Scratchpad-Memory Management for Multi-threaded Applications on Many-Core Architectures

VANCHINATHAN VENKATARAMANI, MUN CHOON CHAN, and TULIKA MITRA,

National University of Singapore, Singapore

Contemporary many-core architectures, such as Adapteva Epiphany and Sunway TaihuLight, employ per-core software-controlled Scratchpad Memory (SPM) rather than caches for better performance-per-watt and predictability. In these architectures, a core is allowed to access its own SPM as well as remote SPMs through the Network-On-Chip (NoC). However, the compiler/programmer is required to explicitly manage the movement of data between SPMs and off-chip memory. Utilizing SPMs for multi-threaded applications is even more challenging, as the shared variables across the threads need to be placed appropriately. Accessing variables from remote SPMs with higher access latency further complicates this problem as certain links in the NoC may be heavily contended by multiple threads. Therefore, certain variables may need to be replicated in multiple SPMs to reduce the contention delay and/or the overall access time. We present Coordinated Data Management (CDM), a compile-time framework that automatically identifies shared/private variables and places them with replication (if necessary) to suitable on-chip or off-chip memory, taking NoC contention into consideration. We develop both an exact Integer Linear Programming (ILP) formulation as well as an iterative, scalable algorithm for placing the data variables in multi-threaded applications on many-core SPMs. Experimental evaluation on the Parallella hardware platform confirms that our allocation strategy reduces the overall execution time and energy consumption by **1.84x** and **1.83x** respectively when compared to the existing approaches.

CCS Concepts: • **Computer systems organization → Embedded software**;

Additional Key Words and Phrases: Scratchpad memory management, Many-core architectures

ACM Reference format:
Vanchinathan Venkataramani, Mun Choon Chan, and Tulika Mitra. 20XX. Scratchpad-Memory Management for Multi-threaded Applications on Many-Core Architectures. *ACM Trans. Embedd. Comput. Syst.* X, X, Article XX (December 20XX), 25 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Many-core architectures containing tens or hundreds of cores on chip are emerging in different domains, ranging from embedded systems to server clusters, for meeting ever-increasing performance requirements. Typical many-core architectures consist of homogeneous or heterogeneous cores with multiple levels of coherent data- and instruction-caches connected using Network-on-Chip (NoC) for fast communication. Having multiple levels of coherent caches is

This work is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Industry-IHL Partnership Grant NRF2015-IIP003 and Huawei International Pte. Ltd. Authors' addresses: V. Venkataramani, C. Mun Choon, T. Mitra, School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417. Authors' Email addresses: {vvanchi, chanmc, tulika}@comp.nus.edu.sg.

Manuscript submitted to ACM

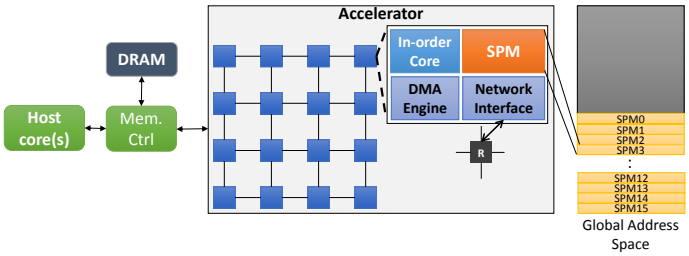


Fig. 1. A generic SPM-based many-core architecture and its memory address space.

not scalable due to the directory structures used for maintaining the state of the blocks in different caches. A number of recent works [16, 54] have proposed mechanisms for either area/power reduction of directory-based cache coherence or support for non-coherent caches [33]. However, these mechanisms are not sufficient when scaling to systems with thousands of cores [10].

Software programmable or Scratchpad Memory (SPM) [7] has been used as an alternative to caches in embedded systems due to energy-efficiency, timing predictability, and scalability. An SPM contains an array of SRAM cells. A portion of the memory address space is dedicated to the SPM. Any address that falls within this dedicated address space can directly index into the SPM to access the corresponding data. Thus, SPMs are power-efficient as they do not require tag arrays and comparators that are essential to caches. Coherency among multiple SPMs is maintained at the software level, thereby eliminating hardware area/power required for cache coherence. The downside is that either the compiler or the programmer needs to take an active role in allocating appropriate data to the SPM explicitly and efficiently. Therefore, data management is the single most challenging issue in systems equipped with SPMs. Since data is managed explicitly, the programmer knows the latency for each memory access. Thus, SPM based architectures are also extensively used for their timing predictability. Still, many emerging architectures are deploying SPM as on-chip memory due to its aforementioned benefits. These include embedded many-core architectures like IBM Cell [13], Adapteva Epiphany [35], and Kalray MPPA [24]. SPM-based many-cores have now also made their way into super-computing domain including the fastest and most power-efficient supercomputer [46] Sunway TaihuLight [18].

Figure 1 shows a simplified schematic of the new generation of SPM based many-core architectures (e.g., Epiphany, TaihuLight). Each core contains a unified SPM for instructions and data. Each SPM holds a distinct portion of the global address space to be used by the applications. As the address space is global, a core can access the data in remote SPMs as well, transparently supported by the underlying architecture. The cores are connected to a NoC that enables a core to access remote SPMs with varying latency based on the distance. Each core is equipped with a DMA (direct memory access) engine to transfer data between the off-chip memory and the local SPM.

Data management for single-threaded applications is a very difficult problem focusing on accurately identifying the “hot" data to be placed in the SPM. Many-core architectures with remote SPM access option adds another layer of complexity as data can be allocated and accessed from a remote SPM (higher access latency than local SPM) rather than off-chip memory. Additionally, memory requests from different cores may need to share the same physical link in the NoC, leading to queuing of requests in the link, causing delay. Thus, data allocation needs to take this delay into account when determining the placement. In multi-threaded applications, the execution time is determined by the critical thread and data allocation needs to ensure that the execution time of the individual threads are balanced. Additionally, multi-threading complicates the problem further as data can be shared across multiple threads. The placement of these shared data in appropriate SPM is crucial to performance. Moreover, we argue that the replication of shared data in multiple on-chip SPMs can further reduce the overall execution time. Thus, for multi-threaded applications on

SPM-based many-core architectures, we not only need to decide on-chip versus off-chip allocation, but also the on-chip placement (which SPM) and the replication degree of shared data (how many copies) with minimal contention delay.

Data allocation for SPM-based systems has been studied extensively. Most of these works focus on data allocation for single-core systems running single-threaded applications [4, 14, 37, 47]. Works on SPM-based multi-core systems [37, 43, 50] deal with pipelined or multi-process applications and ignore optimal placement of shared data in SPM. Also, these works do not consider NoC latency in deciding data placement, as they are designed for systems with a bus based interconnect. For example, [43] considers data placement for multi-threaded applications on multi-cores where all the cores are connected to a bus. Hence, every non-local SPM access has exactly the same latency. This uniform remote SPM access latency assumption does not hold good in NoC based systems where the access latency depends on the number of hops between the source and the destination. An optimal placement of data needs to take into account the variable NoC latency. Though replication of data is well studied in Content Distribution Networks [26] and Distributed systems [20] for lower latency and/or fault tolerance, none of the works in the SPM management literature considers the possibility of replication of shared data, that is, judiciously trading in area for performance.

Data management schemes [6, 31, 32] have also been proposed for IBM Cell architecture [13], consisting of a Power Process Element and eight Synergistic Processing Elements (SPE) with 256KB SPM each. In order for an SPE to access data from a remote SPM, it first needs to bring in the data into its local SPM through DMA. In contrast, the architectures we are considering enable direct access of remote SPM data (without DMA). Thus, the SPM management problem for architectures like Epiphany is very different from IBM Cell and opens up new challenges and opportunities.

Optimization of multi-threaded applications on contemporary SPM-based many-core architectures requires compile-time, NoC-aware data placement techniques. To the best of our knowledge, there are no prior works that exploit the unique opportunity offered by these architectures to orchestrate the on-chip data management towards performance and energy benefits of the applications. Given this context, we propose a compile-time, coordinated data management framework called *CDM*, for many-core SPMs. Our main contributions are as follows:

- We formally define the data allocation problem for multi-threaded applications on SPM-based many-cores including the possibility of replication of read-only shared data.
- We propose NoC contention and latency aware compile-time framework to automatically determine the location of data variables (on-chip or off-chip), the replication degree of shared data (how many SPMs), and on-chip placement (which SPMs) so as to minimize the application execution time (i.e., the maximum execution time across all the threads of the application). We design an Integer Linear Programming (ILP) formulation and also an iterative, scalable solution for this optimization problem.
- We implement and evaluate our proposed solutions on the Epiphany architecture with real world applications. The performance-energy improvements are measured from actual execution of these applications on the Parallella hardware platform.

2 RELATED WORK

2.1 Many-core architectures:

Many-cores have been designed and commercially used in both cache-based [33, 42] and Scratchpad Memory based [13, 19] architectures. Xeon Phi Knights Landing [42] is a cache-coherent many-core architecture, employing distributed tag directory-based caches connected by a ring/Mesh interconnect. [33] introduced a non-coherent many-core architecture

called Intel Single-chip Cloud Computer, where each tile consists of 2 cores with private 16KB instruction and data cache, a shared 256KB L2 cache and 16KB Message Passing Buffer, connected in a 2D Mesh.

Apart from cache-based many-cores, a number of works have considered Scratchpad-based many-core architecture due to its extreme scalability and high performance-per-watt efficiency. IBM Cell [13] consist of a Power Process Element and 8 Synergistic Processing Elements (SPE), each containing a 256KB SPM. The address space of SPE is private, so threads can only communicate through main memory. Adapteva Epiphany [19] is an energy-efficient, SPM-based many-core architecture suitable for embedded systems. Epiphany is a tile based architecture containing 16/64 tiles (with support up to a maximum of 4096 tiles), each consisting one RISC core, DMA Engine, Network Interface and 32KB SPM, connected using a 2D Mesh interconnect. This architecture provides shared address space allowing threads to communicate with each other by accessing non-local SPM. Sunway TaihuLight [18] is the world's fastest and most power-efficient supercomputer, which utilizes accelerators containing 8X8 compute processing elements, each containing a local SPM (64KB) to alleviate memory bandwidth bottlenecks in applications.

2.2 SPM Management:

Single-process: SPM allocation has been extensively studied for sequential applications. Earlier works did allocation for *program code* [3, 22, 51], *program data* [4, 14, 37, 47] or *both* [52, 53]. *Program code* allocation needs to ensure that the program flow is unchanged and supports recursive functions while *program data* allocation needs to consider different types of data - *stack* [4, 48], *global* [4, 25, 29, 34, 36, 48] and *heap* [14, 17]. SPM allocation schemes can also be classified as *compile-time* and *run-time* techniques based on the time at which SPM contents are decided.

