CECS
CENTER FOR EMBEDDED & CYBER-PHYSICAL SYSTEMS

SPM-vSharE:
Memory Management in
SPM-based Many-core Embedded Systems

Majid Shoushtari, Bryan Donyanavard, Luis Angel D. Bathen, Nikil Dutt

Center for Embedded and Cyber-Physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

{anamakis, bdonyana, lbathen, dutt}@uci.edu

CECS Technical Report 16-08

November 18, 2016

SPM-vSharE: Memory Management in SPM-based Many-core Embedded Systems

Majid Shoushtari, Bryan Donyanavard, Luis Angel D. Bathen, Nikil Dutt, *Fellow, IEEE*

Abstract—Traditional approaches for managing software-programmable memories (SPMs) do not support sharing of distributed on-chip memory resources, and consequently miss the opportunity to better utilize those memory resources. Managing on-chip memory resources in many-core embedded systems (MES) with distributed SPMs requires runtime support to share memory resources between various threads with different memory demands running concurrently. Runtime SPM managers cannot rely on prior knowledge about the dynamically changing mix of threads that will execute, and therefore should be designed in a way that enable SPM allocations for any unpredictable mix of threads contending for on-chip memory space. This paper proposes *SPM-vSharE*: an operating-system-level solution, along with hardware support, to virtualize and ultimately share SPM resources across a many-core embedded system in order to reduce the average memory latency. We present a number of simple allocation policies to improve performance and energy. Experimental results show that sharing SPMs could reduce the average execution time of the workload up to 19.5% and reduce the dynamic energy consumed in the memory subsystem up to 14%.

Index Terms—SPM; Scratchpad Memory; Hardware/Software Memory Management; Many-core Architectures; Virtualization.

1 Introduction

Future embedded computing systems are expected to use 10s to 100s of simple processing cores, known as Many-core Embedded Systems (MES), and hold the promise of increasing performance through parallel execution. However, these systems cannot rely on traditional approaches for system integration. Most notably: 1) The bus-based communication architecture is not a scalable solution for these systems – adopting Network-on-Chip (NoC) communication allows the system to reach a high level of parallelism and scalability. 2) The traditional cache-based memory hierarchy imposes a huge inefficiency in power consumption, due to the complexity of caching logic, and also in performance, due to the network traffic generated by coherence protocols. This paper focuses on the latter problem.

Software-Programmable Memories (SPMs) (also referred to as scratchpads) are a promising alternative to hardware-managed caches. For equivalent data capacity, SPMs are around 30% smaller, slightly faster, yet consume about 30% less power than cache [1]. Additionally, no coherence management is required due to their software-programmable nature. However, SPMs require explicit software management. Researchers have previously proposed methods to ease this burden [2], [3], [4], [5], [6], [7].

Most previous efforts have assumed a single thread running on a core with a private SPM in isolation of other threads running concurrently or entering the system at a later time, or they assumed the workload mix is known ahead of time. The shortcomings are twofold: 1) They do not utilize memory resources that are local to idle cores – these idle memory resources present an opportunity for more efficient use of on-chip memory to boost the overall system efficiency. 2) The variation in the memory intensity level as well as working set size between concurrently running threads in emerging diverse workloads is neglected – this variation causes on-chip memory resources to be more valuable for some threads over others. The utilization of data memory can vary not only between threads in a workload, but also within a single thread over the course of its execution. Figure 2

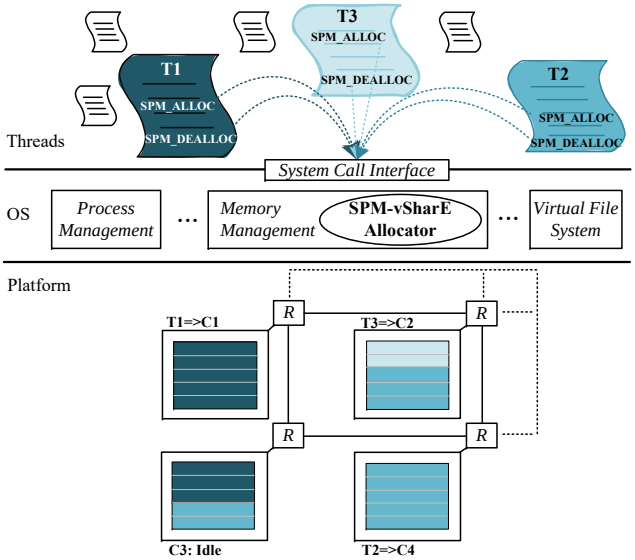
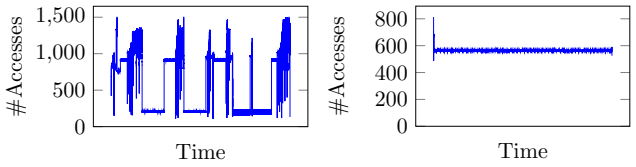


Fig. 1: SPM-vSharE allocator is invoked when a thread calls any of the SPM APIs. It intends to virtualize and ultimately share the entire distributed SPM space between all the concurrently running threads. Dedicated hardware assist enables remote allocations and remote memory accesses.

• *Authors are with the School of Information and Computer Sciences, University of California, Irvine, CA, 92697.*
E-mail:{anamakis, bdonyana, lbathen, dutt@ics.uci.edu}



(a) wikisort (b) fft

Fig. 2: Number of memory accesses measured periodically for the execution of the wikisort and fft benchmarks: utilization of memory resources changes within a thread and between threads.

illustrates the variation in temporal memory access patterns both between and within two different benchmarks.

By *sharing* the physically distributed SPM space between all the running threads through virtualization, memory resources can be utilized more efficiently. Unlike previous approaches, in this work we allow all threads to *opportunistically* exploit the entire physical memory space from all distributed SPMs for unpredictable workloads. SPM-vSharE supports dynamic SPM sharing for workloads in which threads may enter and exit at any time and contend for on-chip memory resources.

Figure 1 shows an example of a platform with four cores, when three threads (T1, T2, and T3) with various memory working set sizes are running on cores C1, C4, and C2 respectively. Previous approaches would restrict each thread to use only the physical SPM local to the core it executes on. However, by sharing the SPM space, T2's data can be placed not only on C4's SPM but also on C3's SPM, which is an idle core, and on C2's SPM, where T3 has a small working set size and does not need its entire local SPM. At the same time, T1 can also benefit from sharing by using part of C3's SPM that is not used by T2. The end result is the improvement in the *overall* performance of these three threads.

This paper proposes *SPM-vSharE*, system software and hardware support for management of distributed SPMs in many-core embedded systems. SPM-vSharE enables the system to efficiently *share* the SPM resources distributed across the chip. Sharing the SPM resources improves the *average* access latency for all the concurrently running threads, and hence improves the overall performance of the system.

It is worth mentioning that SPM-vSharE does not focus on analyzing the access pattern of data objects in order to find the most profitable data objects to bring on chip. That is an orthogonal problem which researchers have looked into extensively [8]. Instead, this paper focuses on sharing a distributed SPM space between a mix of unknown threads where each thread explicitly requests SPM space for its most accessed data objects throughout its execution.

In order to share the SPM space through virtualization, several challenges need to be addressed. This paper makes the following contributions:

- We propose a runtime allocator as part of the operating system's memory manager to support management of explicit SPM allocation/deallocation APIs.
- We present and evaluate multiple allocation policies to share the physical SPM space between concurrently running threads.
- We describe the hardware support needed to enable SPM sharing and SPM-related on-chip communications.
- We prototype SPM-vSharE in the gem5 simulator. Experimental results show improvements in performance and energy consumption.

In this paper, we define the necessary software/hardware components for sharing distributed SPMs under contention from unpredictable workloads, and build the foundation towards SPM-based many-core embedded systems.

The rest of the paper is organized as follows: Section 2 presents a detailed motivational example. Section 3 describes our software/hardware model of the target system. Section 4 describes all components of SPM-vSharE in detail. Section 5 shows the experimental results. Section 6 discusses the overheads and scalability of SPM-vSharE. Section 7 reviews previous work in this domain. We conclude the paper in Sec. 8.

2 Motivational Example

In many scenarios, the amount of data requested by a thread to be allocated on-chip is larger than the capacity of the SPM local to the core that the thread is running on. Therefore, some portion of the data must remain off-chip. In such cases, borrowing physical SPM space from neighboring SPMs could boost the overall performance.

Figure 3 shows a motivating scenario where three threads T1, T2, and T3 are sequentially entering a system with four cores (C1, C2, C3, and C4), where every core has a local data SPM. The left-hand side panel shows the state of the system when the local only allocator is used, while the right-hand side panel shows the same state when a local/remote allocator is used. In both panels, time is progressing from left to right. Details of each thread are shown in Table 1.

We consider hit latency to be 1 cycle for local hits, 5 cycles for remote hits to account for the network latency, and miss latency to be 100 cycles to account for off-chip memory accesses.

T1 enters the system in cycle 0 and is scheduled to run on C1. This thread requests 12KB of SPM space, however with local-only allocator, only 8KB of SPM can be given to this thread although C2, C3, and C4 are all idle (Fig. 3.a). However, if we allow remote allocation on neighboring cores, all 12KB can be granted to T1 (Fig. 3.a').

