrated; at the end of the third, threads are migrated again, and so on. For deciding both thread and data migrations (at each interval), we use the ILP based approaches explained below.

3. DYNAMIC THREAD MAPPING

We now present our ILP based formulation of dynamic thread mapping targeting 2D mesh NoC based CMP architecture. Our goal is to assign these threads to the CMP nodes such that the overall distance-to-data is reduced across all threads. To achieve this, our approach tries to place the threads that share a lot of data between them in close-by locations (cores) in the CMP.

Constant	Definition
$X \times Y$	Number of CMP nodes
T	Number of threads
$G(T)$	Graph representing thread affinities
$E(T)$	Edges in graph $G(T)$
$V(T)$	Vertices in graph $G(T)$
w_e	Weight of edge e or degree of affinity between the corresponding threads
C_{comm}	Unit cost of data access in the CMP

Table 1: The constant terms used in our dynamic thread mapping formulation. These are either architecture specific or program specific. $G(T)$ is built by collecting statistics during execution at each interval.

An integer linear program, or ILP for short, tries to solve a linear objective function via linear functional constraints along with integer solution variables. In 0-1 ILP, however, each solution variable is restricted to be either 0 or 1 [15]. Table 1 gives the constant terms used in our ILP formulation of the thread mapping problem. Although choice of the ILP tool is orthogonal to the focus of this paper, we formulated our ILP problem using a commercial tool, *Xpress-MP* [1]. Recall that in our execution model, the ILP solver is invoked by the helper thread. For the sake of our formulation, we view the CMP as a 2D grid, and assign the threads to the nodes of this 2D grid. Assume that the target CMP is composed of $X \times Y$ nodes, where X denotes the number of nodes on the x-axis and Y denotes the number of nodes on the y-axis. We denote a specific node in this grid using (x, y).

We use 0-1 integer variables R to indicate where the threads are mapped (once our solver returns a solution). More specifically,

- $R_{i,j,k}$: indicates whether thread i is running on (j, k).

The distance between two threads is captured using D_{t_1,t_2}. Specifically, we have:

- D_{t_1,t_2} : indicates the distance between threads t_1 and t_2.

We next give our constraints based on these definitions. A thread denoted by t can be assigned to a unique node in the CMP:

$$\sum_{j=1}^{X}\sum_{k=1}^{Y} R_{t,j,k} = 1, \quad \forall t. \tag{1}$$

We also need to make sure that only one thread is mapped to a node (x, y) (this assumption can be dropped if there are more threads than CMP nodes, or can be limited by an upper bound; in our implementation, we have a parameter that limits the number of threads per node, which is set to 4 in our default configuration):

$$\sum_{i=1}^{T} R_{i,x,y} \leq 1, \quad \forall x, y. \tag{2}$$

We use the Manhattan Distance to capture the cost of the placement of threads that share data. Manhattan Distance is given as the distance between two points measured along axes at right angles. In a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2), it is $|x_1 - x_2| + |y_1 - y_2|$. We assume this distance to be the cost of the relative placements (on the CMP) of thread t_1 and thread t_2. To capture the Manhattan Distance, we use the variable, D_{t_1,t_2} and employ the following constraints:

$$D_{t_1,t_2} \geq (R_{t_1,j_1,k_1} + R_{t_2,j_2,k_2} - 1) \times (|j_1 - j_2| + |k_1 - k_2|) \quad \forall t_1, t_2, j_1, j_2, k_1, k_2. \tag{3}$$

Using the above constraint, we capture the distance between thread t_1 and thread t_2. Note that, R_{t_1,j_1,k_1} is 1 if t_1 is running on CMP node (j_1, k_1), whereas R_{t_2,j_2,k_2} is 1 if t_2 is running on CMP node (j_2, k_2). The role of the first term in the above constraint is to perform a logical 'and' on R_{t_1,j_1,k_1} and R_{t_2,j_2,k_2}. More specifically, if both of these 0-1 variables are equal to 1, then the first term in the constraint is equal to 1. We multiply this term with the absolute distance between the two nodes given with (j_1, k_1) and (j_2, k_2). As can be observed from this expression, the first term given with $(R_{t_1,j_1,k_1} + R_{t_2,j_2,k_2} - 1)$ is going to be either 0 or negative if threads t_1 and t_2 are not running on the indicated CMP nodes. However, our ILP solver will pick the maximum value among all the possible D_{t_1,t_2} in order to satisfy all the constraints. Hence, we will have the distance in variable D_{t_1,t_2}.