Compile-time techniques may use profile information to identify frequently accessed data that needs to be placed on SPM. Since data placement is decided before hand, this technique does not incur additional overhead during application execution. Compile-time techniques can further be classified into static allocation and dynamic overlay. In *Static allocation* scheme, the SPM contents are not changed during an application run. Dynamic programming [3] and 0-1 (binary) Integer Linear Programming [53] are commonly used static techniques for selecting data to be placed on the SPM. *Dynamic overlay* based SPM allocation changes SPM contents at pre-determined program points. [49, 52] use liveness analysis and ILP formulation to determine program points at which SPM contents need to be changed so as to minimize the total energy. Though dynamic overlay changes the contents of SPM, it does not incur run-time overhead as these program points are decided during compilation.

In [15, 34, 39], variables that need to be placed on SPM are determined during *run-time*. These mechanisms are especially useful when SPM size is not available during compilation. These works reduce runtime overhead by pre-computing part of the variable allocation during compilation.

Multi-process/Multi-core: A number of works have allocated scratchpad memory in multi-process systems in which sharing scratchpad space and concurrent execution are crucial. [37] partitions the entire scratchpad address between all processes based on the gains obtained when they are run alone. This simple approach may not utilize scratchpad space completely since processes have varying lifetimes. [50] presents a set of strategies for sharing Scratchpad address space between multiple processes for reducing energy consumption. In this work, processes can have disjoint address space (restoring not required), entire address space (need to copy and bring data for every context switch) and a hybrid of both (replacement only for shared address data). Additionally, this work assumes that all processes have equal priority and processes are executed in a round-robin fashion. [43] proposes an integrated task allocation, scheduling and SPM allocation approach for reducing the Worst Case Execution Time. It uses a Task-Graph

based input and formulates the problem using ILP and provides heuristics methods to obtain close to optimal allocation. It considers Virtually Separate SPM and allows tasks to access other SPMs with increased latency.

Mechanisms for SPM only systems: The aforementioned methods assume that scratchpad is present in addition to caches. Therefore, they map data to SPM for reducing energy and access latency for frequent data. However, a number of architectures like Epiphany, TaihuLight and IBM Cell have SPM only. Methods have thus been proposed for managing Stack Data [31], code [32] and heap variables [27] for SPM only IBM Cell architecture.

[31] performs circular management of stack data using DMA at a function (stack frame) granularity. The basic idea is to copy existing stack frames to memory (if no space is left) and copy function stack frames to SPM just before they are called. [31] also provides helper methods to find the global address for a corresponding local SPM address and vice-versa for allowing functions to access parameters passed as pointers. [27] performs heap management by re-implementing malloc function and allocates space for heap variables in local memory if space is available. Otherwise, it copies some of the existing heap variables to main memory before allocating heap variables. It uses a hash table for storing local SPM to global address mapping for getting the correct location of heap variables. [21] proposes a set of primitives that can be incorporated inside the OS. In this technique, application requests for space locally, within the chip, or across chips and obtains the space if available. In contrast, the proposed approach in this work improves application performance by careful allocation of memory objects using compile-time, static analysis and profiling.

Though allocation of stack and global variables on SPM have been proposed before, none of the aforementioned works perform SPM allocation for shared variable across threads in multi-threaded applications on many-core systems. [43] is the only work that allocates shared variables across tasks in an application. However, they do not perform efficient allocation as they assume constant latency for all remote SPM accesses. Additionally, this work assumes at most one on-chip copy of a variable. As stated before, few works have proposed mechanisms for managing stack data [31], code [32] and heap variables [5], [6] for SPM on IBM Cell architecture. However, this architecture provides private address space for processing elements and does not contain NoC connecting different SPMs. Therefore, these works cannot be applied in architectures where threads can access data from remote SPMs.

To the best of our knowledge, ours is the first work to propose an optimal data allocation for multi-threaded applications on SPM based many-core architectures to reduce the overall application execution time, with evaluation on a real platform.

3 MOTIVATING EXAMPLE

We illustrate the importance of judicious data allocation and replication (if necessary) using a simple motivating example. The execution time of a multi-threaded application is determined by the slowest thread. Therefore, we need to place the variables in such a way that the execution time of the slowest thread in the application is minimized. To simplify the illustration, we assume that the execution time incurred due to computations are the same and set to zero in all the threads. Thus, data allocation is the only component that can be exploited to reduce the execution time of this application. For illustration purposes, we use system parameters of the Adapteva Parallella platform as stated in Table 1 in this motivating example.

We choose a multi-threaded kernel containing sixteen threads where all the threads access the global variables *A* and *B*. In addition, each thread accesses a private variable *C*. Figure 2(a) shows the source code of this application while Figure 2(b) summarizes the number of accesses and the access types for each of these variables.

The execution time due to memory accesses can be computed as the sum of (i) access latency of the variables, depending on where the variable is located: local SPM, remote SPM, or off-chip memory (*AccessLatency*), (ii) latency

```
#define N 80
#define ALPHA 2
int A[N]; int B[16];
void thread_func(void *p)
{
  int *tid = (int*) p;
  int C[N];
  for(int i = 0; i < N; i++)
    C[i] = (*tid)*i;
  for(int i = 0; i < N; i++)
    B[*tid] = ALPHA*(A[i] + A[N-i-1]) + C[i];
}
```

(a)

Variable	Acc. Type	Sharers (Threads)	Acc. Per thread
A[80]	R	All	160
B[16]	W	All	1
C[80]	R	Private	160

(b)

Fig. 2. Motivation example: (a) Multi-threaded application source code (b) variables used

spent in bringing data from off-chip to on-chip SPM or vice-versa (*DMA*), (iii) cycles spent in creating multiple copies (*Replication*) and (iv) delay due to contention between memory requests in the NoC and Memory Controller (*ContDelay)* (Refer to Section 4.2.1 for detailed explanation). For achieving optimal performance, we need to allocate the frequently accessed variables in on-chip memory such that the execution time of the slowest thread is reduced. This includes both the stack (variable *C*) as well as the global variables (variables *A* and *B*).

In conventional SPM based many-cores, the variables are allocated in off-chip DRAM by default and the programmer/compiler needs to explicitly bring the data to the on-chip SPM. However, newer Software Development Kits (SDK) like CO-PRocessing THReads (COPRTHR-2) [45] and OpenSHMEM [40] automatically allocate stack variables in local SPMs while global variables are still allocated in DRAM. In this default strategy, stack variable (*C*) is allocated in each core's SPM while the global variables (*A, B*) are allocated in DRAM as shown in Figure 3(a). Every global variable access first utilizes the NoC to reach the Memory Controller (MC) and then reads/writes data from/to off-chip DRAM. Thus, there will be delay due to contention among the memory requests as the threads share the NoC and the MC. Conventional SPM many-core architectures use in-order cores due to power and thermal constraints, where only one memory request can be issued per core at any given time. Hence, delay in the links cannot be caused by two memory accesses issued from the same thread. From Figure 3 (a), we observe that contention delay dominates the total execution time of each thread in this default strategy. This is because, each memory request contends at the MC for obtaining data from off-chip DRAM. Delay due to contending memory requests in the NoC is negligible compared to the delay experienced at the MC as off-chip DRAM access latency is much higher than on-chip hop latency. Moreover, there is very little difference between the execution time of the different threads.

In this example, each threads issues a total of 160+1=161 accesses to global variables (*A, B*). We illustrate further how the execution time for a particular thread, say thread T_{15}, is computed. The thread T_{15} issues a total of 161 off-chip memory requests. Each memory access follows the path $C_{15} \rightarrow C_{14} \rightarrow C_{13} \rightarrow C_{12} \rightarrow C_8 \rightarrow C_4 \rightarrow C_0 \rightarrow MC$. The total access latency would be $161 \times offchipLat = 161 \times 500 = 80,500$ cycles. Contention delay will occur if more than one thread try to access the same link. In the worst case, the delay can be computed as the sum of the requests from the different sources that utilize a link minus the maximum value of request among all the sources. Thus, T_{15} will experience delay in all the links except $C_{15} \rightarrow C_{14}$, as it is utilized by only one thread. For example, maximum delay that can happen in the link $C_{14} \rightarrow C_{13}$ is $(161 + 161) - MAX(161, 161) = 161 \times HopLat = 241.5$ cycles as accesses from T_{15} and T_{14} utilize this link. In total, the delay in the NoC for T_{15} is $5,796$ cycles. Maximum delay that arises due to queuing of requests in MC is $(161 \times 16 - 161 = 2415) \times offchipLat = 1,207,500$ cycles as all the threads share $NoC \rightarrow MC$ link (detailed explanation of queuing delay computation in Section 5.1).

The total access latency for writing to a given variable j, from thread k is $\#Access_{kj} \times (distance_{ki} \times HopLat + AccLat)$ where $Access_{kj}$ denotes the number of times j is accessed by core C_k, $AccLat$ denotes the SPM access latency, and

HopLat denotes the number of cycles per hop in the NoC. C_i denotes the nearest neighbor of C_k with variable j in its SPM ($i = k$ if j is present in the SPM of C_k) and $distance_{ki}$ is the number of hops between C_k and C_i. A remote SPM read access incurs $2 \times distance_{ki}$ latency for the round-trip.

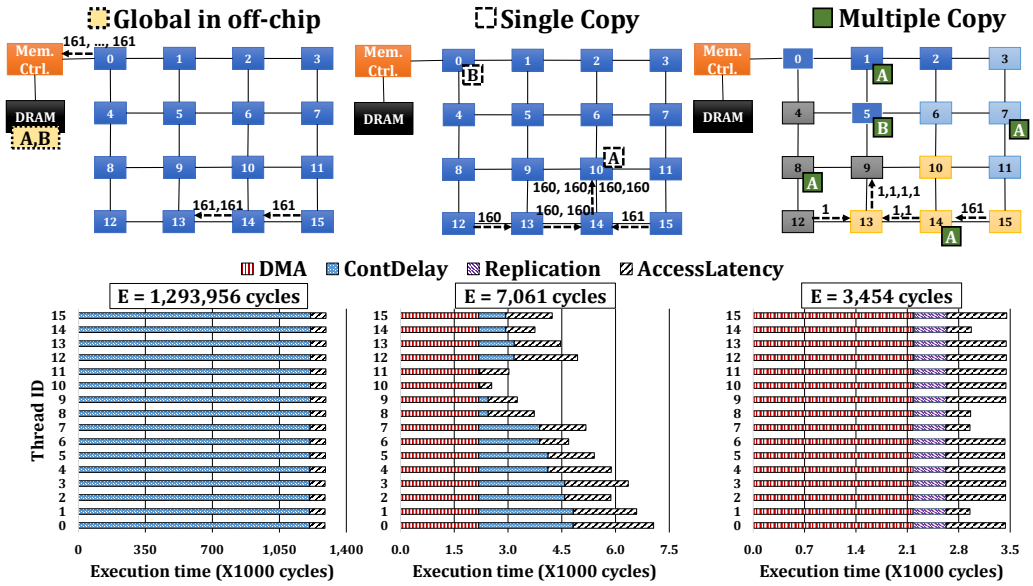


Fig. 3. Optimal data allocation for the motivating example using (a) off-chip memory for global variables, (b) off-chip + on-chip memory with single copy of variables and (c) off-chip + on-chip memory with multiple copy of variables.