T2 gets scheduled on core 4 after 20 kilocycles. This thread needs 20KB of SPM space. While with local-only allocator, a maximum of 8KB can be given to this thread (Fig. 3.b), remote allocation allows us to accommodate all 20KB by borrowing space from neighboring idle cores namely C3 and C2 (Fig. 3.b').

Finally, T3 gets scheduled on core 2 after 45 kilocycles. This thread only needs 6KB of SPM space which can be granted with the local-only allocator (Fig. 3.c). However, as we saw in the previous case, with a local/remote allocator, T2 occupied the entire SPM belonging to core 2. Therefore,

TABLE 1: Details of threads in the motivational example of Fig. 3

Thread	Core	Entrance Cycle	Working Set Size	#Accesses	#Execution Cycles if all accesses are hit with access latency = 1 cycle
T1	C1	0	12KB	18000	60K
T2	C2	20K	20KB	24000	65K
T3	C4	45K	6KB	3000	40K

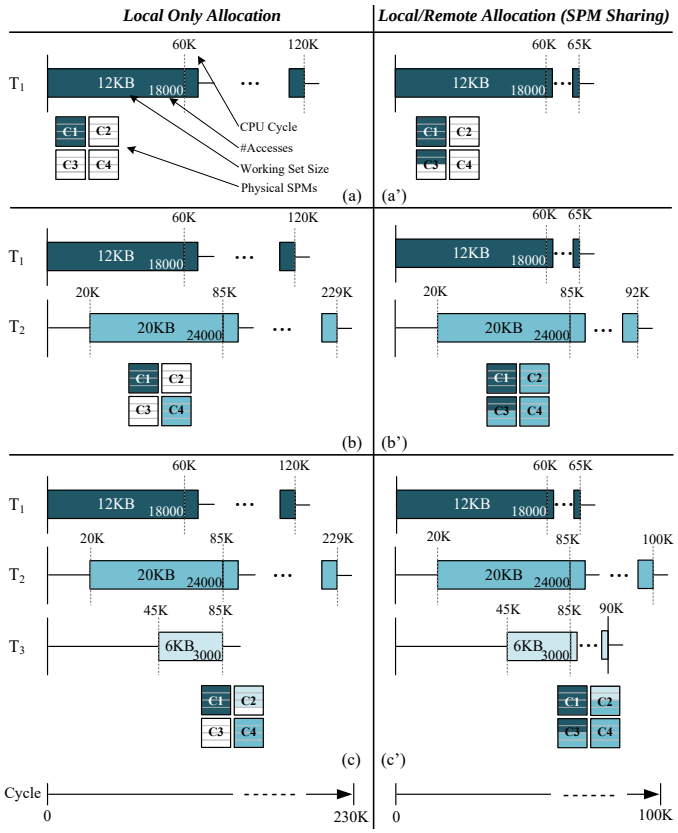


Fig. 3: An example showing how sharing the entire SPM space improves the overall performance of a 2X2 system with 3 thread running on it. The left-hand side column shows the state of SPM allocations when allocations are limited to the local SPM only; and right-hand side column shows the same scenario when remote allocations are also possible in addition to local allocations. (a)&(a'): only thread T1 is running, (b)&(b'): thread T2 starts execution after 20K cycles while T1 is still running, (c)&(c'): thread T3 starts execution after 25K cycles while both T1 and T2 are still running. Average and maximum execution times among all three threads are improved by about 2X and 2.9X respectively.

as shown in (Fig. 3.c'), half of the SPM space on core 2 is made available for T3 by evicting some of the pages that were previously allocated to T2.

To summarize, when we are limited to local SPM allocations (left panel of Fig. 3), all three threads complete their execution after 229 kilocycles, and on average each thread takes about 123 kilocycles to finish, 2.2X more than the ideal case.

But when we virtualize the entire available SPM space and enable all threads to use it (right panel of Fig. 3), all three threads complete their execution after 100 kilocycles. In this case, each thread takes about 60 kilocycles on average, which is 2X faster than the previous case. The reduction in off-chip memory traffic also results in energy savings.

This simple scenario shows how virtualizing the entire SPM space and allowing threads to compete for memory resources distributed across a chip can help improve the *overall* performance of threads running on many-core systems with SPMs.

3 Target System

Figure 1 showed a system in which a number of threads are running concurrently on a many-core system explicitly requesting SPM allocations/deallocations. Here we describe the SPM programming model, architecture model and the execution model of such a system.

3.1 SPM Programming Model

With SPM-based memory hierarchy, the programmer and/or the compiler have explicit control on data movements. This enables them to prefetch a frequently used data object earlier than its actual access. SPM-based systems usually rely on explicit allocation/deallocation requests made by threads utilizing specified Application Programming Interfaces (APIs). A set of APIs, registered with the kernel, will be used by the programmer and/or the compiler to program SPMs. The API calls we define are listed in Table 2:

TABLE 2: SPM-vSharE APIs

Method	Parameters	Type	Note
SPM_ALLOC	BaseVA	uint64	Base virtual address of memory region
	Size	uint	Size of allocation
	AllocationMode	uint	Type of allocation
SPM DEALLOC	BaseVA	uint64	Base virtual address of memory region
	Size	uint	Sizeof deallocation
	DeallocationMode	uint	Type of deallocation

`SPM_ALLOC()` and `SPM_DEALLOC()` APIs can be used explicitly in the source code to request moving parts of a thread's virtual address space between on-chip SPMs and the main memory through Direct Memory Access (DMA) transfers.

`SPM_ALLOC()` receives the base virtual address `(BaseVA)` and the size in bytes as arguments along with a flag `(AllocationMode)` which determines the allocation mode. There are two possibilities for the `AllocationMode` flag: `COPY` and `UNINITIALIZED`. When the flag is set to `COPY`, the allocated data is copied from main memory to SPM, whereas if the flag is set to `UNINITIALIZED`, the allocation simply reserves SPM space without initializing the allocated memory. `UNINITIALIZED` allocations eliminate unnecessary DMA transfers when the data object values are not initialized in memory yet, so the current values do not need to be copied from main memory.

Similarly, `SPM_DEALLOC()` receives the base virtual address `(BaseVA)` and the size in bytes as arguments along with a flag `(DeallocationMode)` which determines the allocation mode. There are two possibilities for the `DeallocationMode` flag: when this flag is set to `WRITE_BACK`, the data is written back to main memory from SPM; if the flag is set to `DISCARD`, the deallocation happens without updating the values in main memory. `DISCARD` flag is useful when the data is not needed anymore, preventing unnecessary memory write-back. A common case is to discard a piece of heap data in SPM that is going to be freed immediately, therefore does not need to be written back to main memory.

Figure 4 shows the number of cycles that allocation and deallocations of different sizes take with different modes. While the overhead of allocation and deallocation with `COPY` and `WRITE_BACK` grows substantially as the size of

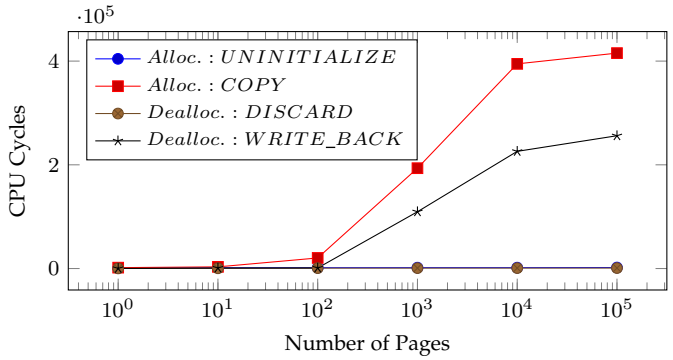


Fig. 4: Number of CPU cycles that a thread has to be stalled for an SPM API to be handled by the OS/HW. SPM page size is assumed to be 64 bytes.

allocation grows, `UNINITIALIZE` and `DISCARD` keep this overhead low and near constant.

These method calls can be inserted manually by a domain expert programmer or automatically by a compiler pass or a combination of both. Finding best candidates for SPM mapping is an orthogonal issue that previous researchers have looked into and is not the focus of this work. Currently, in a preprocessing step, we analyze the source code automatically to identify heap allocation and deallocation sites and insert `SPM_ALLOC` and `SPM_DEALLOC` for each pair, accordingly. We also add `SPM_ALLOC` and `SPM_DEALLOC` to handle each function's stack. Anytime a function is called, a `SPM_ALLOC` is responsible for bringing that function's stack on chip and right before returning from that function, a `SPM_DEALLOC` call will remove that from SPM. These SPM annotations are then pruned by profiling the program's memory accesses and removing annotations for rarely accessed data objects.

The code snippet in Fig. 5 shows how these APIs can be used within the source code of an example thread.

Calling these APIs transfers control to the operating system's SPM allocator. The SPM-vSharE allocator decides whether or not to grant the request and broadcasts messages throughout the platform accordingly. These API calls are blocking, meaning that *control does not return to the thread until the data transfers are complete.*

3.2 Architecture Model

In this work, we target many-core embedded systems in which cores are connected via a mesh-like NoC. The on-chip data memory is distributed evenly among all cores

```
...
SET_SPM_PAGE_SIZE(256);
...
int *buffer = (int*) malloc (BUF_LENGTH*sizeof(int));
SPM_ALLOC (buffer, BUF_LENGTH*sizeof(int), UNINITIALIZED);
...
for (i = 0; i<LENGTH; i++) {
  buffer [i] = buffer [i] * 2;
}
...
SPM_DEALLOC (buffer, BUF_LENGTH*sizeof(int), WRITE_BACK);
...
free(buffer);
...
```

Fig. 5: A code snippet using SPM APIs.