Our helper thread employs a *weighted directed graph* to indicate the data sharing volume between threads. In this graph, each vertex represents a thread in the program and an edge between two vertices indicates the data sharing between the corresponding threads. An edge is annotated with a *weight* which is multiplication of the amount of data shared between the threads and the total number of references made to these shared data elements by the two threads. Note that this weight in a sense represents the *affinity* between the two threads. Clearly, two threads with high affinity should be as close to each other as possible in the 2D grid. In other words, placing two threads with high affinity in locations (on the CMP) far away from each other can incur high access latencies to data which should be avoided.

In mathematical terms, we map the given threads, T, to its graph representation $G(T)$, where $G(T) = V(T) \cup E(T)$ and $E(T) \subseteq V(T) \times V(T)$.

As stated above, each edge carries a weight value between the vertices of the graph. Since we have at this point covered mapping constraints and distance constraints, we can now give our objective function. Our objective function actually targets minimizing the distance between threads of high affinity and does not consider specific optimization problems such as power and performance. However, this formulation, if desired, can easily be converted to serve such specific goals. We express our objective function using the aforementioned graph structure as an input. More specifically, the cost can be captured, for each edge e in $G(T)$, using the distance between the two threads designated by the vertices connected by this edge. This distance is multiplied with the weight of the corresponding edge along with the unit data access cost, which is a constant. We can express this as follows:

$$Comm = \sum_{\forall e \in E(T)} w_e \times D_{e_{v1},e_{v2}} \times C_{comm}. \tag{4}$$

In the above expression, we assume that C_{comm} is the unit cost of data access within the NoC. We multiply this with the volume of the data access, which is specified as w_e, i.e., the weight of the edge between two threads. Distance, $D_{e_{v1},e_{v2}}$, captured in our previous constraints, is used to obtain the overall cost of placement of two threads. It is to be noted that e_{v1} and e_{v2} are the two vertices connected by edge e which actually represent the two threads. Based on these constraints, we can express our objective function as min $\quad Comm$.

4. DYNAMIC DATA MAPPING

Our goal in this section is to present an ILP formulation of the problem of minimizing data access latencies by determining the optimal placement of data blocks in the CMP. A data block in this section corresponds to a set of consecutive cache lines. Each dataset is assumed to be divided into *blocks* of equal size, which is the granularity of migration in our scheme across the memories attached to CMP nodes. Unless stated otherwise, we allow at most one copy of each data block to exist within the cumulative on-chip memory space (data cache can have its own copy). As previously stated, we assume a 2D grid CMP composed of $X \times Y$ nodes, and we represent a specific node in this 2D grid using (x, y). Table 2 gives the constant terms used in our ILP formulation of the data mapping problem.

We use A to indicate where a data block is assigned within the CMP (once the ILP solver return a solution). More specifically,

Constant	Definition
$X \times Y$	Number of nodes
D	Number of data blocks
l_m	Size of a local memory of a core
$BSize$	Size of a data block
$CT_{i,j,k}$	Number of accesses to data block i by processor (j,k)
C_{tr}	Cost of moving a data block to an adjacent core
C_{acc}	Cost of accessing a data block from the local memory
$C_{off-chip}$	Cost of bringing a data block from off-chip memory

Table 2: The constant terms used in our dynamic data mapping formulation. These are either architecture specific or program specific. The values of $CT_{i,j,k}$ are obtained by collecting statistics at runtime.

- $A_{i,j,k}$: indicates whether data block i is in core (j,k).

Similarly, M is used in our formulation to identify whether a data block is in off-chip memory. Observe that the cumulative on-chip memory space may not be sufficient, in general, to store all the data blocks manipulated by the multithreaded program.

- M_i : indicates whether data block i is in off-chip memory.

The distances between a core and a data block is captured using $D_{i,j,k}$. Specifically:

- $D_{i,j,k}$: indicates the distance between data block i and core (j,k).