It is evident that placing the global variables in off-chip memory leads to high execution time. We first bring in global variables to on-chip memory but put the constraint that a variable can have only a single copy in the entire memory system. The goal is to determine whether to bring the variable on-chip or not and to choose the location of that single copy so as to minimize the total execution time. Figure 3(b) shows the allocation results under this strategy. Shared variables *(*A, B) are brought on chip and allocated in SPM_{10} and SPM_0 respectively as it yields the least execution time. Private variable C is allocated in the local SPM for minimal latency as before. With limited SPM, there will be competition among the variables and only a subset of the variables will be placed on-chip by this strategy. As shown in Figure 3 (b), the total execution time under this strategy including one-time DMA cost is 7,061 cycles. From this figure, we see that the DMA cost is the same across the threads because all the threads need to wait for DMA to complete before the computation starts. Although contention delay at the MC is zero as all the variables are allocated in on-chip SPM, there is still contention in the NoC due to remote SPM accesses to the single copy of each variable. Note that the access latency and contention delay is now different for each thread depending on the distance of its respective core from the single copy of the variable. T_0 is the critical thread in this allocation as it has the highest access latency for variable A and NoC contention delay.

We further optimize performance by strategically replicating read-only shared variables across multiple SPMs. For example, Figure 3 (c) shows that as A is a read-only variable and all the threads access it, it may be replicated to reduce access latency and contention delay. Note that B cannot be replicated as it is a write shared variable and there is no hardware support for coherence among the on-chip copies. For making multiple copies, it is better for one thread to bring the data from off-chip memory through DMA and utilize on-chip network to create multiple copies as (i) the

contention delay arising from queuing of requests in MC and NoC is reduced and (ii) on-chip network is significantly faster than copying directly from off-chip DRAM. However, all the threads that access a variable need to wait till the data is brought in from DRAM through DMA and multiple on-chip copies are created. The overhead of replication is the cost of making multiple copies using NoC.

In this example, (Figure 3 (c)), the rationale behind the partial replication of A (4 versus 16 copies) comes from the cost (DMA plus replication) versus benefit (memory access latency and contention delay) analysis. Note that, apart from determining the number of copies, we also need to determine the placement of these copies. The contention delay reduces in the NoC with multiple copies of the variable; but it cannot be zero as some threads still need to access remote SPM. The total execution time under Multiple copy including one-time DMA cost and replication cost is 3,454 cycles. T_{15} is the critical thread in this allocation as it accesses variable A from remote SPM_{14} and has highest access latency for variable B. Thus, the total execution time is **2.05x** lower in Multiple copy approach compared to the Single copy approach. The current policy of Epiphany compiler is to only allocate the stack and code segments in the SPM and the global data stays in off-chip memory. Compared to this default strategy, Multiple copy has **374.7x** lower execution time.

4 COORDINATED DATA MANAGEMENT

In this section, we state the objectives of the data allocation problem. We then explain the proposed Coordinated Data Management (CDM) framework for allocating multi-threaded application variables in SPM-based many-cores.

Objective: Assume that a multi-threaded application consists of a set of threads T. Let t be the number of threads in this application. Let C be the set of m processing cores in the system and M be the set of memory resources in the system. The system has $m + 1$ memory resources, m on-chip SPMs plus off-chip DRAM with maximum capacity c_1, c_2, …, c_{m+1}, where index $m + 1$ represents the DRAM. Let L denote the set of links in the NoC and the Memory Controller. Thus, t threads is running on m cores ($t \leq m$) with thread T_i assigned to core C_i. Let $MemLat_{ik}$ denote the latency of accessing memory resource M_k by thread T_i. The application accesses n data variables ($v_1, v_2, \ldots v_n$) of size $w_1, w_2, \ldots w_n$ with $access_{ij}$ denoting the number of times T_i accesses variable v_j.

The execution time E_i of a thread T_i can be represented as the sum of computation time ($Ecomp_i$) and total memory accesses time ($Emem_i$) (Equation 1). Application execution time can be represented as the execution time of the critical thread (Equation 2). Our objective in this work is to allocate data variables on the available memory resources such that capacity constraints are satisfied and the application execution time is minimized.

$$E_i = Ecomp_i + Emem_i \tag{1}$$

$$E = MAX(E_1, E_2, \ldots, E_t) \tag{2}$$

CDM framework: The input to the proposed CDM framework is a multi-threaded application source code with marked regions of interest. The SPMs have limited space and require programmer/compiler to explicitly bring in data from off-chip to local memory. In general, the loops in an application may access a number of variables (e.g., arrays) with large sizes. It may not be possible to accommodate the entire array into SPM. Loop tiling is a common technique used to restrict the working set size. Conventionally, polyhedral model is used to perform automatic loop tiling [2, 9, 30]. We recommend the programmers to tile the loops either manually or using any of the aforementioned tools (e.g., PLUTO [9]) for efficient use of SPMs.

The CDM framework consists of the following components as shown in Figure 4:

- *Application Analysis:* Used for obtaining memory profile per-thread for all the data variables in a given application.

- *Data Allocator* - It takes the per-thread memory profile and system configuration as input and obtains replication degree of variables and their placements using two different strategies: (i) an ILP formulation (exact solution) and (ii) **S**ynergistic **N**oC **A**ware **P**lacement, *SNAP*, a scalable algorithm (near-optimal solution).

We now describe these two components in detail.

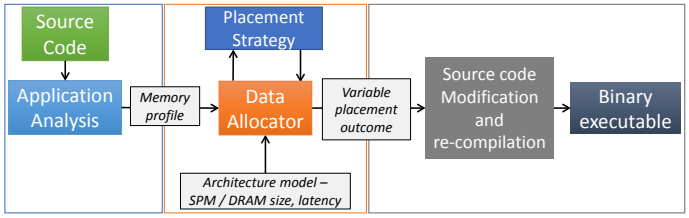


Fig. 4. Workflow of the proposed Coordinated Data Management (CDM) framework.

4.1 Application Analysis

We use static and dynamic analysis to obtain the per-thread memory access profile.

4.1.1 ***Static Analysis:*** We perform static analysis using LLVM Intermediate Representation (IR) to identify the access types of the global variables: Read-Only, Write-Only and Read-Write. We also obtain the start and end address of every global variable and the base address of the stack variables for the dynamic analysis phase.

Replicable variables can be independently brought on-chip by the threads. This is not ideal as (i) off-chip memory latency is much higher than on-chip memory, and (ii) the redundancy leads to contention in the network and off-chip memory. Thus, a coordinated mechanism is proposed for bringing in replicable variables from off-chip to multiple local SPMs. In this mechanism, one of the threads brings in the data from off-chip memory to local SPM and writes it in the other SPMs interested in a copy using the NoC. The number of on-chip copies of replicable variables, called replication degree, is dependent on the on-chip, off-chip network bandwidth and the number of accesses performed by the different threads. The replication degree is obtained using the different variable placement strategies as described in section 4.2.

4.1.2 ***Dynamic Analysis:*** We run and profile the application with representative inputs to obtain per-thread dynamic memory access traces. We use these traces in conjunction with the static analysis to obtain the per-thread Memory Profile. We use representative inputs for obtaining the memory profile. Note that we show in Section 6 that our mechanism provides similar performance improvement for different input sizes/data for the same benchmark even though we rely on profiling.

Array Partitioning: In shared-memory, multi-threaded programming model, the shared array variables are typically declared as global variables. Depending on the parallelization strategy, each thread may access one or more regions of these variables. We identify the array regions accessed by a particular thread (using dynamic memory trace) and partition the arrays into smaller-sized sub-arrays for easier data management.

Loop Tiling: Loop tiling is a natural requirement for SPM-based architectures. A tiled loop is confined to access a smaller portion of a large array at any point in time. In such loops, we only need to allocate the space required for the tile in the SPM instead of the entire array. As the execution moves from one tile to next, the data corresponding to the new tile is brought into on-chip SPM from the off-chip memory. Thus, in case of tiling, we perform profiling and identify the space requirement and accesses for a given tile.

We now generate Memory Profile for each variable v_j in the format: $name$, $size$, $type$, $access_{1j}$, ..., $access_{ij}$, ..., $accesses_{tj}$ where $accesses_{ij}$ denotes the number of accesses of v_j by T_i $\forall i \in [1, t]$.

In the profiling stage, the variable type is statically determined while accesses per variable is obtained from dynamic memory profile. Variation in number of accesses due to input can only change the performance. However, functional correctness cannot be affected as the variable type is obtained using static analysis.

4.2 Data Allocator

In this stage, the memory profile of the application and SPM configuration (size, access latency between cores and memory resources, etc.) are utilized to allocate variables in on-chip or access it directly from off-chip memory, such that the application execution time is minimal. The SPM allocation for multi-threaded application variables with replication can be modeled as an Uncapacitated Facility Location Problem (UFLP), which has been proved to be NP-Complete [28]. We first formulate the SPM allocation problem for multi-threaded applications using architecture specific parameters. Next, we find an exact optimal solution of this allocation problem using Integer Linear Programming (ILP) formulation. We also propose *SNAP*, a scalable algorithm for obtaining feasible solutions in shorter span of time. This is because ILP solvers can take enormous time to obtain an optimal solution for this NP-complete problem.

4.2.1 ***Problem Formulation:*** Let us assume that the application accesses r_n replicable (Read-Only) and nr_n non-replicable variables (Write-Only, Read-Write). Let r_v_j of size $r\_w_j, \forall j \in \{1, \ldots, r\_n\}$ represent the replicable variables and nr_v_j of size $nr\_w_j, \forall j \in \{1, \ldots, nr\_n\}$ represent the non-replicable variables accessed in this application. Let r_access_{ij} denote the number of times replicable variable r_v_j is accessed by thread T_i and nr_access_{ij} denote the number of times non-replicable variable nr_v_j is accessed by the same thread. Section 5.1 specifies how architecture specific parameters and overheads defined in this section are obtained using architecture manuals and micro-benchmarking.

