Automatic Management of Software Programmable Memories in Manycore Architectures

Aviral Shrivastava[1,*], Nikil Dutt[2], Jian Cai[1], Majid Shoushtari[2], Bryan Donyanavard[2], Hossein Tajik[2]

[1]Arizona State University
[2]University of California, Irvine
[*]Aviral.Shrivastava@asu.edu

Abstract: Software Programmable Memories, or SPMs, are raw on-chip memories that are not implicitly managed by the processor hardware, but explicitly by software. For example, while caches fetch data from memories automatically and maintain coherence with other caches, SPMs explicitly manage data movement between memories and other SPMs through software instructions. SPMs make the design of on-chip memories simpler, more scalable, and power efficient, but also place additional burden for programming of SPM-based processors. Traditionally, SPMs have been utilized in embedded systems, especially multimedia and gaming systems, but recently research on SPM-based systems has seen increased interest as a means to solve the memory scaling challenges of manycore architectures. This article presents an overview of the state of the art in SPM management techniques in manycore processors, summarizes some recent research on SPM-based systems, and outlines future research directions in this field.

1. Introduction

Caching as a concept has been highly successful in various aspects of computer system design. Examples of cache memory abound in contemporary designs: a disk buffer cache is a small amount of buffer memory present on a hard drive to speed up the access to the disk; a web cache provides a mechanism for temporary storage of web documents to improve user experience; a DNS cache stores queried results for a period of time in the domain name system to make DNS lookup faster; P2P caching is a technique to reduce bandwidth costs for content on peer-to-peer networks; and database caching is a mechanism used to cache database content in multi-tier applications.

Perhaps the earliest use of cache memory was in processor design, where caching in various forms (e.g. data caching, instruction caching, page table caching) was exploited for improving performance by reducing accesses to slower off-chip memories. Caches store frequently accessed data in a memory close to the processor to make the memory accesses faster and consume less power. Traditionally, caches in the processor have been implemented in hardware. Here the movement of the data between caches and main memory (i.e., caching) is performed automatically by hardware—with the software oblivious to the caching mechanisms. Caching in computer architecture is typically implemented in hardware to reduce latency because each cache access is typically in the timing critical path of instruction execution. Another advantage of hardware caching is that programmers can develop software without worrying about caching since it is handled automatically in hardware. Furthermore, in order to bridge the increasing latency of memory accesses,

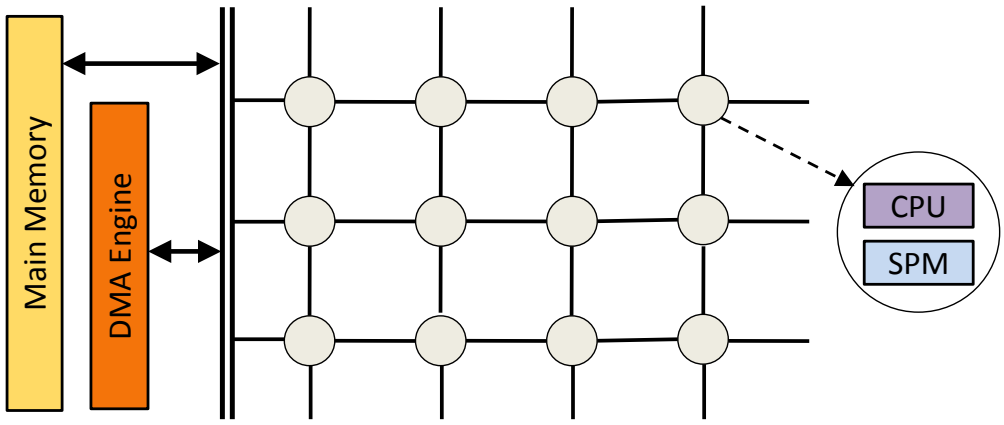

Fig. 1. *A Software programmable Memory Manycore (SMM) architecture has multiple cores, each core with a Software Programmable Memory (SPM). The data transfers between the SPMs and the main memory of the system takes place through Direct Memory Access (DMA) instructions that must be explicitly specified in the software.*

multiple levels of caching (organized as a cache hierarchy) became popular in processor architectures.