Based on these definitions, we can start describing our constraints. A data block given by d needs to be assigned to a unique core or off-chip memory:

$$M_d + \sum_{j=1}^{P}\sum_{k=1}^{Q} A_{d,j,k} = 1, \quad \forall j,k. \tag{5}$$

As before, the Manhattan Distance is a factor in determining the cost when data block d is accessed by processor core (x,y). This is also referred to as the *data access cost* in this section, and is the metric whose value we want to minimize. We want to remind the reader that a data block can be shared across multiple cores. To capture the Manhattan Distance, we use variables $D_{d,x,y}$ and accommodate the following constraints:

$$D_{d,x,y} \geq \sum_{j=1}^{X}\sum_{k=1}^{Y} A_{d,j,k} \times (|x-j| + |y-k|) \forall j,k. \tag{6}$$

In the above constraint, we capture the distance of data block d to the corresponding core denoted using (x,y). Note that, $A_{d,j,k}$ is used to indicate the location of the data block and we take the vertical and horizontal distances with the core (x,y). As can be seen from this expression, in the case of data not being stored in the on-chip memory, the distance value given with $D_{d,x,y}$ will be returned as 0. However, as we will explain in more detail, the off-chip memory access costs will be captured separately.

We also need to make sure that the local memory size (capacity), which is given by l_m, is not exceeded. This can be captured using the following constraint:

$$\sum_{i=1}^{D} A_{i,x,y} \times BSize \leq l_m \forall x,y. \tag{7}$$

This expression sums up the total amount of space required to keep the assigned data blocks within the local memory of core (x,y). $BSize$ indicates the data block size being used and may vary depending on the implementation and the data block granularity. Although it is possible to add a constraint to check the overall on-chip memory size, this is not necessary as we do it for individual local memories.

We next discuss our objective function. For clarity reasons we give our objective function in two main components. First, on-chip data access cost can be captured, for each data block access, using the distance between the location of that data block and the location of the core accessing it multiplied with the cost of moving this data block within the chip, C_{tr}. On top of this cost, we have the local access cost which is given by C_{acc}. This will then be multiplied by the frequency of these accesses. As a result, we have:

$$TC_{on-chip} = \sum_{i=1}^{D}\sum_{j=1}^{X}\sum_{k=1}^{Y}((D_{i,j,k} \times C_{tr} + C_{acc}) \times CT_{i,j,k}). \tag{8}$$

In a similar fashion, we can compute the off-chip memory access cost using the following expression:

$$TC_{off-chip} = \sum_{i=1}^{D}\sum_{j=1}^{X}\sum_{k=1}^{Y}((M_i \times C_{off-chip} + C_{acc}) \times CT_{i,j,k}). \tag{9}$$

In the above expression, we assume that $C_{off-chip}$ captures the cost of bringing the data block to the local memory of the requesting core. That is, we assume that each core can access the off-chip memory with uniform latency. Alternatively, if the accesses are not uniform for some reason, one can introduce individual off-chip access costs for each core such as $C_{x,y}$ for off-chip access cost of core (x,y). Thus, our overall objective function can be written as:

$$\min \quad TC = TC_{on-chip} + TC_{off-chip}. \tag{10}$$

Within each interval (at the end of which we want to migrate data blocks), the helper thread collects access statistics for each data block, build the ILP formulation summarized above, and invokes the ILP solver to solve it. Based on this solution, the required data block migrations are performed by the helper thread.

Data Replication. Recall that so far in our ILP formulation, we assumed that there is only one copy of each data block, either on-chip memory or off-chip memory. We can expand the scope of our formulation by allowing data block replication within the local memories if this helps to reduce the total data access costs. We replicate a data block only if the block is read-only (to avoid coherence-related issues). In order to reflect these changes on our baseline formulation, we modify the constraint given in Expression 5 as follows:

$$M_d + \sum_{j=1}^{X}\sum_{k=1}^{Y} A_{d,j,k} \geq 1, \quad \forall j,k. \tag{11}$$

This modified constraint enables us have multiple copies of the data block in different local memories, i.e., for a specific data block d, we can have multiple $A_{d,j,k} = 1$. However, this requires additional modifications such as selecting the closest data block in case of multiple data blocks residing within the local memories. In order identify the closest data block, we define an additional 0-1 variable, $N_{d,j,k,l,m}$. More specifically:

- $N_{i,j,k,l,m}$: indicates whether core (j,k) can access the closest data block i from core (l,m).