Memory Resource Access Latency: Let $HopLat$ represent the number of cycles spent per hop, i.e., transferring a message packet from one router to another in the NoC. $\forall i \in \{1, \ldots, m\}$, $\forall k \in \{1, \ldots, m+1\}$, let $distance_{ik}$ represent the number of hops for core C_i to reach memory resource M_k and $AccLat_k$ represent the latency to only access the resource. Let $offchipLat$ represent the off-chip memory ($k = m+1$) access latency. The total latency to access M_k from core C_i in a many-core architecture with a 2D Mesh interconnect and XY routing is:

$$MemLat_{ik} = \begin{cases} distance_{ik} \times HopLat + AccLat_k, & \text{if } k \leq m, write \\ 2 \times distance_{ik} \times HopLat + AccLat_k, & \text{if } k \leq m, read \\ offchipLat, & \text{if } k = m+1 \end{cases} \tag{3}$$

For reads, $distance$ is multiplied by two, as it comprises of one request and one response message.

XY is a deterministic dimension-order routing scheme in which packets from source moves along the X-dimension first followed by the Y-dimension until the destination is reached. Thus, every source, destination pair have only one path. This scheme is predominantly used in recent many-core architectures due to its simplicity and deadlock freedom [23]. Note that the proposed mechanism can work with any deterministic dimension-order routing scheme. Handling systems with adaptive routing schemes is left as future work.

Data transfer using on-chip and off-chip network: The DMA cost, $cost_{dma}$ to bring in data from off-chip memory:

$$cost_{dma}(var\_size) = var\_size \times \frac{System_{freq}}{offchip\_r_{dx}} \tag{4}$$

where var_size is the size of the variable, $offchip\_r_{dx}$ (in MB/s) denotes the transfer rate for copying data from off-chip DRAM to SPM, and $System_{freq}$ is the core frequency.

The cost for writing data back to off-chip memory, $cost_{wb}$ using DMA is:

$$cost_{wb}(var\_size) = var\_size \times \frac{System_{freq}}{offchip\_w_{dx}} \tag{5}$$

where $offchip\_w_{dx}$ (in MB/s) denotes the transfer rate for writing data to off-chip DRAM from SPM.

Creating multiple copies of a variable incurs overhead $cost_{copy}$ as the on-chip network is used to transfer data to all the required threads.

$$cost_{copy}(var\_size) = var\_size \times \frac{System_{freq}}{NoC\_w_{dx}} \tag{6}$$

where NoC_w_{dx} (in MB/s) denotes the transfer rate for writing data from one SPM to another.

Memory Access Overhead: If a Read-Write variable is placed on chip, it needs to be brought from off-chip memory to SPM and written back from SPM to off-chip memory. However, for Write-Only and Read-Only variables, data needs to be written/read to/from off-chip memory. The overhead $ovhd_j$ for allocating and bringing variable v_j of size w_j to on-chip memory resource can be defined as:

$$ovhd_j = \begin{cases} copy_{dma}(w_j), & \text{if } j \text{ is R} \\ copy_{wb}(w_j), & \text{if } j \text{ is W} \\ copy_{dma}(w_j) + copy_{wb}(w_j), & \text{if } j \text{ is RW} \end{cases} \tag{7}$$

Threads accessing replicable variable r_v_j of size r_w_j experience an additional overhead of $cost_{copy}(r\_w_j) \times (\#Copies(r\_v_j) - 1)$ when multiple on-chip copies are created.

Allocation decisions: Let x_{ijk} with value 1 denote that thread T_i accesses replicable variable r_v_j from memory M_k. Let $y_{jk} = 1$ mean that replicable variable r_v_j is allocated in memory M_k and d_{jk} with value 1 denote that non-replicable variable nr_v_j is allocated in memory M_k. Let $a_j = 1$ imply that replicable variable r_v_j is allocated on-chip, while $b_j = 1$ imply that non-replicable variable nr_v_j is allocated on-chip.

Access latency per data item: The total latency r_cost_{ij} spent by thread T_i for accessing replicable variable r_v_j is represented using overheads, access latency and decision variables as:

$$r\_cost_{ij} = (ovhd_j \times a_j) + (cost_{copy}(r\_w_j) \times \sum_{k=1}^{m}(y_{jk} - 1)) + \sum_{k=1}^{m+1}(r\_access_{ij} \times MemLat_{ik} \times x_{ijk}) \tag{8}$$

The total latency nr_cost_{ij} incurred by T_i for accessing non-replicable variable nr_v_j can be defined as:

$$nr\_cost_{ij} = (ovhd_j \times b_j) + \sum_{k=1}^{m+1}(nr\_access_{ij} \times MemLat_{ik} \times d_{jk}) \tag{9}$$

Contention delay: A memory access experiences contention in a link present in the NoC (and/or) the memory controller when there are other accesses that are simultaneously trying to utilize the same link. For variables allocated off-chip, requests need to reach the node (S) connecting NoC to the Memory controller. We define $Path(i,k)$ as the set of links utilized by core i to reach destination k using NoC routing protocol. The total number of accesses issued by T_i that utilize a link l (dependent on the variable placement decision), $LAcc_{il}$ can be computed as:

$$\begin{aligned} LAcc_{il} = &\sum_{k=1}^{m}\sum_{j=1}^{n_{nr}} nr\_access_{ij} \times d_{jk} + \sum_{k=1}^{m}\sum_{j=1}^{n_r} r\_access_{ij} \times x_{ijk}, (if\ l \in Path(i,k)) \\ &+ \sum_{j=1}^{n_{nr}} nr\_access_{ij} \times d_{j(m+1)} + \sum_{j=1}^{n_r} r\_access_{ij} \times x_{ij(m+1)}, (if\ l \in Path(i,S)) \end{aligned} \tag{10}$$

Contention at the link between NoC and MC can be computed as:

$$LAcc_{iMC} = \sum_{j=1}^{n_{nr}} nr\_access_{ij} \times d_{j(m+1)} + \sum_{j=1}^{n_r} r\_access_{ij} \times x_{ij(m+1)} \tag{11}$$

When multiple threads access a link at the same time, one of them will be successful. Based on this, the worst-case delay $delay_{il}$ in T_i attributed to the queuing of requests in every link l belonging to the set of links (L) in NoC and MC is computed as:

$$delay_{il} = \begin{cases} HopLat \times (\sum_{p=1}^{t} LAcc_{pl} - MAX(LAcc_{1l}, LAcc_{2l}, \ldots, LAcc_{tl})), & LAcc_{pl} > 0 \\ 0, & \text{Otherwise} \end{cases} \quad (12)$$

$HopLat$ is replaced by $offchipLat$ when contention happens in the link between NoC and Memory Controller.

For example, in Figure 3 (a), threads T_{15} and T_{14} may always contend at link $14 \rightarrow 13$. The worst case delay in this case is $(161 + 161) - MAX(161, 161) = 161 \times HopLat = 241.5$ cycles. Note that Epiphany uses separate meshes for off-chip and on-chip requests.

Total Memory Access Cost: Execution time of a thread T_i due to memory accesses ($Emem_i$) can be computed as:

$$Emem_i = \sum_{j=1}^{nr\_n} nr\_cost_{ij} + \sum_{j=1}^{r\_n} r\_cost_{ij} + \sum_{l \in L} delay_{il} \quad (13)$$

Objective Function and Constraints:

The objective function is represented as:

$$Minimize : MAX(E_1, E_2, \ldots, E_t) \quad (14)$$

where E_i represents the execution time of thread T_i subject to the following constraints:

Capacity constraint:

$$\sum_{j=1}^{nr\_n} nr\_w_j \times d_{jk} + \sum_{j=1}^{r\_n} r\_w_j \times y_{jk} \leq c_k, k \in 1, \ldots, m$$

Non-replicable variables related constraints:

a) Data allocated in only one memory resource:

$$\sum_{k=1}^{m+1} d_{jk} = 1, j \in \{1, ..., nr\_n\}$$

(b) Variable allocated on-chip or off-chip:

$$b_j \geq d_{jk}, \forall k \in \{1, ..., m\}, j \in \{1, ..., nr\_n\}$$

Replicable variables related constraints:

(a) Variable read from one memory resource only by a thread even with multiple copies:

$$\sum_{k=1}^{m+1} x_{ijk} = 1, j \in \{1, ..., r\_n\}, i \in \{1, ..., t\}$$

(b) Identify where variable is allocated:

$$y_{jk} \geq x_{ijk}, \forall k \in \{1, ..., m+1\}, j \in \{1, ..., r\_n\} \text{ and } i \in \{1, ..., t\}$$

(c) Variable allocated on-chip or off-chip:

$$a_j \geq y_{jk}, \forall k \in \{1, ..., m\}, j \in \{1, ..., r\_n\}$$

Binary constraints:

$$d_{jk} \in [0, 1], \forall j\{1, ..., nr\_n\}, k \in \{1, ..., m+1\}, b_j \in [0, 1], \forall j \in \{1, ..., nr\_n\}$$

$$x_{ijk} \in [0, 1], \forall i \in \{1, ..., t\}, j \in \{1, ..., r\_n\} \text{ and } k \in \{1, ..., m+1\}$$

$$y_{jk} \in [0, 1], \forall j \in \{1, ..., r\_n\}, k \in \{1, ..., m+1\}, a_j \in [0, 1], \forall j \in \{1, ..., r\_n\}$$

We obtain the exact solution for this problem through Integer Linear Programming (ILP) formulation.

4.2.2 ***Synergistic NoC Aware Placement (SNAP):.*** Computing the exact solution of the NP-complete variable placement problem by implementing it using ILP does not scale with number of threads and variables (refer Table 5 for data allocator run-time). Therefore, an iterative strategy: **S**ynergistic **N**oC **A**ware **P**lacement (*SNAP*) is proposed in this work to determine the placement (leave it on DRAM or bring it on-chip), replication degree of variables and location from which each thread accesses a variable (in case of multiple copies) for improving multi-threaded application performance. Every thread has one opportunity to allocate variables in an SPM closer to it. If there are n variables and t threads, there will be a total of $n * t$ iterations.

All the global variables are first allocated in DRAM and access latency (per variable) and contention delay is calculated per application thread. Next, for each thread, the (execution time, thread id) pair is computed and added to a vector $ExecTimePQ$, while indices of all variables accessed by each thread is added to $remVar$. $ExecTimePQ$, $remVar$, memory profiling results from application analysis and system parameters are then supplied to *SNAP* algorithm as input.

The execution time of a multi-threaded application is determined by the slowest thread (Equation 2). At every iteration in *SNAP* (Algorithm 1, Lines 1 - 29), we first identify the critical thread (note that the critical thread may change from one iteration to next as we allocate variables to SPMs). Then, we try to improve its performance by reducing the latency of the variable that has the maximum access latency density (Line 6), by moving it to a location closer to the critical thread. The access latency density is computed as (access latency + contention delay) / (variable size) using Equations 3, 12.