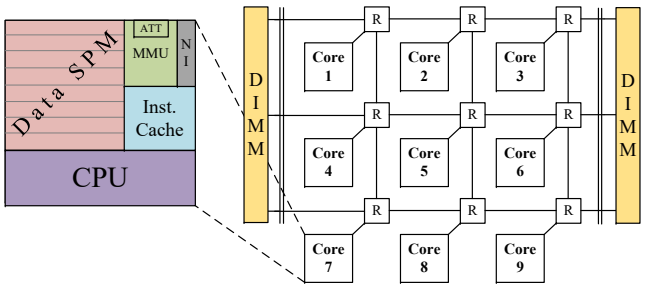


Fig. 6: A 3X3 example of target architecture model.

throughout the system. Each core consists of a simple in-order Central Processing Unit (CPU), a SPM, a Memory Management Unit (MMU), an instruction cache, and a network interface to manage NoC traffic for inter-core communication.[1] A 3X3 example of such an architecture is shown in Fig. 6. In our system, we assume that SPM address space is logically disjoint from the main memory address space, creating another layer in the memory hierarchy. Subsequently, an additional translation layer is required to map virtual memory addresses to physical SPM addresses. The Address Translation Table (ATT) within MMU does this translation.

3.3 Execution Model

Many modern embedded platforms support the concurrent execution of multiple independent applications on shared resources where applications start and stop at any time. In many-core systems, coherency is a major bottleneck for scalability. If a piece of data is duplicated on different cores, the consistency of those copies should be guaranteed by a coherence protocol. If any thread updates one of these copies, all other copies should be invalidated which is a very costly operation [9].

To avoid coherence issues, we only keep one copy of data on chip and all threads access the same physical copy. Consider a multi-threaded application with virtual address space shared between threads. Once a thread requests a part of the virtual address space to be mapped on-chip, accesses from all other threads to that region of virtual address space will be forwarded to the single copy of that data in the on-chip SPM.

4 SPM-vShare Details

SPM-vSharE is an operating system implementation to support virtualization and sharing of on-chip SPMs, and includes required architectural support. Software consists of the SPM allocator (Sections 4.1 and 4.2) and hardware support includes custom memory management units along with a network protocol (Section 4.3).

4.1 SPM Allocator

The SPM allocator, as part of the operating system's memory manager, is the core of the SPM-vSharE scheme. An SPM API call from any thread invokes the operating system

1. In this early work, for simplicity, we assume there is no data cache and focus on the feasibility of incorporating SPMs in many-core platforms. Future work will consider a hybrid Cache/SPM hierarchy.

and this call will be forwarded to the SPM-vSharE's SPM allocator to make a decision about the request. The allocation is done at the granularity of a page. The allocator aligns the boundaries of an allocation request and also makes sure that a page is not allocated more than once. The allocator, based on its policy (discussed in Sec. 4.2), determines the set of actions required and subsequently broadcasts proper SPM management messages throughout the platform to accommodate the request. The requests are served by a centralized manager in a first-come, first-served (FCFS) order and therefore there is no chance for erroneous behavior and/or deadlock in decision makings.

4.2 Allocation Policies

SPM-vSharE's allocation policies have to balance the memory needs of locally-pinned threads, while allowing for sharing of SPM resources among cores. We present both non-preemptive as well as preemptive heuristics. All policies work in a best effort manner. Their goal is to reduce the average execution time of all threads that are running, possibly degrading some for the benefit of the whole mix. They attempt to map the maximum number of requested pages to on-chip SPMs. Anything that can not be mapped to on-chip SPMs due to size limitation, stays in the main memory.

4.2.1 Non-preemptive Allocation Heuristics

In non-preemptive allocators, once the data has been allocated on-chip, it stays there until the thread explicitly calls `SPM_DEALLOC()` for that piece of data.

■ *Local Allocator(LA):*

With the *Local* allocator, a thread's allocation request is granted only if there is space available on the local SPM. The pseudo-code of this policy is shown in Policy 1. In each iteration (lines 3...7), the maximum number of free contiguous physical pages on the SPM are found and allocated. Depending on the current allocation status of the local SPM, pages requested by the thread could get allocated contiguously or dispersed throughout the physical SPM. At the end, the number of pages that are successfully mapped is returned.

The Local allocator implements the simplest allocation policy and does not need any hardware support for remote accesses. Since the state-of-the-art techniques for SPM management are limited to the physical SPM local to each core, we will use this policy as the baseline.

Policy 1: Local Allocator (LA)

```
   Input: a new allocation request for P pages from
          thread running on core[i];
   Output: number of mapped pages M;
1  R = P;
2  T = maximum number of contiguous free pages on
     SPM[i];
3  while R >0 and T >0 do
4  |   allocate min(R,T) pages on SPM[i];
5  |   R -= min(R,T);
6  |   T = maximum number of contiguous free pages on
   |     SPM[i];
7  end
8  M = P - R;
```

Policy 2: Random Remote Allocator (RRA)

```
   Input: a new allocation request for P pages from
          thread running on core[i];
   Output: number of mapped pages M;
1  R = P - LocalAllocator(P, i);
2  RL = list of cores;
3  while R >0 and RL is not exhausted do
4  |   j = Randomly pick a core from RL;
5  |   R -= LocalAllocator (R, j);
6  end
7  M = P - R;
```

Policy 3: Closest Neighborhood Allocator (CNA)

```
   Input: a new allocation request for P pages from
          thread running on core[i], hop-threshold = T;
   Output: number of mapped pages M;
1  R = P - LocalAllocator(P, i);
2  DL = list of cores within T radius of core[i] sorted
     based on distance from that core;
3  while R >0 and DL is not exhausted do
4  |   j = next element in DL;
5  |   R -= LocalAllocator(R, j);
6  end
7  M = P - R;
```

■ *Random Remote Allocator (RRA):*

The *Random Remote* allocator grants a thread's allocation requests if there is space available *anywhere* on-chip, and assigns data to random SPMs. This allocator implements the simplest allocation policy for remote SPM allocation. We have intentionally implemented this policy to evaluate the pure advantage of sharing discarding the importance of data placements. The pseudo-code of this policy is shown in Policy 2. It initially behaves like the local allocator trying to allocate as many pages as possible on the local SPM. If there are more pages to allocate, in each iteration (lines 3...6), a core is chosen randomly and the maximum number of free pages on its SPM is allocated for the thread's request.

■ *Closest Neighborhood Allocator (CNA):*

With the *Closest Neighborhood* allocator, the *nearest* available SPM slots are allocated for any allocation request. The pseudo-code of this policy is shown in Policy 3.

The order in which the neighboring SPMs are considered for data allocation (lines 2...6) is based on the hop distance from the core that issued the allocation request. Only cores that are within T radius of the home core will be considered. Figure 7 shows one example where the white core in the middle issues an allocation request. First, the allocator searches for free physical pages in the local SPM (hop distance=0). Next candidates are the SPMs distanced within one hop from the core requesting allocation (labeled 1 through 4). If more SPM pages are needed, the allocator will search the SPMs within hop distance two (labeled 5 through 12). And this process continues until all pages are allocated or all candidates are exhausted.

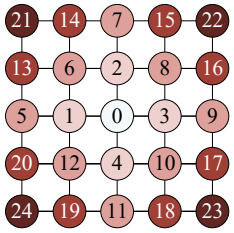


Fig. 7: Hop-distance-based search for empty SPM space in Closest Neighborhood allocation policy.

4.2.2 Preemptive Heuristics

Preemptive allocation heuristics may allow for more efficient sharing of memory resources, but could affect previously allocated data by (1) migrating it to a different place on-chip or (2) evicting it to main memory.

■ *Closest Neighborhood with Guaranteed Local Share Allocator (CNGLSA):*

This heuristic is a modified version of the Closest Neighborhood allocator that finds the nearest available SPM slot for any allocation request. However, it also guarantees that a predefined percentage of each SPM is allocated for requests from the thread running on that SPM's local core, *should that thread need it*. The pseudo-code of this policy is shown in Policy 4. Note that this policy does not reserve the SPM ahead of time, rather it frees up the required SPM space by relocating the data from there to a different SPM or to off-chip memory.

With this policy, if a new thread is assigned to an idle core while that core's SPM was already allocated for another thread, all or a portion of that SPM will be relocated/deallocated in order to be used by the new thread. Currently, we support two ways of determining the page(s) that should be relocated (line 4): (1) least recently accessed page, (2) least frequently used page. The assumption is that there are hardware counters that can keep track of these statistics at runtime. Once the page is selected, we attempt to relocate that page to a free SPM nearest to its owner. If the entire on-chip SPM space within a certain hop distance is utilized, we evict that page to the main memory.