A core (j,k) can only access a data block d if it resides in the said core (x,y), therefore:

$$N_{d,j,k,x,y} \leq A_{d,x,y}, \quad \forall j,k. \tag{12}$$

Also, we have to make sure that there is one and only one closest data block d for a core (j,k). Note that, if the data block is not located within local memories, then the closest location is the off-chip memory, which can be expressed as M_d. In mathematical terms:

$$M_d + \sum_{l=1}^{X}\sum_{m=1}^{Y} N_{d,j,k,l,m} = 1, \quad \forall j,k. \tag{13}$$

In addition to these constraints, we have to modify Expres-

NoC topology	5 × 5 2D mesh
Core (fetch,issue,retire width)	(4,2,2)
L1 cache	16KB per node, 2-way, 2 cycle
On-chip memory	256KB per node, banked, 10 cycle
Scheduling frequency	50 msec
Mapping frequency	500 msec
Data block size	512 bytes

Table 3: Our simulation parameters and their default values.

sion 6 accordingly:

$$D_{d,x,y} \geq \sum_{l=1}^{X} \sum_{m=1}^{Y} N_{d,x,y,l,m} \times (|x-l| + |y-m|) \forall l, m. \quad (14)$$

As explained earlier, we only allow a certain number of copies for a given data block, which is τ. This requirement can be enforced using:

$$\sum_{j=1}^{X} \sum_{k=1}^{Y} A_{d,j,k} \leq \tau, \quad \forall d. \quad (15)$$

In the above expression, the total number of copies of data block d stored within the on-chip memory space is captured with the *sum*. Finally, our objective function give earlier remains the same since both $TC_{on-chip}$ and $TC_{off-chip}$ are the same as in our original ILP formulation.

5. EXPERIMENTAL EVALUATION

The baseline thread and data mapping used in our experiments is a static one, where a compiler determines appropriate data-thread mapping statically and these mappings do not change during the course of execution. Note that, this static scheme is in fact derived from the ILP based formulations explained in Sections 3 and 4. Specifically, we analyze the entire application and record the number of accesses made by threads to data blocks and the affinities between thread pairs (for the entire execution). Once these counts have been obtained, we first execute our thread mapping scheme and then our data mapping scheme. Note that, this baseline static mapping scheme can also be seen as a specific instance of the dynamic mapping scheme where the mapping interval is set to the entire execution period. In the results presented in the next subsection, all values are given as *normalized* with respect to this static scheme. In addition to this static scheme, we conducted experiments with the following dynamic schemes:

• TM. In this scheme, data mapping is decided statically by the compiler (as in the static mapping case) and only dynamic thread mapping is applied at runtime.

• DM. This is in a sense the opposite of scheme TM. In this scheme, thread mapping is fixed at compile time (as in the static mapping case) and only dynamic data mapping is exercised at runtime.

• TM+DM (C). This is our integrated scheme which exercises both data and thread mapping at runtime. The initial data and thread mappings are decided by the compiler. And at runtime, we use data and thread mappings in alternate fashion as explained earlier.

• TM+DM (R). This is similar to the previous scheme except that the initial thread and data mappings are random, that is, threads and data blocks are assigned to the CMP nodes randomly at compile time but we employ the integrated scheme at runtime.

All these mapping schemes are implemented on top of SIMICS [14] and have been tested using the platform summarized in Table 3. Recall that the target architecture is a shared memory based one, i.e., although each core has a portion of the on-chip memory space (as its local memory), all on-chip memory space are accessible (albeit with different costs) by all cores. Table 4 on the other hand lists the benchmark codes used in this study. We selected five parallel benchmark codes from the Splash-2 benchmark suite [19] and executed them under all the mapping schemes discussed above. For each benchmark code in our experimental suite, 100 threads are generated and mapped onto 25 CMP nodes (we later present results with different number of threads and different number of CMP nodes). Finally, in all the experiments we made, the default node-level thread scheduling scheme is round-robin (clearly, if there is only one thread per node, no scheduling is needed).

Program	Application Domain	Dataset Size (MB)	Execution Latency (sec)
radiosity	Graphics	6.44	7.21
radix	General	5.68	6.17
raytrace	Graphics	4.96	5.89
volrend	Graphics	7.26	8.73
water	Simulation	5.17	5.37

Table 4: Benchmark codes used in our evaluation. The last column gives the execution latency values for the baseline static (thread and data) mapping scheme.