We find the memory location that can accommodate the maximum access latency density variable and yields the least thread execution time for the critical thread by trying all SPMs with increasing NoC hop latency (Lines 8 - 25). In each iteration of this loop, we find all SPMs that are *rad* hops away from the critical thread and have sufficient space to hold the variable. If there is no SPM that has sufficient space, we increase *rad* by one and try again. For each location found in this step, we invoke allocateVar procedure to (i) update DMA and replication cost for each thread that accesses this variable (Equations 4, 6) (ii) place the variable in that location (iii) identify the location from which each thread will access this variable from and (iv) update the contention delay and application execution time (Equations 2, 12).

In this algorithm, we accept an allocation only if the execution time monotonically decreases (Line 19) at every iteration. This condition is essential as the overall application execution time may increase (due to change in access latency and contention delay in other threads), although the critical thread's execution time reduces. However, when the other threads have similar execution time as the critical thread and the same replicable variable (*vid*) as the highest density variable, the new application execution time will be higher as replication cost is added to all the accessing threads. If this decision is not allowed, the best solution can be worse than optimal. Therefore, we look ahead and try to replicate *vid* until it is the highest latency density variable in subsequent critical threads. We accept all these allocation decisions only if the overall application execution time reduces (Line 23).

Recall that the total number of steps in *SNAP* strategy is $n * t$ in the worst case. We terminate early if the critical thread has (i) no more variables to allocate and (ii) zero contention delay (Line 5). However, we allow other threads to look into the unconsidered variables when contention delay is non-zero. This is because when threads allocate variables in other SPMs, the critical thread's contention delay may reduce.

5 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of our proposed Coordinated Data Management framework on the SPM-based Epiphany many-core architecture.

Algorithm 1: Synergistic NoC Aware Placement (SNAP) algorithm

Input : W - set containing size of each variable, m - #on-chip SPM, *Alloc* - variable allocation result, $Alloc[j][k] = 1 \implies$ variable j is allocated in memory resource M_k, 0 otherwise, ExecTimeVec- vector containing (execution time, thread id) pair, remVar- list of variables accessed by each thread

Output : *Alloc*

```
1  while ExecTimeVec ≠ ∅ do
2      tid ← ExecTimeVec [0].second;
3      Erase ExecTimeVec [0];
4      if remVar [tid] == ∅ and contDelay[tid] != 0 then continue ;
5      else if remVar [tid] == ∅ then break ;
6      vid ← getMaxLatDensityVar(tid);
7      Loc_curr ← currVarLoc [vid][tid];
8      for rad ← 0 to MAXRAD do
9          dest_ids = getDest(tid, rad, W[vid]);
10         if dest_ids == ∅ then continue ;
11         Loc_new ← −1; E_new ← E;
12         foreach dest in dest_ids do
13             E_temp ← allocateVar(E, vid, dest);
14             if E_new[tid] < E_temp[tid] then
15                 Loc_new ← dest;
16                 E_new ← E_temp;
17         end
18         if Loc_new ≠ −1 then
19             if getMax(E_new) < getMax(E) then
20                 Update best solution, Alloc, system state and E;
21             else if vid is replicable and getMax(E_new) − repl_cost > E[tid] then
22                 Look ahead and allocate until vid is the highest latency density variable in
                     subsequent critical threads;
23                 Update best solution, Alloc, system state and E if application execution time decreases;
24         break;
25     end
26     Update execution time of threads in ExecTimeVec as per E;
27     ExecTimeVec.sort();
28 end
29 return Alloc;
```

5.1 Epiphany platform

Figure 1 illustrates an abstracted Adapteva Parallella platform used in our evaluation and Table 1 summarizes its specifications. The Adapteva Parallela platform is designed for developing parallel processing applications using the on-board Epiphany chip. The 16-core Epiphany SoC consists of an array of simple RISC processors (eCores) programmable in C connected together in a 2D-mesh NoC and supporting a single shared address space. The Epiphany SoC acts as an accelerator and is supported by a Xilinx Zynq SoC on the same development board. The Zynq SoC contains dual-core ARM Cortex-A9 processors, Memory Controller and eLink (implemented in Field Programmable Gate Arrays) for connecting Zynq SoC and Epiphany. The ARM processor can launch multi-threaded applications on Epiphany.

Memory Architecture: Each eCore contains a unified 32KB SPM for both program instructions and data. As SPM is more energy-efficient than caches, Epiphany does not provide cache memory at any level of the memory hierarchy. Apart from accessing the local SPM, an eCore can also access any remote SPM through the mesh network at a latency proportional to the number of hops between the source and the destination core. The eCores can also access 1GB shared off-chip memory (SDRAM) with high latency. We estimate the local SPM ($AccLat_k$, $\forall k \in [1, m]$) and off-chip DRAM (*offChipLat*) access latency to be 1 and 500 cycles, respectively (eCores running at 600 MHz) by executing micro-benchmarks. The off-chip access latency is high as Epiphany accesses SDRAM through the Zynq SoC. The Epiphany supports a 32-bit shared memory address map, where each eCore is assigned a unique address space. The first 12 bits of the address space indicate the row and the column index of the eCore (12-bits can support up to 4,096 cores) while the remaining 20 bits specify the exact location within the corresponding SPM. As the address space is shared, an eCore can access any SPM location.

Network-on-Chip: Epiphany architecture is supported by a 2D-mesh Network-on-Chip (eMesh). The eMesh NoC consists of three distinct and orthogonal channels, *cMesh* for on-chip writes, *xMesh* for off-chip write transactions, and *rMesh* for all read requests. The on-chip and off-chip write channels have data transfer rates of 8 and 1 byte per cycle, respectively, while reads are issued once per 8 cycles. The data transfer rates are higher for writes as Epiphany is generally used for Message Passing applications where communication latency is crucial for performance. As mentioned previously, the mesh interconnect allows an eCore to access non-local SPMs (known as Remote SPMs) with varying latency using the row and column index of the remote SPM. Remote SPM accesses take a deterministic path in the network using XY routing. In XY routing, an access moves along the row-axis first and then along the column-axis. Each router hop (*HopLat* used in Equation 3) takes 1.5 cycles as mentioned in the Epiphany reference manual [1].

Programming model: A number of Software Development Kits (SDK) like eSDK, CO-PRocessing THReads (COPRTHR-2) [45] and OpenSHMEM [40] are available for programming on Epiphany. The COPRTHR-2 library provides API for POSIX thread (*pthread*) like programming model and DMA transfer between the on-chip and the off-chip memory. It also automatically allocates stack variables and instructions in on-chip memory. The OpenSHMEM library is utilized for efficient inter-core data transfers.

Direct Memory Access (DMA): Each eCore has a DMA engine for transferring data between on-chip SPM and off-chip DRAM. We use micro-benchmarks to obtain the on-chip and off-chip memory data transfer rate. The read ($cost_{dma}$ in Equation 4) and write ($cost_{wb}$ in Equation 5) data transfer rates between off-chip DRAM and local SPM are measured to be 87.71 MB/s and 234.35 MB/s. The read and write ($cost_{copy}$ used in Equation 6) data transfer rate between the farthest on-chip SPMs are measured to be 392.00 MB/s and 1236.81 MB/s, respectively. The read data transfer rate is lower than the write data transfer rate as the number of the bytes transferred per cycle in the read mesh is lower than the write mesh.

Table 1. Specifications of Parallella platform with Epiphany

Cores	2 ARMv7 host cores 16 Epiphany in-order (dual-issue) cores, 600 MHz
SPM	Unified I & D, 32KB, 4 banks, 1-cycle access latency
Network	2D Mesh, 1.5 cycle per hop latency, XY routing
Memory	1GB, 500-cycle access latency
DMA data transfer rate	on-chip: write 1236.81 MB/s, read 392 MB/s off-chip: write 234.35 MB/s, read 87.71 MB/s

5.2 Experimental Setup

5.2.1 ***Benchmark Application Kernels.*** The characteristics of the multi-threaded benchmark application kernels used in our experimental evaluation are shown in Table 2. The applications from the prevalent multi-threaded benchmark suites e.g. *PARSEC*[8] (primarily designed for the high-performance computing domain) cannot be compiled directly on

the Epiphany architecture due to lack of support for libraries (e.g., Standard Template Library used in these applications). Hence, we choose a set of representative application kernels, such as *1DFFT, 2DCONV, ATAX, GEMM, GESUMMV*, from the embedded, multi-threaded benchmark suites *Rodinia*[12] and *Polybench/C*[38]). We also include *AESD* and *AESE*, which are commonly used kernels for decryption and encryption of data, respectively. Additionally, we choose three kernels: *PHY_ACI*, *PHY_DEMAP* and *PHY_MICF* from the Long Term Evolution (LTE) Uplink Receiver PHY benchmark [41]. The PHY benchmark implements baseband processing in mobile base stations. There has been increasing interest in mapping baseband processing to many-core architectures instead of conventional ASIC- or DSP-based designs for improved programmability and flexibility. In particular, a number of existing works [44], [11] have explored mapping of the PHY benchmark on the Epiphany architecture. However, none of the previous works have considered SPM management on the Epiphany architecture.

5.2.2 ***Porting of Kernels on Epiphany.*** We evaluate our SPM-management approach on Adapteva Parallella platform with on-board Epiphany architecture as presented in Section 5.1. Table 1 summarizes the system configuration.

Few kernels (e.g. PHY_MICF, PHY_ACI), in Table 2 offer pthread-based multi-threaded versions. For the remaining benchmark kernels that are available in OpenCL/OpenMP versions, we manually port them for Epiphany. The kernels *PHY_ACI* and *PHY_DEMAP* use twelve threads, while the remaining kernels all utilize sixteen threads. We implemented these kernels using COPRTHR-2 and OPENSHMEM libraries (described previously in Section 5.1). We manually determine the optimal thread to core mapping so that the threads with higher levels of data sharing are mapped to neighboring cores on chip. We pin the threads to the appropriate cores according to this mapping.

We also perform loop tiling (loop blocking) for the benchmarks where the total data set size exceeds the on-chip SPM capacity. We carefully select the tile size such that the entire working set corresponding to a tile can be accommodated on-chip. The data corresponding to a tile is brought into and out of the SPM through DMA operations in the beginning and the end of the execution of the tile, respectively. Thus, there are no off-chip memory accesses during the execution of each iteration of the tiled code. Conventional SPM-allocation approaches that are agnostic to the exact placement and replication of the data cannot optimize this tiled code any further. However, starting with the tiled code, our coordinated data management framework can significantly improve both the performance and the energy consumption by carefully controlling the placement and the replication of the shared variables.