Policy 4: Closest Neighborhood with Guaranteed Local Share Allocator (CNGLSA)

```
   Input: a new allocation request for P pages from
          thread running on core[i], hop-threshold = T,
          guaranteed local share = X%;
   Output: number of mapped pages M;
1  R = P - LocalAllocator(P, i);
2  S = local SPM share;
3  while R >0 and S <X do
4  |  EP = select a page which doesn't belong to the
   |    thread running on core[i] to be evicted;
5  |  core[j] = owner of EP;
6  |  relocate EP to the closest free SPM to core[j];
7  |  allocate one page on SPM[i] for the new request;
8  |  R -= 1;
9  |  S = local SPM share;
10 end
11 if R >0 then
12 |  R -= ClosestNeighborhoodAllocator(R,i);
13 end
14 M = P - R;
```

4.3 Hardware Support

Many of the previous allocation policies require some hardware support to manage the distributed memory mappings as well as to handle remote accesses via the NoC. This subsection explains all the required hardware assists.

4.3.1 Distributed Memory Management Units

MMUs are core on-chip components enabling all the memory-related communications. A MMU sends/receives all memory-related network messages from/to their local SPM. They are distributed throughout the chip, one MMU per core. Every memory request from the CPU is handled by the MMU. Each MMU has its own Address Translation Table (ATT) which holds all the virtual to physical address translations for the thread on that core. Each entry of an ATT (Fig. 8) holds four kinds of information: 1) the virtual page address, 2) the id of the core hosting that page, 3) the physical address on host SPM, and 4) a bit flagging if the entry is a valid entry. If no mapping exists on ATT for the memory location requested by CPU, the data cannot be found on-chip; the address translation will be done through the Translation Lookaside Buffer (TLB) and the request is forwarded to off-chip memory.

valid	virtual page address	host core id	physical SPM address

Fig. 8: ATT entry

4.3.2 Network Protocol

Table 3 lists all the network messages required to implement the SPM-related communications between MMUs via the NoC. There are 6 kinds of request messages and five types of response messages. SPMReq_READ and SPMReq_WRITE are used for remote memory read and write accesses which are respectively responded with SPMResp_DATA and SPMResp_WRITE_ACK. SPMReq_ALLOC and SPMReq_DEALLOC are also used for allocating and deallocating a virtual page and are both responded with SPMResp_GOV_ACK. In certain scenarios, allocating a page for a thread requires relocating a page from another thread. In such cases, the core that has to relocate its page sends a SPMReq_RELOCATION_READ to the current host of that page. When the data is read from SPM, the current host will forward the data with a SPMReq_RELOCATION_WRITE to the future host of that page while sending back a SPMResp_RELOCATION_HALFWAY to the owner core. The owner then can signal the core who wants to allocate a new page that the space is freed up. Future host node receives the request and after copying the data on its SPM sends a SPMResp_RELOCATION_DONE to the owner core so it can resume its execution.

4.4 Data Movements in SPM-vSharE

Three types of events encompass all possible scenarios for managing SPM data in SPM-vSharE: allocation/deallocation, access, and migration.

TABLE 3: SPM-related Network Messages

SPM Request Messages		SPM Response Messages	
Type	**Purpose**	**Type**	**Purpose**
SPMReq_READ	Request for a data read	SPMResp_DATA	Data for read requests
SPMReq_WRITE	Request for a data write	SPMResp_WRITE_ACK	Acknowledgment for write requests
SPMReq_ALLOC	Request for allocating a page on a physical SPM	SPMResp_GOV_ACK	Acknowledgment for ALLOC/DEALLOC requests
SPMReq_DEALLOC	Request for deallocating a page from a physical SPM		
SPMReq_RELOCATE_READ	Request for read part of an on-chip page migration	SPMResp_RELOCATION_HALFWAY	Used to notify a MMU that data has been relocated from specific SPM slots and those slots are now free
SPMReq_RELOCATE_WRITE	Request for write part of an on-chip page migration	SPMResp_RELOCATION_DONE	Used to signal the end of a relocation request

4.4.1 Allocations/Deallocations

Allocations and deallocations are initiated via explicit API calls in the application. An allocation request with its related information (address, size) is sent to the SPM allocator, which holds the state of all on-chip memory and can determine whether or not the request will be granted. If the SPM allocator can grant the allocation request it sends one or message(s) back to the requesting core. Every message contains a number of allocation information: number of pages granted, core hosting the allocated SPM space, base physical address on the target SPM, etc. The requesting node updates its ATT entry for the virtual page(s), it communicates with the host core and finally the host core's MMU initiates DMA transfers to fulfill the request.

Deallocation follows a similar flow: upon receiving the deallocation request, the SPM allocator notifies the requesting node so it can invalidate the associated ATT entry, and also notifies the host node so that it can invalidate the SPM space and initiate a write-back to main memory.

4.4.2 Accesses

Every memory read/write request is initially forwarded to the local MMU. The MMU looks up its ATT to see whether the corresponding data is on the local SPM, a remote SPM or off-chip memory. If the page containing the requested data exists on the local SPM, the MMU provides the physical SPM address and forwards the request to the local SPM, which returns or updates the data.

If the page is on a remote SPM, the MMU initiates a request (SPMReq_READ or SPMReq_WRITE) for the remote MMU on the node that holds the data while the requesting CPU stalls. Upon receiving the request from the NoC, the remote MMU fetches or updates the data, and returns a message (SPMResp_DATA or SPMResp_WRITE_ACK) to the requester. Finally, the requesting MMU notifies its local core upon receiving the response and execution continues.

If the page was not allocated on-chip the request is forwarded to the next level of the memory hierarchy.

4.4.3 Data Migration

Under certain conditions, we may need to relocate a page from one SPM to another SPM or evict it to main memory. This is usually needed in order to free space for a thread that has more priority to use a physical SPM in the case of preemptive allocation policies. Migrations are initiated by the allocator, and carried out by network messages (SPM_Req_RELOCATION_READ, SPM_Req_RELOCATION_WRITE, SPM_Resp_RELOCATION_HALFWAY, SPM_Resp_RELOCATION_DONE) exchanged between the affected MMUs; namely: owner of that page, current host of that page, and future host of that page. During migration, the thread whose data is being migrated is stalled to make the execution safe.

5 Evaluations

We conducted experiments to evaluate the capability of SPM-vSharE for improving the overall performance and power consumption of the memory sub-system for a SPM-based platform.

5.1 Experimental Setup

We prototyped the SPM-vSharE scheme in the gem5 architectural simulator [10]. We extended gem5 by adding the SPM, MMU, and ATT components; defining a new network protocol for SPM-related communications; and adding support for SPM API calls through pseudo-instructions. We ran our experiments in the system emulation (SE) mode of gem5 using in-order alpha cores to configure mesh NoCs, which models the contention delay in the network. Every SPM API call invokes the SPM-vSharE manager implemented inside the simulator to emulate the behavior of a real operating system.

In the following experiments, all threads begin their execution at the same time.

We used Noxim [11] to estimate the energy consumed within the NoC and CACTI [12] to estimate the SPM access energy. DRAM access energy is estimated using the results from [13].

5.2 Remote vs. Off-chip Access Latencies

While on-chip accesses tend to be faster than off-chip accesses, if the hop distance between the home core and the host core becomes too long, which is quite possible in larger platforms, the network delay could overcome the benefit of remote allocations. In the most extreme cases, an on-chip memory access may become more costly than an off-chip access. The exact hop distance that could result in degraded performance is dependent on the implementation of the network as well as the latency of off-chip DRAM. We conducted experiments to measure the pure effect of network delay on remote accesses in a 8X8 mesh NoC. For that, we run a thread on corner of the mesh (i.e., core (0,0)) accessing a 4KB stack array within a loop performing some basic arithmetic operations on elements of that array. All other cores are idle. In 15

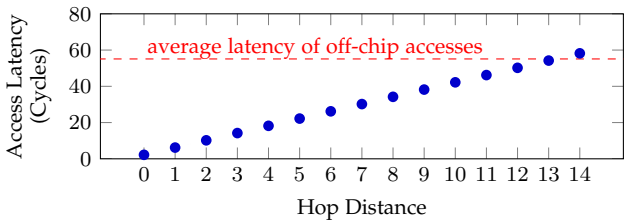


Fig. 9: Average access latency for various hop distances.

different runs, we allocate this 4KB array on all 15 possible hop distances from the home core (i.e., 0 to 14). In another run, we allocate that array on the off-chip DRAM.

Figure 9 shows the average memory access latency of this thread for different hop distances. The dashed red line shows the average access latency when the data is accessed from off-chip DRAM. On average, every hop adds 4 cycles to the latency. After 13 hops, the latency experienced by the thread to access on-chip SPM will exceed the latency of off-chip DRAM. This analysis is done assuming there is no other network traffics generated by other cores and therefore a more conservative threshold has to be selected. In our experiments we set the hop distance threshold to 10.

5.3 Memory Microbenchmark

Since our goal is to evaluate the efficiency of different allocation policies, we generated synthetic threads using an in-house configurable memory microbenchmark (mem-ubench) generator designed to stress the memory resources with different intensities (Fig. 10). This microbenchmark has three phases (prologue, body, epilogue) that are repeatedly executed and can be independently adjusted in order to make them more compute-bound or memory-bound. The prologue and epilogue consist of mainly regular and random memory operations respectively; while the body consists mainly of compute operations. Each function operates on three arrays: two globally allocated arrays, and a third local to the function. This way, we can exercise many diverse scenarios using the coarse-grained intensity setting in combination with the finer-grained duration settings.