However, as we scale to manycore systems, it becomes increasingly challenging to scale the corresponding cache-based memory hierarchies[1, 2, 3]. One important reason is because the overhead of coherence logic increases rapidly with the number of cores. Some processors have already tried to alleviate this problem by removing hardware cache coherence from processors either partially or completely, e.g. Intel SCC [4], Kalray MPPA-256 [5]. In these architectures, the coherence—whenever needed by the application/system—must be implemented in software. However in these systems, caching—without coherence—is still implemented in hardware. The fact that hardware caching in manycore architectures becomes power-hungry due to the complexity of caching logic is another challenge hardware implemented caches are facing in scaling to manycore architectures.

An alternative mechanism is to deploy software caching mechanisms for smart data management, using the raw memories in the processor. Here the data movement between the close-to-processor memory and the main memory has to be done explicitly in software, typically done through the use of Direct Memory Access (DMA) instructions. We refer to such architectures as Software programmable Memory Manycore architectures (SMM), and the raw memories in such processors as Software Programmable Memories (SPM). Figure 1 shows an example of a typical SMM architecture.

SPMs offer many advantages over caches. When application designers have deep understanding of the data requirements of their applications—especially in embedded systems—the use of SPMs allows developers to exploit application semantics effectively to achieve efficient execution. SPMs offer many other advantages over caches. The first is power efficiency by eliminating the hardware overhead of traditional caching. The second is predictability, a critical factor for real-time systems. It is hard to estimate the worst-case execution time (WCET) of software executing in cached architectures, since cache replacement policies executing in hardware result in unpredictable execution times for cache hits and misses. On the other hand, SPMs allow predictable estimation of WCET since all memory accesses are explicitly controlled in software. Third, there is potential for performance improvement by orchestrating the management of data transfers ex-

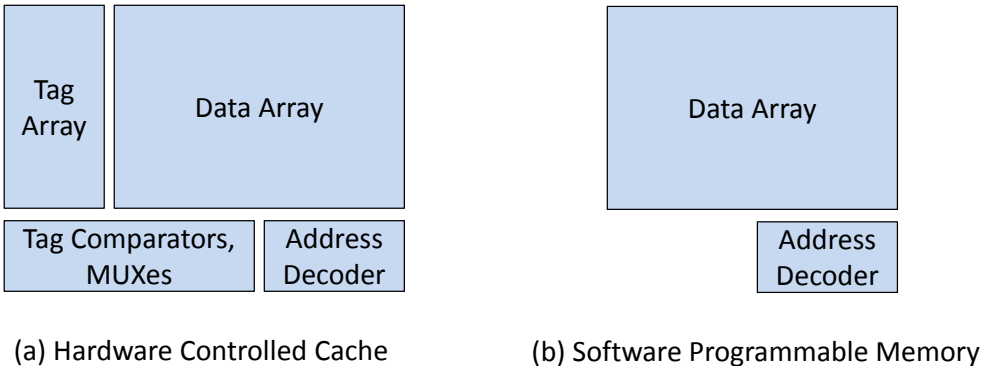

Fig. 2. *Difference between (a) Cache and (b) SPM—the hardware view: SPM is raw memory without the hardware mechanism to manage it (as is present in caches).*

plicitly in software. Other potential benefits include the ability to explicitly manage data accesses for thermal and wearout constraints, particularly for emerging memory technologies, e.g. non-volatile memories (NVM). While there is a large body of work on using SPMs for guaranteeing WCET (e.g. for real-time applications), this article focuses on the use of SPMs for efficiency, such as in improving average-case performance, reducing power consumption, managing thermal constraints, mitigating the effects of aging, etc.

Early efforts in programming SPM-based architectures required application developers to insert data management instructions manually. However, with the increasing complexity of embedded software [6], as well as the diversity of the underlying architectures, automated techniques are required to understand the application and insert data management instructions automatically. Automatic insertion of data management instructions can be achieved statically (by programmers or the compiler), or dynamically (through runtime systems that execute additional instructions to achieve the desired effect).

In the following we first describe SPMs and compare them with caches (Section 2). We then provide a comprehensive overview of SPM management for many-core architectures, including recent and ongoing research on this topic (Section 3). We conclude in Section 4 with some notes on future work.