Table 2 shows that per-thread code plus stack size of the kernels varies from 5.69 KB to 12.24 KB. As mentioned in Section 5.1, the code and the stack are automatically allocated in on-chip SPM using COPRTHR2 [45] library. Hence, we only consider the allocation of global data in this work. The global data size ranges from 20.05 to 12,288 KB, clearly exceeding the on-chip SPM capacity (32KB per SPM x 16 = 512KB) and necessitating loop tiling. Note that 32KB SPM space per core needs to accommodate the code, global data, heap, and stack segments. We reserve space for code and stack variables as per kernel requirements in Table 2 and utilize the remaining space for global data. Thus, the global working set size of our tiled code ranging from 20 to 257 KB can be easily accommodated in on-chip SPMs. None of the benchmark kernels uses heap; the reserved SPM space for the heap can be managed using existing approaches [14], [6].

We statically analyze the tiled application programs and profile their execution with representative inputs to obtain the memory access traces as explained in Section 4.1. Given the memory access profile, the CDM framework generates the replication degree and the placement of the global data variables. We use the python library of *Gurobi* optimizer version 6.5.2 to solve the ILP formulations.

A POSIX-thread like shared memory program is utilized in this work, i.e., each thread executes the same function. In Epiphany architecture, every thread needs to access data either directly from off-chip DRAM or after explicitly

Table 2. Characteristics of the application kernels. *PHY_ACI* and *PHY_DEMAP* have 12 threads while others have 16 threads.

Kernel	Input	Code+Stack (KB)	Global Data (KB)	
			Original	Tiled
1DFFT	1024	8.57	20	20
2DCONV	1024 X 1024	8.73	8,192	128
AESD	256 KB	9.67	513	257
AESE	256 KB	9.78	513	257
ATAX	1024, 1024 X 1024	9.16	4,108	140
GEMM	1024 X 1024	8.55	12,288	132
GESUMMV	1024, 1024 X 1024	9.18	8,204	140
PHY_ACI	4Antenna, 2Layers, 64QAM, 100 RB	10.72	182	182
PHY_DEMAP	4Antenna, 2Layers, 64QAM, 100 RB	5.69	450	150
PHY_MICF	4Antenna, 4Layers, 64QAM, 100 RB	12.24	117	117

copying it on SPMs. SHMEM library provides *shmem_malloc* function which is executed in all the threads and contain the same local address. Similarly, local SPM addresses can be converted to global address using library function *e_get_global_address* by passing the local address and location of the target SPM id. SHMEM/COPTHR-2 library also provides blocking and non-blocking DMA copy. We bring in data for the next iteration while current iteration is being executed by using non-blocking DMA calls. A coordinated mechanism is proposed for replication as the off-chip memory data transfer rate is much lower than the on-chip NoC memory data transfer rate. Thus, we wait for the first copy to be brought on to the chip before multiple copies are created using NoC. SHMEM library also provides efficiently implemented broadcast and multicast functions. These library functions are used for replicating variables.

Since these library functions provide layers of abstraction, we only change variable placements using SNAP-S and SNAP-M results. Thus, the API calls for data transfer between off-chip memory and SPM, and replication are suitably modified for the different strategies, re-compiled and executed on the Parallela board to measure the execution time and energy consumption. We use the timer function provided by the Parallela platform for execution time.

5.2.3 ***Energy measurement.*** The Epiphany co-processor does not have sensors for measuring the chip power. Therefore, we measure the average power consumption of the entire Parallela board using *ODROID Smart Power* and compute the energy consumption as the product of the average power and the execution time.

5.2.4 ***Evaluation Mechanisms.*** To the best of our knowledge, there are no existing works that perform SPM allocation for multi-threaded applications on many-core systems for improving the application execution time. Therefore, we devise a *GREEDY* strategy as our baseline in which variables are sorted in descending order of access densities (total accesses/size) and allocated in the SPM of the highest accessing thread. Variables are allocated in DRAM if there are is no space in any of the accessors.

To measure the importance of NoC placement and contention delay while allocating variables, we consider two strategies. In the first strategy, *ILP-S*, we obtain the exact data placement using ILP in multi-threaded applications with single copy of each variable that yields the least overall execution time. In the second strategy, *SNAP-S*, we evaluate how the proposed *SNAP* allocation strategy reduces the execution time of the critical thread by placing variables appropriately with single copy of variables. At every step in *SNAP-S* and *SNAP-M*, the critical thread is obtained based on the current location of variables and the access counts obtained from one-off dynamic profiling using representative inputs using Equation 13.

Next, we evaluate the importance of replication in addition to NoC placement and contention through two strategies: *ILP-M* and *SNAP-M*. *ILP-M* finds an exact solution for the data allocation problem with multiple copies of replicable variables (when required). *SNAP-M* on the other hand, finds the allocation based on the proposed *SNAP* strategy with replication of variables.

The application source code is modified based on the outcome obtained from the above mechanisms to allocate and place variables in SPM or off-chip memory.

5.3 Memory Access Profile Characterization

We present the memory access characteristics of the kernels obtained from profiling.

5.3.1 ***Distribution of stack and global variables:*** As can be observed from Table 2, the code plus stack size is much smaller than the global data size. In particular, the global data occupies 96.78% (on average) of the global data and stack space. Recall that our benchmarks do not use heap segment. Figure 5 shows the distribution of the global and stack accesses. Clearly, a significant fraction of the memory accesses are to global data except for *PHY_DEMAP*.

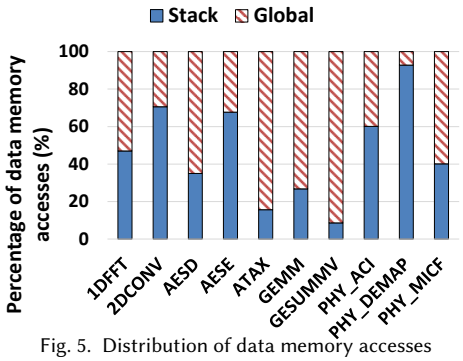


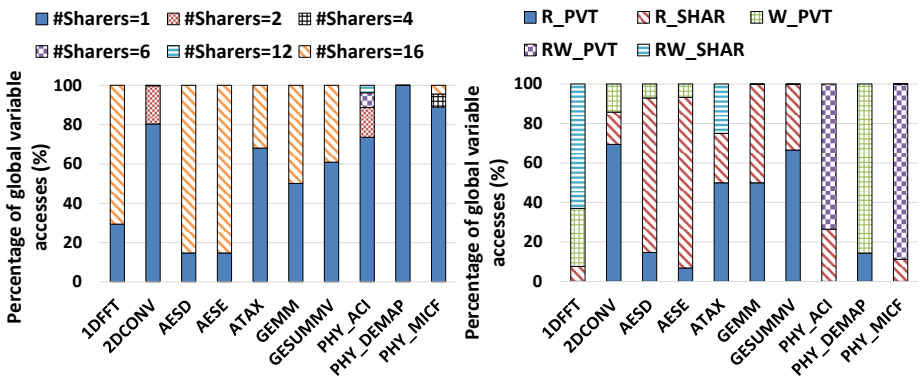
Fig. 5. Distribution of data memory accesses

5.3.2 ***Sharing degree:*** We show the sharing degree of the global data memory accesses in Figure 6 (a). The Y-axis shows the distribution of global data accesses to variables with different sharing degree: 1, 2, 4, 6, 12, and 16. For example, the orange part of the bar graph show the percentage of global data memory accesses to variables with 16 sharers. Note that this figure excludes the private stack accesses. From this figure, we identify that *PHY_DEMAP* is completely data parallel with no sharing as all the accesses are to variables with sharing degree 1. For *2DCONV*, we observe that the maximum sharing degree across all variables is 2. This is because each thread shares the first and last row of the input matrix with the previous and following thread respectively. *PHY_MICF* has variables shared between 4 and 16 threads. *PHY_ACI* has maximum sharing degree of 12 as it has only 12 threads. For the remaining kernels, global variables are either accessed only by one thread or by all 16 threads.

5.3.3 ***Distribution of access types:*** Figure 6 (b) shows the distribution of global data memory access types, where R means Read only, W means Write only, and RW means Read-Write accesses. Apart from accessing private variables belonging to the above types (R_PVT, W_PVT and RW_PVT), many of the kernels have significant Read-Only shared (R_SHAR) variables. These variables are ideal candidates for replication. Similarly, the number of accesses to Read-Write shared variables (RW_SHAR) is close to zero in most of the kernels.

5.4 Global Data Allocation Results

Figure 7 shows the outcome of global data placement and replication with different data placement strategies: *GREEDY*, *ILP-S*, *SNAP-S* and *ILP-M*, *SNAP-M*. Note that the tiled version of the kernels can already accommodate the entire working global data set in on-chip SPM and there are no off-chip accesses. For most benchmarks, a portion of the global data memory accesses go to remote SPMs using *ILP-S* and *SNAP-S* strategies. This is because *ILP-S* and *SNAP-S* allow only one copy of a global variable even if it is shared across multiple threads. Most of these remote SPM accesses



(a) Sharing degrees (b) Access types

Fig. 6. Distribution of global data memory accesses under different parameters

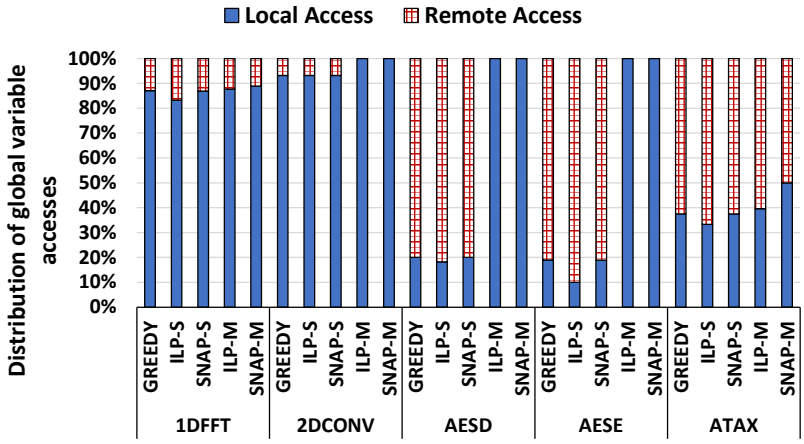
encountered in *ILP-S* and *SNAP-S* can be converted to local SPM accesses by using replication mechanism in *ILP-M* and *SNAP-M* respectively. However, firstly it is not possible to achieve 100% local SPM accesses in kernels that have read-write shared variables as reported in Figure 6 (b), for example: *1DFFT* and *ATAX*. Secondly, in some cases, even when there are Read-only shared variables among threads, it might not be possible to perform complete replication as on-chip SPM space is limited. For example, in *PHY_ACI*, each thread accesses a total of 32.8 KB data belonging to either private RW or shared R access types. However, since each SPM only has 32KB for instructions, stack and data, it is not possible to replicate all the read-only variables and achieve 100% local SPM accesses. Finally, some applications may not perform full replication due to cost (DMA and replication copy latency) versus benefit (memory access latency, contention delay) analysis. For example, in *PHY_MICF*, the optimal strategy *ILP-M* does not perform full replication and some variables are still accessed from remote SPMs.