The list of configuration parameters for mem-ubench is:

- length: This parameter controls the length of execution.
- working-set-size: This parameter sets the amount of data that the benchmark allocates and accesses.
- mem/comp-intensity: This parameter defines the ratio of memory operations to compute operations for all phases.
- prologue-duration: This parameter controls the relative duration of the first phase of execution.
- body-duration: This parameter controls the relative duration of the second phase of execution.
- epilogue-duration: This parameter controls the relative duration of the third phase of execution.

5.4 Performance Comparisons: SPM-vSharE vs. Software Cache

In the first set of experiments, we compare SPM-vSharE with a software cache. A software cache consists of a simple memory array (similar to SPM) and a software system

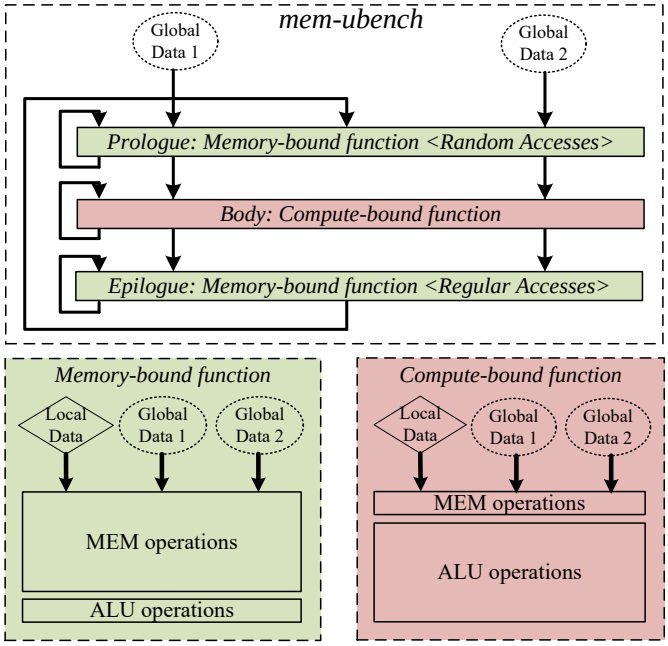


Fig. 10: Overall flow of mem-ubench execution: Each phase (prologue, body, epilogue) is either memory or compute intensive to a configurable degree. Operations are performed on three data arrays of configurable size with both regular (strided) and randomized access patterns. The number of executions of each phase as well as the entire 3-phase loop are configurable.

that is capable of automatically managing that memory to behave similar to a hardware-managed cache. Instead of using specially-designed hardware for cache management, a software cache uses general-purpose instructions.

Here, we compare the performance of a system using SPM-vSharE with a similar system that uses software caching. The software cache has the same size as the SPMs in SPM-vSharE. It implements a direct-mapped cache and its hit and miss latencies are 11 and 430 cycles respectively [14]. A SPM access which hits the local SPM takes two cycles: one cycle to translate the virtual address to physical SPM address and one cycle to access the SRAM array.

We consider two cases. In both cases, we configure a 5X5 mesh-based many-core platform with a utilization of 50%: Case A) The size of the SPM and software cache is larger than the thread's working-set, therefore SPM-vSharE allocates every memory object on the local SPMs and the software cache has a very low miss rate (Fig. 11). The address

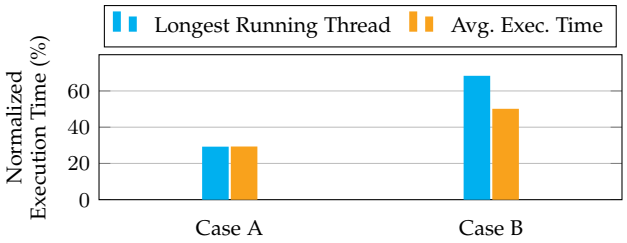


Fig. 11: SPM-vSharE vs. software cache – Case A: thread's working-set is smaller than software cache and SPM, Case B: thread's working-set is larger than software cache and SPM. Lower values are better.

translation for hit accesses in a software cache has to be done in software as opposed to the address translation table (ATT in Fig. 6) which is used in SPM-vSharE. Because of that, although most accesses are hit in both cases, the thread's performance with SPM-vSharE is significantly better compared to a system that uses software cache. Case B) The working-set-size of some threads is larger than local SPM and the software cache (miss rate = ∼15%) and some threads' working-set are smaller than the local SPM and the software cache (Fig. 11). Figure 11 shows the improvements in the average and maximum execution time among all running threads for both cases. More improvements are observed in case A compared to case B, because in case A, SPM-vSharE is allocating all data objects on the local SPM and therefore hit accesses are faster.

5.5 Performance Comparisons: SPM-vSharE Policies

In the second set of experiments, we evaluate various SPM-vSharE policies with a workload composed of different mem-ubench configurations. In all of the following experiments, we consider an 8X8 mesh-based many-core with equally distributed physical SPMs. To explore the design space, we vary local SPM size from 4KB to 16KB and core utilization from 25% to 100%. We consider four different scenarios to show the benefits of SPM-vSharE policies under various conditions. Scenario 1 evaluates SPM-vSharE in the face of core underutilization. Scenario 2 evaluates SPM-vSharE when threads have different working set sizes. Scenario 3 explores the effect of reducing network traffic for remote memory accesses. Scenario 4 explores scenarios that guarantee a share of each SPM for its locally-pinned thread, thereby improving performance in some cases.

For these evaluations, we use three configurations of mem-ubench with different working set sizes. Figure 12 shows the miss rate of each of these configurations based on the available SPM capacity. In all three configurations, around 28% of executed instructions are memory operations.

■ *Scenario 1: Core underutilization:*

The first scenario we examine is when the number of threads running on the platform is smaller than the number of cores (i.e., core underutilization). In this case not all CPUs are active, therefore some cores are not occupied and traditional SPM management techniques result in idle cores' SPM space not being used. For this experiment, we replicate identical config 2 mem-ubenchs to generate the workload.

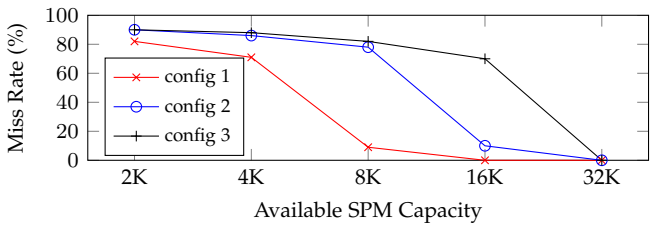


Fig. 12: Miss rate of various mem-ubench configurations for different available SPM capacities. Miss rate does not have a linear relationship with SPM capacity since not all the data is of the same importance.

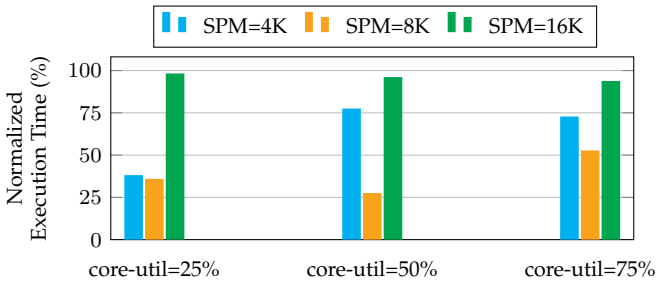


Fig. 13: Scenario 1: Average execution time among all threads in an 8x8 system - RRA normalized to LA. Core underutilization provides the opportunity to RRA to allocate more SPM space for the threads that have large working set size.

Figure 13 shows the average execution time of all threads in the workload when sharing is enabled using the simplest remote allocation (i.e., RRA) normalized to the same metric when threads can only use their local SPM (i.e., LA). This ratio is reported for three different core utilizations from 25% to 75%. For SPM sizes of 16KB, improvements are minimal to none because 16KB can fit most of the config 2's working set according to Fig. 12. In all cases, when SPMs are sized smaller, significant improvement in execution time is observed. This is due to SPM sharing – when SPMs are smaller than a thread's working set, sharing of remote SPMs prevents off-chip memory accesses that are otherwise required for allocation policies that only allow threads to use their local SPMs.

Highest improvements are observed for 8KB SPMs because the miss rate drops significantly when the available capacity becomes larger than 8KB. This is justified by observing the sharp decline in the miss rate for config 2 when moving from 8K to to 16K SPM space. The maximum improvement happens when SPM is 8KB and the utilization is 50%. This is because at this utilization and with the local allocator, 32 threads have most of their working set off-chip. Therefore, there is a high traffic on the off-chip memory side further increasing the off-chip access latency that each thread experiences. With remote random allocator, all threads get most of their working set allocated on chip, reducing the number of off-chip accesses which in turn reduced the off-chip access latency as well. At 75% core utilization, not all thread gets their working set allocated, therefore improvement is smaller.

On the other hand, when sharing is enabled with 4KB SPMs, in lower utilizations (e.g., 25%) a lot of nearby SPM space is available to compensate for the shortage of local SPM space. The improvements are expected to be slightly lower than when SPMs are 8K because in this case more accesses will be remote with higher average latency. As expected, lower utilizations generally result in larger improvements because more remote SPM space is available for threads.