Figure 2 gives the normalized execution latencies under different mapping policies. Our first observation is that while both TM and DM generate better results than the static thread-data mapping, the integrated scheme generates the most savings for all the benchmarks. The second observation is that the difference between TM+DM (C) and TM+DM (R) is not significant, meaning that initial mapping may not matter too much if the dynamic re-mapping is activated at runtime. We also see that TM performs better than DM. This is mainly because, in general, a data block can be shared by more than one thread and positioning threads with respect to that data turns out to be easier than finding a good location for that data. The average execution latency improvements brought by TM, DM, TM+DM (C) and TM+DM (R) over the static scheme are 16.3%, 8.2%, 29.1% and 26.7%, respectively.

The sensitivity of these mapping schemes to the number of threads is presented in Figure 3. Each bar in this plot represents the average (normalized) execution latency across all five benchmarks. Recall that the default number of threads was 100 for our benchmarks. We see from these results that the integrated scheme consistently generates good results for all the thread counts considered. The results are a bit higher with larger thread counts as it gives more flexibility to our mapping scheme. Figure 4 on the other hand gives the sensitivity of the mapping schemes with respect to the number of nodes. In addition to our default 5 × 5 mesh, we experimented with 4 × 4 and 6 × 6 mesh sizes (and the number of threads is fixed at 100 in this experiment). The average execution latency results are presented in Figure 4 indicate that all dynamic mapping schemes achieve slightly better results with larger meshes as larger meshes give more flexibility to a dynamic approach in moving data and threads across CMP nodes.

We next study the sensitivity of our dynamic mapping schemes to the mapping frequency. Recall from Table 3 that the mapping frequency used in experiments so far was 500 msec. The results with mapping frequencies of 100 msec, 200 msec, 800 msec, and 1000 msec are shown in Figure 5. Maybe the most important observation from these results is that working with very low or very high frequencies may not be the best option. Though the results change from one benchmark to another, we see that, for each benchmark, there is an ideal value (among the frequency values tested) that generates the maximum improvement in execution latency.

So far in our experiments no data block is replicated across CMP nodes, i.e., each data block has exactly one copy in the on-chip memory space. Figure 6 plots the results when each data block is replicated n times (x-axis), where $1 \leq n \leq 5$. Note that, when n is 1, we have no replication. We note that, as may be expected, the DM scheme takes advantage of block replication. This also reflects on, to varying extent, other schemes as well. However, we also witness a reduction in improvements (in some cases) as we move from $n = 4$ to $n = 5$. This is because too much replication reduces effective

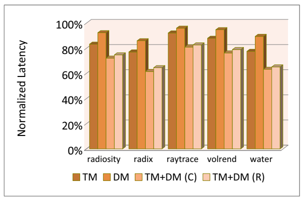

Figure 2: Normalized execution latencies under different mapping policies.

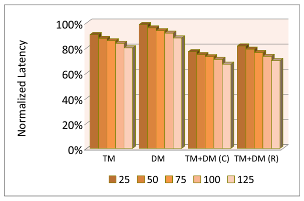

Figure 3: Sensitivity of the mapping schemes to the number of threads.

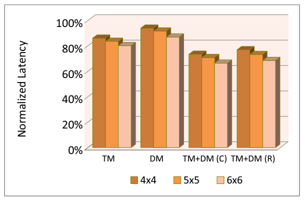

Figure 4: Sensitivity of the mapping schemes to the number of nodes.

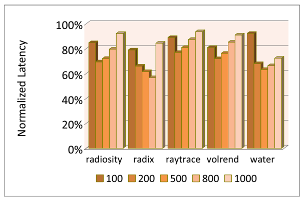

Figure 5: Sensitivity of the dynamic mapping schemes to the mapping frequency.

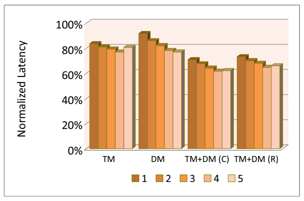

Figure 6: Results when each data block is replicated n times (x-axis), where $1 \leq n \leq 5$.

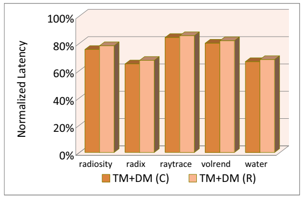

Figure 7: Results with an alternate mapping invocation scheme.