Space utilization: Table 3 shows the amount of space allocated across all SPMs for global data under the different allocation strategies. As each kernels have different working set sizes, the space allocation varies. Also, *ILP-M* and *SNAP-M* utilizes the available SPM space more than *ILP-S* and *SNAP-S* respectively. This is because *ILP-M* and *SNAP-M* strategies employs replication of shared variables to reduce the memory access latency and NoC queuing delay using the extra SPM space available on-chip.

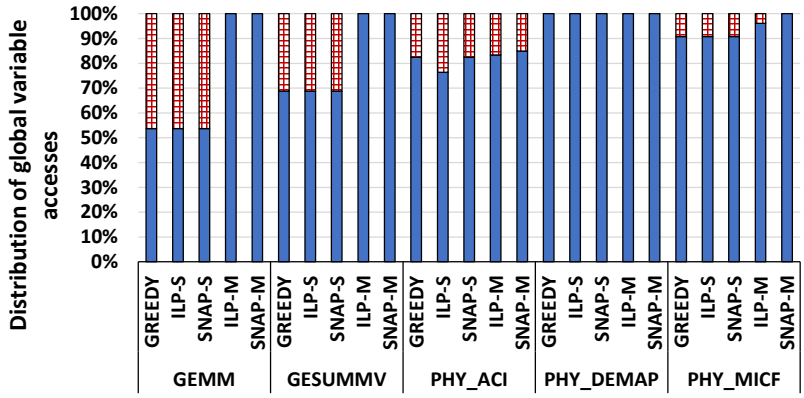
Table 3. Total on-chip SPM space (in KB) allocated across all cores for global data using different strategies

	1DFFT	2DCONV	AESD	AESE	ATAX	GEMM	GESUMMV	PHY_ACI	PHY_DEMAP	PHY_MICF
GREEDY	20	128	257	257	140	132	140	182	150	117
ILP-S	20	128	257	257	140	132	140	182	150	117
SNAP-S	20	128	257	257	140	132	140	182	150	117
ILP-M	26	143	278	274	148	192	200	220	150	225
SNAP-M	34	143	278	274	164	192	200	225	150	300

Replication degree: Figure 7, also captures the replication degree using the number of local and remote accesses. In *PHY_ACI*, even though every shared variable is Read-only, full replication is not performed as sufficient on-chip space is not available to accommodation all variables. Hence, only crucial variables to performance are replicated with a degree of 1, 2 or 3. In *ATAX*, only 1 copy of Read-Write variables is present while 7 copies of shared Read variables exist. In *1DFFT*, shared variables have a replication degree of either 4 or 5 based on cost versus benefit analysis, while single copy of Read-write variables is present. *PHY_DEMAP* does not do any replication as there are no shared variables (Figure 6). From Figure 7, we find that *2DCONV, AESD, AESE, GEMM, GESUMMV and PHY_MICF* have zero remote accesses. These benchmarks have sufficient space to accommodate all shared variables (Table 3) and hence are able to replicate them



(a) Benchmarks 1 - 5



(b) Benchmarks 6- 10

Fig. 7. Data allocation results when using different strategies

in all accessing threads. In order to demonstrate the effectiveness of *SNAP-M* towards partial replication, we restrict the available SPM space to 16KB and obtain the replication degree of all benchmarks (Table 4). In this experiment, the replication degree of shared variables in *AESE, AESD* reduces to 2, while *PHY_MICF* reduces to 1, 2 or 6.

Table 4. Replication degree in SNAP-M when each SPM has 16KB space

	1DFFT	2DCONV	AESD	AESE	ATAX	GEMM	GESUMMV	PHY_ACI	PHY_DEMAP	PHY_MICF
Repl. Degree	1,4,5	2	2	2	1,7	16	16	1,2,3	N.A.	1,2,6

Run-time of Data allocator: Table 5 summarizes the run-time of the data allocator for obtaining variable placement decisions when using the different strategies. From this table, we observe that for some application kernels, ILP based solutions *ILP-S* and *ILP-M* take substantial run-time and do not produce the optimal allocation result even after days as many combinations need to be explored in the contention component of the allocation problem for obtaining the optimal placement of variables. Hence, pruning allocation paths is challenging. However, the execution time of *GREEDY*, *SNAP-S* and *SNAP-M* are much lesser as they have polynomial run-time complexity.

Table 5. Run-time (in sec) for obtaining data allocation results using different strategies. Allocation results are obtained with a timeout of 1 hour, for strategies that do not terminate (indicated using *)

	1DFFT	2DCONV	AESD	AESE	ATAX	GEMM	GESUMMV	PHY_ACI	PHY_DEMAP	PHY_MICF
GREEDY	0.0003	0.0005	0.0003	0.0002	0.0001	0.0001	0.0002	0.0001	0.0002	0.0002
ILP-S	*	0.2563	*	*	*	*	*	*	0.0708	*
SNAP-S	0.0211	0.0138	0.0124	0.0102	0.0060	0.0071	0.0126	0.0109	0.0018	0.0084
ILP-M	*	0.0122	2.2361	0.1463	*	0.1957	0.1628	*	0.0735	4.0351
SNAP-M	0.0218	0.0124	0.0114	0.0077	0.0062	0.0026	0.0057	0.0058	0.0120	0.0082

5.5 Performance and Energy Improvement

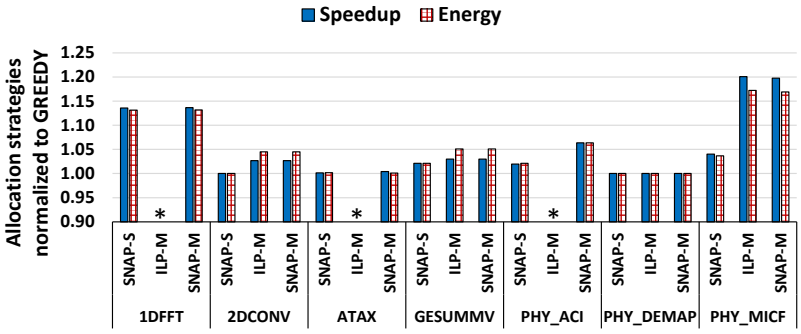
We now compare the performance and energy behavior of the different allocation strategies as explained in Section 5.2.4. To evaluate the effectiveness of NoC latency/contention aware placement and replicating shared variables, we use *GREEDY* as the baseline to compare with the proposed allocation mechanisms. The reduction in execution time is attributed to the distribution of accesses across threads, the type of accesses performed on a given variable and the sharing degree of the variables. We do not present the results for *ILP-S* and *1DFFT, ATAX, PHY_ACI* utilizing *ILP-M* strategy as the ILP solver does not produce the optimal solution event after **1 day**.

As seen in Figure 8, the proposed *SNAP-M* approach provides an average speedup of **1.84x** and energy reduction of **1.83x** when compared to the *GREEDY* strategy. Specifically, the kernels *AESD, AESE* and *GEMM* (Figure 8 (b)) can achieve higher performance as they contain shared variables that are heavily accessed by all the threads. *1DFFT* has 1.14x improvement with *SNAP-M* when compared to *GREEDY* as it has fewer accesses to Read-only shared variables. In *1DFFT*, accesses to any read-write shared variable is dominated by one thread and allocated to its private SPM. Therefore, frequent accesses are served locally, while other threads contribute to infrequent remote accesses. *2DCONV* has very little sharing as only the first and last row of the input matrix is shared. The improvement in execution time for *PHY_ACI, PHY_MICF* are similar to the percentage of accesses to the replicable variables, while *PHY_DEMAP* has no variables to replicate. The improvement in *ATAX, GESUMMV* are not significant as the performance bottleneck relies on computations and other variables in the kernel. From figure 8, we also observe that the speedup and energy reduction in *SNAP-M* is similar to the optimal solution obtained from *ILP-M*. *SNAP-M* also obtains a higher speedup than *SNAP-S* in all benchmarks, using replication when necessary. Note that *SNAP-M* does not fully replicate Read-only variables in *1DFFT* as the benefit is lesser than the cost. Therefore, creating multiple copies is crucial for improving application performance in SPM-based many-cores as it reduces memory access latency and contention delay.

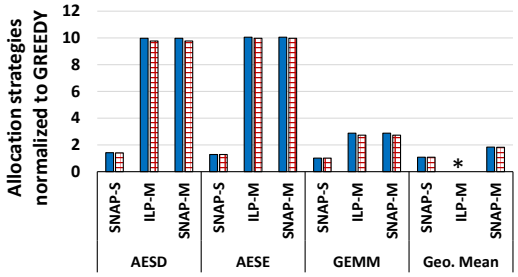
From Figure 8, it is seen that *SNAP-S* provides an average speedup and energy reduction of **1.09x** when compared to the *GREEDY* strategy. *GREEDY* allocates variables in the thread that accesses it the most. This strategy works well for private variables and shared variables that are predominantly accessed by one thread. Therefore, *SNAP-S* can only improve performance by placing shared variables that have similar accesses from all sharers. The speedup in *AESD, AESE, 1DFFT, PHY_MICF, PHY_ACI* show the importance of considering NoC latency, contention delay and improving the critical thread when determining the placement of variables. Kernels *2DCONV, PHY_DEMAP* have a speedup of 1 as the placement decisions are similar in both *SNAP-S* and *GREEDY*. *GESUMMV, ATAX* cannot be improved much as private variables and computations are crucial for performance.

Thus, we observe that the proposed *SNAP-S* and *SNAP-M* mechanisms are effective in reducing the execution time and energy of the evaluated kernels.

In SPM based many-core architectures like Epiphany, the current policy of the compiler is to only allocate the stack/local variables and code segments in the SPM. The global data stays in off-chip memory. Note that even GREEDY does not currently exist in Epiphany like architectures. Allocating selected global data to on-chip memory with appropriate replication degree is the contribution of this work. We evaluate the end-to-end performance of a multi-threaded application (includes overheads in bringing data on-chip, creating multiple copies and accessing it from remote



(a) Benchmarks 1 - 7



(b) Benchmarks 8- 10

Fig. 8. Execution time and energy in different allocation strategies normalized to GREEDY.

locations) under SNAP-M and the default allocation strategy (global variables are left off-chip). Table 6 summarizes the speedup of SNAP-M with respect to default strategy. From this Table, we find that SNAP-M has an average improvement of 22.9x when compared to the default allocation strategy.