■ *Scenario 2: Variation in memory working set size:*

Next, we examine a scenario that unlike scenario 1, there is a variation in the memory working set size of threads concurrently running on the platform. This would result in some cores not utilizing their entire SPM space, and can therefore lend SPM space to other cores in need even if

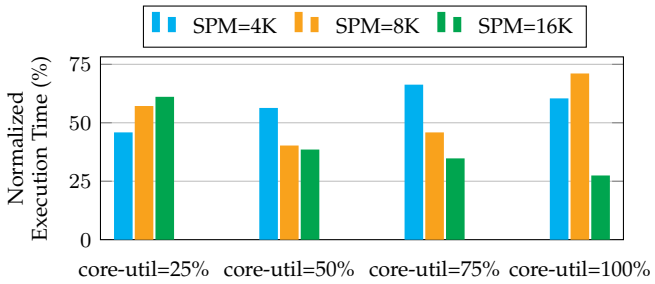


Fig. 14: Scenario 2: Average execution time among all threads in an 8x8 system - RRA normalized to LA: Threads with small working set size provide the opportunity to RRA to utilize the SPM space unused by the locally-pinned thread in order to allocate more SPM space for the threads that have large working set size.

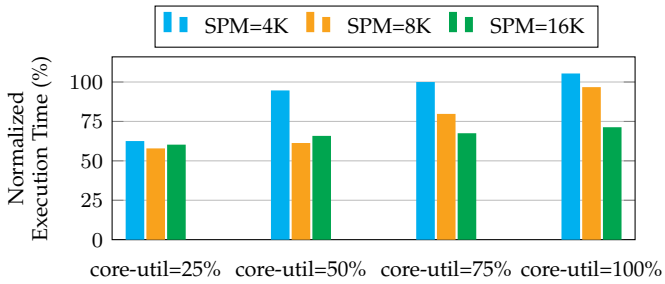


Fig. 15: Scenario 3: Average execution time among all threads in an 8x8 system - CNA normalized to RRA: CNA allocates remote pages as close as possible to the owner core resulting in reduced network traffic, faster access latency and ultimately better overall performance compared to RRA.

the core utilization is high. We examine this scenario under different core utilizations. For this experiment, we use a mix of config 1 and config 3 mem-ubenchs where config 3 has a significantly larger working set size compared to config 1.

Figure 14 shows the average execution time of different threads in the workload when sharing is enabled using the simplest remote allocator (i.e., RRA) normalized to the same metric when threads can only use their local SPM (i.e., LA).

The trend of execution time ratios with 16KB SPMs is noteworthy. Improvements increase at higher utilizations. With 16KB, all config 1 threads have enough space but config 3 threads are close to their tipping point. On the other hand, with the local allocator, at higher utilizations a significant memory traffic is generated which degrades the performance of config 3 threads dramatically because off-chip accesses take a long time to complete. The random remote allocator allocates more space for config 3 threads, alleviating both issues.

The execution time reductions when even all cores are utilized shows the benefit of SPM sharing for workloads with diverse memory requirements. This is due to the fact that although half of the threads require more SPM space than what is locally available, the other half do not fully occupy their local SPMs and can lend it to more demanding threads.

■ *Scenario 3: Reducing network traffic:*

The third set of experiments are devised to show the effect of hop distance on memory latency when borrowing space from remote cores. Like scenario 2, we use a mix of config 1 and config 3 mem-ubenchs for this experiment.

Figure 15 shows the average execution time of the workload for the neighborhood allocator normalized to the random remote allocator. Improvements are minimal for 4K SPM since a lot of them are already occupied by their locally-pinned thread especially when core utilization is higher than 25%. At the low core utilization of 25% – because there is more available SPM space in the pool of free SPMs – the execution time will be reduced more significantly when the remote allocations are done as close as possible to the owner core compared to when they are done at random hop distances.

The simple neighborhood allocator performs better than the random remote allocator in almost all cases except one. The execution time becomes higher at the highest utilization and the smallest SPM size because of the config 3 threads having a higher probability of occupying the SPM space that otherwise would have belonged to config 1. As shown in Fig. 12, config 1 threads can benefit more of 4K of SPM compared to config 3 threads. Because of the way that two types of threads are evenly distributed, every config 3 thread has four config 1 threads within hop distance of 1. Therefore with the closest neighborhood allocator, there is a 100% probability that a config 3 thread fills in the SPM of one of its config 1 neighbors should they have free space. With the random remote allocator there is a 50% chance that a config 3 fill in a config 1's local SPM.

■ *Scenario 4: Guaranteeing SPM share for locally-pinned thread:*

The neighborhood allocator looks for available SPM space with the shortest hop distance. With this policy, it is possible that a core occupies the entire SPM that is local to another core while that core itself needs it. In the fourth set of experiments, we evaluate if guaranteeing a portion of each SPM for its local thread helps improve the overall performance. Again, we use a mix of config 1 and config 3 mem-ubenchs for this experiment. The local SPM share is set to 100%.

The top part of Fig. 16 shows the difference in average local hit ratio of different threads running on a system with the neighborhood allocator with and without guaranteed local share. The improvement in local hit rate as utilization increases indicates that this improvement is due to reduced remote SPM accesses. Because the threads are intentionally mapped as sparsely as possible, at lower utilizations there are very few conflicts between neighboring cores competing for SPM space, and therefore local hit rates are nearly identical with both policies. However, this increase in local hit rate does not always turn into execution time reduction. Sometimes, the overhead associated with on-chip data relocation might offset the benefit of more local hits if the data that is being allocated locally is not accessed frequently enough.

The lower part of Fig. 16 shows the corresponding normalized execution times and as it can be seen only in two cases a noticeable execution time reduction is the gained. The large improvement when SPM size is 8K at higher utilization is due to config 1 threads. At that SPM size and with the neighborhood allocator, the neighboring config 3 thread(s)

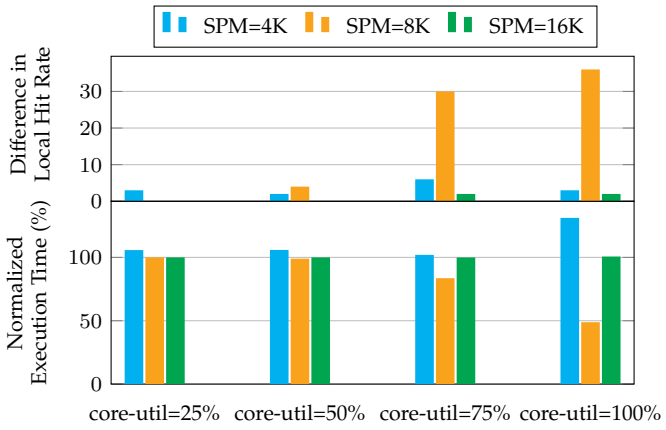


Fig. 16: Scenario 4: Difference in the local hit ratio - CNGLSA-CNA and average execution time among all threads in an 8x8 system - CNGLSA normalized to CNA: CNGLSA returns the remotely allocated SPM space to the locally-pinned thread should that thread needs it. In some cases, this results in improved local hit ratio and decreased average memory access latency compared to CNA.

that need more than 8K of SPM space, immediately fill up the entire platform leaving nothing for the config 1 threads. But when neighborhood allocator with guaranteed local share is used, config 1 threads will get all 8K SPM space that they need which results in dramatic reduction of their execution time.

5.6 Energy Comparisons: SPM-vSharE Policies

Access to off-chip memory is an order of magnitude more costly than on-chip accesses in terms of energy consumption.Although SPM-vSharE primarily targets improving the overall performance, it can also help reduce the total energy consumed by the memory subsystem.

Figure 17 shows total dynamic energy consumed for SPM accesses, off-chip DRAM accesses, and the NoC when distributed SPMs are shared using the closest neighborhood

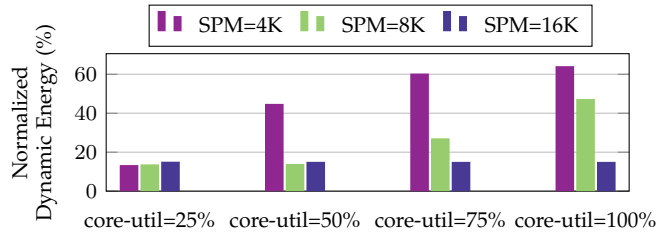


Fig. 17: Dynamic energy in memory subsystem and network: CNA normalized to LA.

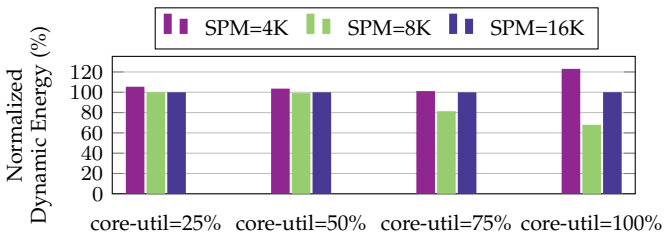


Fig. 18: Dynamic energy in memory subsystem and network: CNGLSA normalized to CNA.

policy normalized to the same metric when allocations are limited to local SPMs only. As expected, the energy savings trend is similar to the performance improvement, as both are primarily a result of reducing off-chip memory accesses.

A similar comparison is made in Fig. 18 between the neighborhood allocator with and without the use of guaranteed local share. Similar to the performance comparisons in Fig. 16, only when the benefit of accessing the data locally is higher the cost of migration energy savings can be obtained.