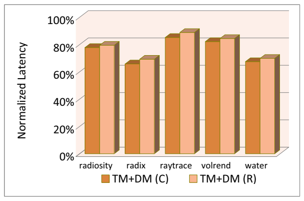

Figure 8: Impact of the shape of the allocation.

capacity of on-chip memory space, and this increases average access latency. Ideally, one may want a dynamic scheme that selects the best n value at runtime; exploring such a scheme is in our future research agenda.

We now change the order in which thread mapping and data mapping are invoked at runtime. Recall that our default approach is to invoke first thread migration and then data migration, and continue with this pattern until the end of execution. Figure 7 illustrates the results with an alternate scheme which invokes thread mapping twice followed by two invocations of data mapping, which in turn is followed by two invocations of thread mapping, and so on. Compared to the results in Figure 2, the results in Figure 7 are slightly worse. Clearly, there are many other potential invocation patterns and we plan to explore this issue further in our future work. We want to mention here however invoking both thread mapping and data mapping in the same interval did not help too much as it caused conflicting migrations for threads and some data blocks.

In our final set of experiments, we study the impact of the shape of the allocation (as defined in Section 2). All the results presented so far have been collected assuming a 2D mesh allocation (5×5 by default). We also performed experiments when the set of nodes allocated to an application is selected randomly. More specifically, we assumed a set of 25 nodes are assigned to an application randomly from a 6×6 mesh based CMP. The values of all the other experimental parameters are as given in Table 3. The results presented in Figure 8 show that the integrated scheme generates significant improvements in this scenario as well. In fact, compared to the savings shown in Figure 2, those in Figure 8 are only slightly worse.

6. CONCLUSIONS

As processor design has become severely power limited, it is now commonly accepted that staying on the current performance trajectory will come about through the integration of multiple processors on a chip rather than through increases in the clock rate of single processors. As a result, several manufacturers already have CMP architectures on the market, and we can expect future CMPs to employ NoC for scalable and reliable communication. In this paper, we have presented dynamic thread and data mapping schemes for NoC based CMPs and evaluated their performance. The results from our experimental evaluation show that application-specific thread mapping can bring significant improvements in performance for all CMP sizes and applications we tested.

7. ACKNOWLEDGMENTS

This work is supported in part by NSF grants 0811687, 0720645, 0720749, 0702519, a grant from Microsoft Research and a grant from GSRC.

8. REFERENCES

[1] Xpress-mp. *http://www.dashoptimization.com/pdf/Mosel1.pdf*, 2002.
[2] J. H. Anderson and J. M. Calandrino. Parallel real-time task scheduling on multicore platforms. In *Proc. of RTSS '06*.
[3] V. Balasundaram et al. A static performance estimator to guide data partitioning decisions. In *Proc. of PPOPP '91*.
[4] G. Chen et al. Application mapping for chip multiprocessors. In *Proc. of DAC '08*.
[5] S. Chen et al. Scheduling threads for constructive cache sharing on cmps. In *Proc. of SPAA '07*.
[6] Z. Chishti et al. Optimizing replication, communication, and capacity allocation in cmps. In *Proce. of ISCA '05*.
[7] C.-L. Chou and R. Marculescu. User-aware dynamic task allocation in networks-on-chip. In *DATE*, pages 1232–1237, 2008.
[8] Z. Guz et al. Utilizing shared data in chip multiprocessors with the nahalal architecture. In *Proc. of SPAA '08*.
[9] L. Jin et al. A flexible data to l2 cache mapping approach for future multicore processors. In *Proc. of MSPC '06*.
[10] M. Kandemir. Data locality enhancement for cmps. In *Proc. of ICCAD '07*.
[11] T. Kempf et al. A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms. In *DATE*, pages 876–881, 2005.
[12] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM TOPLAS*, 20(4):869–916, 1998.
[13] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. of PACT '04*.
[14] P. S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.
[15] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA, 1988.
[16] P. R. Panda et al. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, 2000.
[17] R. Pop and S. Kumar. Mapping applications to noc platforms with multithreaded processor resources. *NORCHIP Conference, 2005. 23rd*, pages 36–39, Nov. 2005.
[18] V. Suhendra et al. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proc. of CASES '06*.
[19] S. C. Woo et al. The splash-2 programs: characterization and methodological considerations. In *Proc. of ISCA '95*.
[20] J. Zhuo and C. Chakrabarti. System-level energy-efficient dynamic task scheduling. In *Proc. of DAC '05.*.