Table 6. Speedup of SNAP-M w.r.t. default allocation in off-chip DRAM

1DFFT	2DCONV	AESD	AESE	ATAX	GEMM	GESUMMV	PHY_ACI	PHY_DEMAP	PHY_MICF
24.0	8.3	43.9	44.1	5.6	28.4	7.8	16.6	5.9	44.4

5.6 NoC latency and contention delay

It is not possible to isolate and measure NoC latency and contention delay on our platform as it is coupled with computations and memory accesses. Thus, to show these effects, we allocate all the local variables in AESE benchmark in local SPM and do a design space exploration on locations for two shared variables, assuming that there is single copy of each variable. Each variable can be allocated in 16 possible on-chip locations. For two variables, we have a total of 16x16 = 256 possibilities. The distribution of execution time in shown in Figure 9. From this figure, we find that the speedup of the optimal placement with respect to the placement with the highest execution time is 1.32. This shows that variable placement is crucial for improving application performance. From Figure 9, we find that placing both variables in the same locations leads to the highest execution time primarily because of contention delay. *GREEDY* belongs to this category as both variables were allocated in SPM 0. We also find that the difference in execution time between SNAP-S and optimal allocation is close to each other.

5.7 Scalability of the proposed solution

From Section 5.4, we observed that the ILP solver does not produce the allocation outcome even after 1 day. In this section, we show how the proposed *SNAP* algorithm scales with increasing number of threads. We utilize the kernels

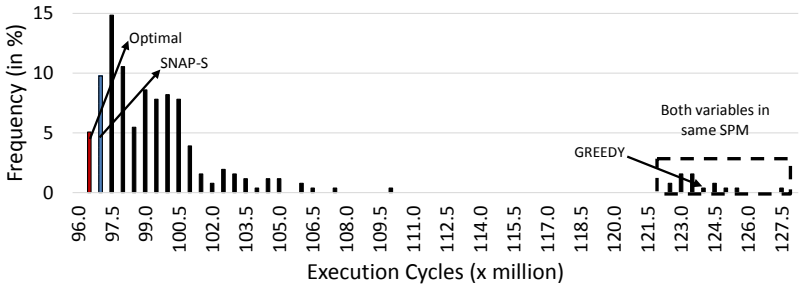


Fig. 9. Variable placement design space exploration

that can be extended to 256 threads and scale the memory profile accordingly. Next, we allocate data variables using *SNAP-M*. Note that we only have 16 cores in the Parallella platform. Table 7 states the *SNAP-M* run-time for different kernels. From this table, it is seen that run-time for obtaining solutions is less than 10 seconds for all these kernels.

Table 7. *SNAP* allocation strategy run-time (in sec) for 64/256 threaded application in a 64/256 core system

Benchmark	2DCONV	AESD	AESE	ATAX	GEMM	GESUMMV
64-threads	0.24	0.19	0.15	0.24	0.10	0.23
256-threads	7.01	4.54	3.54	7.85	2.67	7.37

6 DISCUSSION

Different input sizes: In the profiling stage, the variable type is statically determined through compiler while accesses per variable is obtained from dynamic memory profile. Variation in the number of accesses per variable due to different inputs can only change the performance. However, functional correctness cannot be affected as variable type is obtained using static analysis. We run the applications with two different inputs by changing the size and the data value. Table 8 shows how the speedup in *SNAP-M* with respect to the *GREEDY*. From this figure, we see that the performance improvement is similar across inputs. This is because changing the input data does not change the memory profile dramatically, while input size only varies the number of iterations executed in the application.

Table 8. Speedup of SNAP-M with respect to GREEDY for different inputs.

	1DFFT	2DCONV	AESD	AESE	ATAX	GEMM	GESUMMV	PHY_ACI	PHY_DEMAP	PHY_MICF
ProfileInput	1.14x	1.03x	9.98x	10.05x	1.00x	2.89x	1.03x	1.06x	1.00x	1.20x
Input1	1.14x	1.01x	10.21x	10.23x	1.00x	2.69x	1.01x	1.03x	1.00x	1.14x
Input2	1.14x	1.03x	9.99x	10.01x	1.00x	2.81x	1.01x	1.05x	1.00x	1.17x

Other platforms: In CDM framework, the memory profile information of the multi-threaded application and systems specifications, i.e. SPM size, NoC interconnect configuration, etc. are taken as input for finding optimal data allocation. Therefore, this work can be utilized for other platforms as long as the system parameters can be obtained.

Allocation of Heap variables: In this work, none of the benchmark kernels uses heap. Hence, we do not manage data allocation for such variables. However, the reserved SPM space for the heap can be managed if necessary using existing approaches [6, 14].

7 CONCLUSION

In this work, we propose Coordinated Data Management (CDM), a compile-time framework for allocating multi-threaded applications variables in SPM based many-cores. This framework identifies shared/private variables and obtains access counts (per thread) through dynamic profiling. It next utilizes the profiling results in an exact Integer Linear Programming (ILP) formulation as well as *SNAP*, an iterative, scalable algorithm for placing the data variables in multi-threaded applications, taking NoC into consideration and replicates variables when required on available memory resources. The proposed scalable strategy *SNAP-M* improves the application execution time by **1.84x** and achieves an energy savings of **1.83x** with respect to *GREEDY*.

ACKNOWLEDGMENT

We would like to acknowledge Farzaneh Salehi Minapour's contributions towards discussions in the initial phase of this project.

REFERENCES

[1] Adapteva. Epiphany Architecture Reference Manual - Adapteva, 2014. http://www.adapteva.com/docs/epiphany_arch_ref.pdf.
[2] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *Int. J. Parallel Program.*, 29(5):493–544, Oct. 2001.
[3] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '04, pages 259–267, New York, NY, USA, 2004. ACM.
[4] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, Nov. 2002.
[5] K. Bai and A. Shrivastava. Heap data management for limited local memory (llm) multi-core processors. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 317–326. ACM, 2010.
[6] K. Bai and A. Shrivastava. Automatic and efficient heap data management for limited local memory multicore architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 593–598. IEEE, 2013.
[7] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.
[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT'08*.
[9] U. Bondhugula, A. Acharya, and A. Cohen. The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. *ACM Trans. Program. Lang. Syst.*, 38(3):12:1–12:32, Apr. 2016.
[10] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.
[11] P. Brauer, M. Lundqvist, and A. Mällo. Improving latency in a signal processing system on the epiphany architecture. In *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*, pages 796–800. IEEE, 2016.
[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.
[13] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementationâĂŤa performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.
[14] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *J. Embedded Comput.*, 1(4):521–540, Dec. 2005.
[15] B. Egger, J. Lee, and H. Shin. Dynamic scratchpad memory management for code in portable systems with an mmu. *ACM Trans. Embed. Comput. Syst.*, 7(2):11:1–11:38, Jan. 2008.
[16] L. Fang, P. Liu, Q. Hu, M. C. Huang, and G. Jiang. Building expressive, area-efficient coherence directories. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 299–308, Piscataway, NJ, USA, 2013. IEEE Press.
[17] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 238–243, New York, NY, USA, 2004. ACM.
[18] H. Fu et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 2016.
[19] L. Gwennap. Adapteva: More flops, less watts. *Microprocessor Report*, 2011.
[20] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. *Replication techniques in distributed systems*, volume 4. Springer Science & Business Media, 2006.
[21] W. Hu, G. Wang, J. Chen, X. Lou, and T. Chen. Efficient scratchpad memory management based on multi-thread for mpsoc architecture. In *Scalable Computing and Communications; Eighth International Conference on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM'09. International Conference on*, pages 429–434. IEEE, 2009.
[22] A. Janapsatya, A. Ignjatović, and S. Parameswaran. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 612–617. IEEE Press, 2006.
[23] N. E. Jerger, T. Krishna, and L.-S. Peh. *On-Chip Networks*. Morgan and Claypool Publishers, 2nd edition, 2017.
[24] S. Kalray. Kalray mppa manycore 256, 2014.
[25] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, pages 628–633, 2002.
[26] J. Kangasharju, J. Roberts, and K. W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4):376–383, 2002.
[27] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, (2):66–76, 2003.
[28] J. Krarup and P. M. Pruzan. The simple plant location problem: Survey and synthesis. *European Journal of Operational Research*, 12(1):36 – 81, 1983.

[29] L. Li, H. Feng, and J. Xue. Compiler-directed scratchpad memory management via graph coloring. *ACM Trans. Archit. Code Optim.*, 6(3):9:1–9:17, Oct. 2009.

[30] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 228–237, New York, NY, USA, 1999. ACM.

[31] J. Lu, K. Bai, and A. Shrivastava. Ssdm: Smart stack data management for software managed multicores (smms). In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–8, May 2013.

[32] J. Lu, K. Bai, and A. Shrivastava. Efficient code assignment techniques for local memory on software managed multicores. *ACM Trans. Embed. Comput. Syst.*, 14(4):71:1–71:24, Dec. 2015.

[33] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, et al. The 48-core scc processor: The programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[34] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *ACM Trans. Embed. Comput. Syst.*, 8(3):21:1–21:32, Apr. 2009.

[35] A. Olofsson, T. Nordström, and Z. Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *48th Asilomar Conference on Signals, Systems and Computers*, 2014.

[36] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. Sdrm: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Proceedings of the 15th International Conference on High Performance Computing*, HiPC'08, pages 569–582, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] P. R. Panda, N. D. Dutt, and A. Nicolau. 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, July 2000.

[38] L.-N. Pouchet and T. Yuki. Polybench/c 3.2, 2012.

[39] R. A. Ravindran, P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke, and R. B. Brown. Compiler managed dynamic instruction placement in a low-power code cache. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 179–190, Washington, DC, USA, 2005. IEEE Computer Society.

[40] D. A. Richie and J. A. Ross. Opencl+ openshmem hybrid programming model for the adapteva epiphany architecture. In *Workshop on OpenSHMEM and Related Technologies*, 2016.

[41] M. Sjalander, S. A. McKee, P. Brauer, D. Engdal, and A. Vajda. An lte uplink receiver phy benchmark and subframe-based power management. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS '12, pages 25–34, Washington, DC, USA, 2012. IEEE Computer Society.

[42] A. Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE, 2015.

[43] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 401–410, New York, NY, USA, 2006. ACM.

[44] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–11. IEEE, 2016.

[45] B. D. Technology. COPRTHR2 API Reference, Jun 2016. https://bit.ly/2SIEvnf.

[46] Top 500 The List. List of Top 500 Supercomputers, Nov 2017. https://www.top500.org/list/2017/11/.

[47] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 276–286, New York, NY, USA, 2003. ACM.

[48] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

[49] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(8):802–815, Aug. 2006.

[50] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *Embedded Systems for Real-Time Multimedia, 2005. 3rd Workshop on*, pages 115–120. IEEE, 2005.

[51] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 1264–1269 Vol.2, Feb 2004.

[52] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2Nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '04, pages 104–109, New York, NY, USA, 2004. ACM.

[53] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, pages 114–120, New York, NY, USA, 2004. ACM.

[54] H. Zhao, A. Shriraman, and S. Dwarkadas. Space: Sharing pattern-based directory coherence for multicore scalability. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on*, pages 135–146. IEEE, 2010.