5.7 Experiments with Real Workload Mixes

In the last set of experiments, we configure a 4x4 system and run a number of real benchmarks on the platform under different core utilizations. Every core's SPM is 32 kilobytes. The benchmarks and their respective sources are listed in Table 4.

In order to annotate the source code of these benchmarks with SPM-vSharE APIs: (1) We run the benchmark on gem5 to collect memory traces. These memory traces are generated independent of the memory hierarchy. In the same run we collect the cycles each function was called and when it returned. We also collect the virtual address boundaries of various data objects (function stacks, heap objects, etc) (2) We feed the trace along with this information into our memory trace analyzer which analyzes the trace knowing each access in the memory trace maps to which data object tagged to its corresponding function. The report generated by the analyzer is finally used to add annotations for a subset of the most frequently accessed data objects within proper places in the source code.

We evaluate this system under different core utilizations, various working set sizes, and with two local and closest neighborhood with guaranteed local share allocators. Figure 19 shows all the mixes of benchmarks running at the same time. We start from having only two benchmarks running and in each step we add two more benchmarks to the workload mix until the core utilization becomes 75% (i.e., 12/16).

Figure 20 shows the execution time of workload with closest neighborhood with guaranteed local share allocator normalized to the execution time with local allocator which is our baseline. It reports four different values, namely: (1)normalized longest execution time among all threads, (2)normalized average execution time among all threads (3)normalized execution time of the thread benefiting the most and (4)normalized execution time of the thread benefiting the least.

The longest running thread in all first five mixes ((a) through (e)) is HUF where its execution time is reduced to around 50% in all cases. In mix (f) another thread of HUF

TABLE 4: List of benchmarks

Benchmark	Acronym	Source
fast fourier transform	FFT	[15]
huffman encoding	HUF	[16]
wikisort	WS	[16]
nbody	NB	[17]
susan-cornering	SU-C	[15]
susan-edging	SU-E	[15]
susan-smoothing	SU-S	[15]

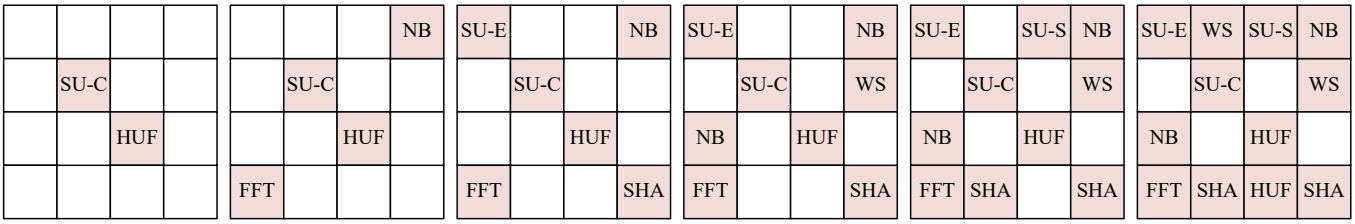


Fig. 19: Workload mixes with various core utilizations in a 4X4 platform.

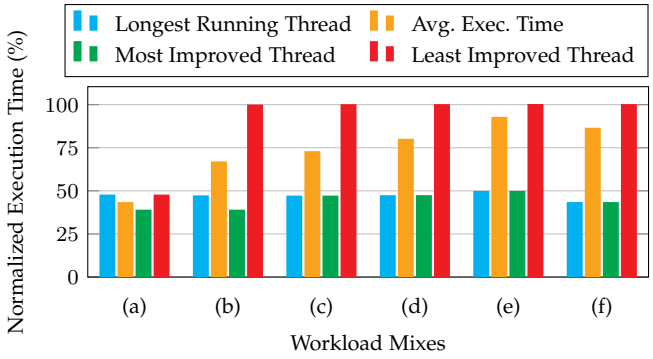


Fig. 20: Execution time comparison for various workload mixes as shown in Fig 19: CNGLSA normalized to LA. Lower values are better.

benchmark with a larger input/working set size is the longest running thread and its execution time is reduced to 43.5%. The second HUF benchmark benefited more because of its larger working set size and the availability of remote SPM capacity when all other threads have finished execution.

The average execution time has reduced to 43% - 92% depending on the core utilization and the working set size demands. Generally less core utilization means more performance improvement. But it also depends on the mix of the threads that are running at the same time. As shown in Fig. 20, going from mix (e) to mix (f) results in the average execution time to be reduced due to the extra advantage that the second HUF thread receives, as discussed earlier.

The highest improvement bars report the best performance improvement experienced by any of the threads in the mix. Similarly, the lowest improvement bars show the the least benefit experienced by a thread. Except for mix (a), in all other cases there is at least one thread whose execution time is not reduced by using the closest neighborhood with guaranteed local share allocator. And that is because there are threads whose working set size is smaller than each core's local SPM. In fact, in mixes (c) through (f) there is a thread whose execution time is increased by 0.4%. Although the local SPM share was set to 100% for this experiment, there are times that a thread has to wait for guest thread's pages to be relocated which incurs some small overhead.

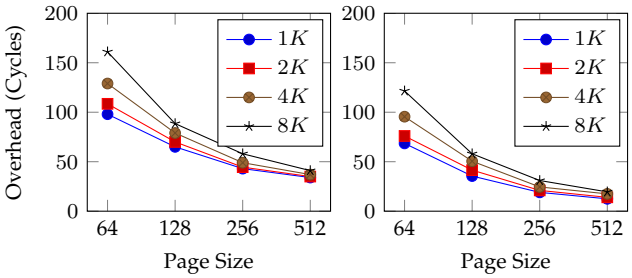
6 Discussion on Overheads

Not all components of SPM-vSharE are unique to the solution. SPM-based platforms require an API and a mechanism for address translation to enable allocation and access for local SPMs. However, additional hardware and software components of SPM-vSharE are required to enable sharing of remote SPMs.

The software component that incurs additional overhead for SPM-vSharE is the SPM allocator. All SPM-based platforms require a mechanism for allocation and a policy to resolve contention (e.g. *sharing* or *over-subscription* of the local SPM). For an allocator that does not share the entire SPM space, if we assume the SPM page size is P and the allocation size is S, the worst-case time complexity is O(P*S). Since SPM is not shared, the time complexity is not dependent on the number of cores. For allocators that share the entire SPM space between threads, if we assume the SPM page size is P, the allocation size is S, and the number of remote SPMs accessible by the core requesting allocation is N, the worst-case time complexity is O(P*S*N). Accurate analysis of runtime overhead requires implementing the policies inside operating system which is the topic our future work. To estimate the upper bound of execution times, we measured the time spent in the allocation and deallocation routines inside gem5 every time a thread calls an SPM API. Figures 21 and 22 show the runtime of these routines for different page sizes and allocation sizes.

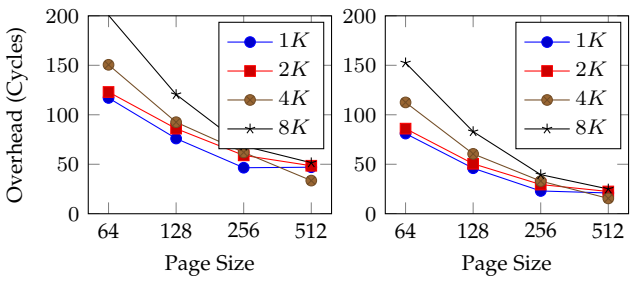
The SPM allocator must additionally maintain system-wide SPM state, which includes information specifying each unique virtual page occupying each on-chip SPM page, as well as a list of free pages.

Hardware overhead associated with SPM-vSharE manifests in the form of address translation and network communication. Similar to the allocator algorithm, storage for translation required in SPM systems depends on the number of pages per SPM: each core must maintain address translation for its local SPM. Supporting sharing means that each core must now have the capacity to additionally store translation information for remote SPM pages it has access



(a) Allocation routine (b) Deallocation routine

Fig. 21: Runtime overhead of SPM allocator for different combinations of allocation size and page size: local policy (simplest policy).



(a) Allocation routine (b) Deallocation routine

Fig. 22: Runtime overhead of SPM allocator for different combinations of allocation size and page size: closest neighborhood with local reservation policy (most complex policy).

to. In order to mitigate this storage overhead, we can adjust the page size to reduce the number of ATT entries, or limit the amount of total SPM space a core can allocate to a subset of the entire physical SPM space.

As illustrated in Sec. 5, the additional on-chip communication required by SPM-vSharE yields an overall improvement in average access latency as well as system energy consumption. Although more transactions are required in the NoC to allocate, access, and update local ATT entries, the reduction in off-chip accesses mitigates this penalty. However, as the size of the platform grows to 10s or 100s of cores, the latency penalties incurred by excessive hop distances between cores and remote SPMs could become comparable to off-chip accesses. Therefore, there needs to be a platform-specific threshold for the maximum hop distance when an allocation policy searches for available space during a remote SPM allocation.

7 Related Work

Extensive research exists on SPM management. Initially, researchers focused on static approaches to manage the SPM for a single thread running on a single-core architecture. The static SPM management techniques identify the part of overall data set that maximally improves the runtime performance of applications (e.g. most frequently used data) upon placement in SPM [18], [19], [20], [21], [22], [23]. However, data placement is fixed throughout the application's execution. Alternatively, dynamic approaches allow the movement of data at runtime, enabling the system to swap in the more frequently accessed data and swap out less-used data over time and achieve greater performance optimization [24], [25], [26].

Other works have considered a scenario in which multiple tasks are simultaneously running on a single-core architecture and sharing a single physical SPM. [27] proposes a compile-time analysis approach to support concurrent execution of tasks sharing the same SPM resource and assumes all working sets are known at compile time. [28] provides a high-level programming interface for SPM and DMA which can be used by the programmer for heap management. At runtime, a dynamic memory manager responds to memory space requests and maps data to the physical SPM as long as there is space.

More recently, the advent of integrating multiple processor cores on a single chip has resulted in a shift of focus for SPM management in the direction of multi-core processors. In the realm of multi-core SPM management, [3], [5], [6], [29], [30] all propose various compiler-based dynamic data management techniques for a memory hierarchy that incorporates SPM transparently without requiring explicit memory management.

Additionally, approaches have been proposed for managing SPM data in multi-core systems with multiple tasks sharing SPM space. [4], [31] both propose compile time static analysis techniques for SPM allocation for a fixed set of tasks. Similarly, [32] defines an OpenMP extension and compiler optimization to allocate parts of data arrays to distributed SPM for parallel programs executing on an MPSoC. [33] proposes algorithms that use profiling information to produce SPM mappings in order to minimize the worst case response time of a multitasking workload sharing SPM. These approaches all define SPM data allocation and mapping prior to execution, and, whether static or dynamic, therefore cannot handle diverse and unpredictable workloads.

Runtime adaptive approaches typically incorporate compile-time information with runtime observations to make allocation decisions at runtime. [34] profiles a fixed set of multimedia applications with varying inputs and passes this information to a runtime routine. The runtime routine monitors the application behavior and attempts to match it to one of the known profiles, and maps data to SPM accordingly to reduce energy consumption. [35] defines a framework that allows programmers to guide runtime decisions for allocating heap data to SPM in order to reduce energy consumption. [36] and [37] both propose hybrid memory hierarchies (i.e. caches + SPM) that support globally addressable and coherent address spaces. [38], [39], [40], [41] require programmer guidance for allocation, but also specify a virtualization layer that abstracts the explicit SPM management from the programmer and supports over-subscription. All of these adaptive approaches make allocation decisions at runtime for multi-core architectures with multiple tasks sharing and contending for SPM space.

In SPM-vSharE, we enable unpredictable mix of threads contending for SPM space to transparently allocate and access SPMs physically distributed throughout a chip. Through virtualization, our solution supports unlimited SPM allocation sizes, and makes global allocation decisions to benefit the performance of the overall workload. The access and allocation mechanism and policies are scalable for many-core systems.

8 Conclusion and Future Work

We presented SPM-vSharE system software and hardware support for management of distributed SPMs in many-core embedded systems. SPM-vSharE enables the system to efficiently share the SPM resources distributed across the chip. Sharing the SPM resources improves the average access latency for all the concurrently running threads, and hence improves the overall performance of the system by reducing the average execution time (up to 19.5%). It also could save dynamic energy consumed in the memory subsystem by reducing off-chip memory accesses as well as network traffic (up to 14%).

Our ongoing work targets integrating SPM-vSharE into Linux operating system. In the future, we plan to devise more intelligent policies for the SPM allocator that consider heterogeneity in functional/power characteristics of on-chip memory resources. We would like to explore a mixed SPM/cache hierarchy for many-core embedded systems. We also would like to design policies that take the pattern of accesses to data objects into account in order to perform more efficient local and remote allocations.

REFERENCES

[1] R. Banakar *et al.*, "Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002.

[2] A. Pabalkar *et al.*, "SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories," in *High Performance Computing - HiPC 2008*, ser. Lecture Notes in Computer Science, P. Sadayappan *et al.*, Eds., 2008, vol. 5374, pp. 569–582.

[3] J. Lu *et al.*, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)," in *Proceedings of the 50th Design Automation Conference*, 2013.

[4] V. Suhendra *et al.*, "Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[5] K. Bai *et al.*, "Stack Data Management for Limited Local Memory (LLM) Multi-core Processors," in *Proceedings of the International Conference on Application Specific Systems, Architectures and Processors*, 2011.

[6] A. Kannan *et al.*, "A Software Solution for Dynamic Stack Management on Scratchpad Memory," in *Proceedings of the Conference on Asia and South Pacific Design Automation*, 2009.

[7] K. Bai *et al.*, "CMSM: An Efficient and Effective Code Management for Software Managed Multicores," in *Proceedings of the international symposium on Hardware/Software Codesign and System Synthesis*, 2013.

[8] J. Cong *et al.*, "A reuse-aware prefetching scheme for scratchpad memory," in *Proceedings of the 48th Design Automation Conference*, 2011.

[9] Q. Zhao *et al.*, "Dynamic Cache Contention Detection in Multi-threaded Applications," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2011.

[10] N. Binkert *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.

[11] V. Catania *et al.*, "Noxim: An Open, Extensible and Cycle-accurate Network On-chip Simulator." in *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2015.

[12] N. Muralimanohar *et al.*, "CACTI 6.5: A Tool to Model Large Caches," HP Laboratories, Tech. Rep., 2009.

[13] A. N. Udipi *et al.*, "Rethinking DRAM Design and Organization for Energy-constrained Multi-cores," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[14] C. Pinto and L. Benini, "A highly efficient, thread-safe software cache implementation for tightly-coupled multicore clusters," in *Proceedings of the 24th International Conference on Application-Specific Systems, Architectures and Processors*, 2013.

[15] M. R. Guthaus *et al.*, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proc. of the Workload Characterization*, 2001.

[16] "Coyotebench," https://github.com/Microsoft/test-suite/tree/master/SingleSource/Benchmarks/CoyoteBench, accessed: 2016-10-31.

[17] "The computer language benchmarks game," http://benchmarksgame.alioth.debian.org/, accessed: 2016-10-31.

[18] P. R. Panda *et al.*, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," in *Proceedings of the European Conference on Design and Test*, 1997.

[19] J. Sjödin and C. von Platen, "Storage Allocation for Embedded Processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[20] O. Avissar *et al.*, "Heterogeneous Memory Management for Embedded Systems," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[21] N. Nguyen *et al.*, "Memory Allocation for Embedded Systems with a Compile-time-unknown Scratch-pad Size," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2005.

[22] M. Verma *et al.*, "Data Partitioning for Maximal Scratchpad Usage," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2003.

[23] S. Steinke *et al.*, "Assigning Program and Data Objects to Scratchpad for Energy Reduction," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, 2002.

[24] A. Dominguez *et al.*, "Heap Data Allocation to Scratch-pad Memory in Embedded Systems," *J. Embedded Comput.*, vol. 1, no. 4, pp. 521–540, 2005.

[25] L. Li *et al.*, "Memory Coloring: A Compiler Approach for Scratchpad Memory Management," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[26] M. T. Kandemir *et al.*, "Dynamic Management of Scratch-Pad Memory Space," in *Proceedings of the Design Automation Conference*, 2001.

[27] L. Gauthier *et al.*, "Minimizing Inter-task Interferences in Scratch-pad Memory Usage for Reducing the Energy Consumption of Multi-task Systems," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2010.

[28] P. Francesco *et al.*, "An Integrated Hardware/Software Approach for Run-time Scratchpad Management," in *Proceedings of the 41st Annual Design Automation Conference*, 2004.

[29] K. Bai and A. Shrivastava, "Heap Data Management for Limited Local Memory (LLM) Multi-core Processors," in *Proceedings of the 23th international symposium on System Synthesis*, 2010.

[30] ——, "Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures," in *Proceedings of the International Conference on Design Automation and Test in Europe*, 2013.

[31] D. Cho *et al.*, "Compiler Driven Data Layout Optimization for Regular/Irregular Array Access Patterns," in *Proceedings of the ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2008.

[32] A. Marongiu and L. Benini, "An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs," *Computers, IEEE Transactions on*, 2012.

[33] V. Suhendra *et al.*, "Scratchpad Allocation for Concurrent Embedded Software," *ACM Trans. Program. Lang. Syst.*, 2010.

[34] D. Cho *et al.*, "Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[35] N. Deng *et al.*, "A Semi-automatic Scratchpad Memory Management Framework for CMP," in *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, 2011.

[36] L. Alvarez *et al.*, "Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[37] R. Komuravelli *et al.*, "Stash: Have Your Scratchpad and Cache It Too," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[38] L. A. D. Bathen *et al.*, "SPMVisor: Dynamic Scratchpad Memory Virtualization for Secure, Low Power, and High Performance Distributed On-chip Memories," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2011.

[39] ——, "VaMV: Variability-aware Memory Virtualization," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012.

[40] L. Bathen and N. Dutt, "HaVOC: A hybrid Memory-aware Virtualization Layer for On-chip Distributed ScratchPad and Non-Volatile Memories," in *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference*, 2012.

[41] L. A. D. Bathen and N. D. Dutt, "SPMCloud: Towards the Single-Chip Embedded ScratchPad Memory-Based Storage Cloud," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 19, no. 3, pp. 22:1–22:45, Jun. 2014.