2. What is an SPM? or SPMs vs. Caches

A Software Programmable Memory (SPM) refers to the internal data and instruction memory array incorporated into a processor or System on Chip (SoC) architecture and controlled by software (the application itself, compiler, operating system, or a combination of them), e.g. scratchpad memory in the IBM Cell processor [7], TCM in ARM processors [8], or configurable memories in TI DSP processors [9]. SPM is attached to the processor in much the same way as the L1 cache. However, SPM is *raw memory*, in the sense that it only contains decoding and column access logic, without the complex circuitry required to achieve hardware control of replacement policies, and managing coherence (tag directory, tag look-up circuitry, etc.). As Figure 2 shows, while a cache stores both the data and its address, an SPM only stores data, avoiding the extra lookup circuitry. As a result, SPMs use less area yet consume significantly less power than caches (for the same data capacity) [10].

Functionally SPMs are similar to caches, in that they allow for fast access to frequently used

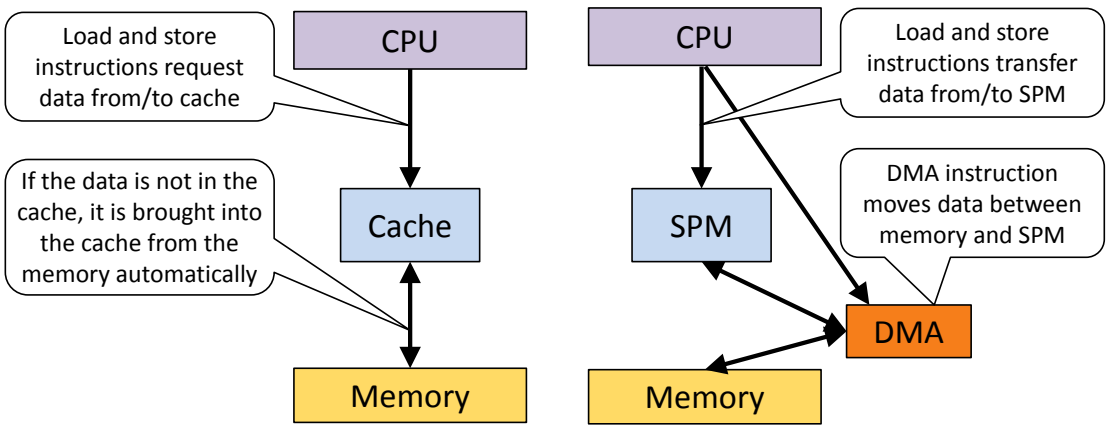

Fig. 3. *Difference between Cache and SPM—the software view: the data movement to and from the cache is performed automatically in hardware, in SPM-based systems, it must be present in the software in the form of data movement instructions.*

data, but with lower power and latency. However, replacing caches with SPMs comes with its own set of challenges, as in Figure 3. Using caches is automatic; if desired data is not present in the cache, hardware mechanisms are built to bring the requested data into the cache, potentially preventing the necessity of a repeated operation if the data is reused. However, SPM contains no such hardware mechanism to automatically bring the data that is requested to the SPM. It must be brought in explicitly through memory transfer instructions that trigger DMA transfers. Furthermore, once data brought in, it must be accessed using its new address in the SPM, and not the original address in the main memory.

While there are some challenges in using SPMs instead of caches, the promise is that execution on SPM-based systems can be more efficient. Caches are a one-size-fits-all approach. They have one way of managing data, regardless of how the data is actually accessed. Whether some data is accessed randomly, or is accessed in a first-in-first-out manner, on a cache-based system, it will always be accessed in the manner implemented in hardware. On the other hand, SPM-based systems allow more efficient management of data by exploiting application semantics and knowledge of data access patterns, thereby enabling customization of data movement across the memory hierarchy. For example, stack data in SPMs can be managed on stack-frame level instead of cache blocks, since whenever a function call happens, all the data within the stack frame will be needed during the execution of the function most of the time. By loading all the data in a stack frame at once, we can reduce the overhead for checking if the requested cache blocks during the execution of the function are already in the SPM. More importantly, by analyzing application data access patterns, we can achieve further efficiency. If we know multiple stack frames of function calls along some path in the call graph can be held in the SPM at the same time, we can bring all these stack frames from the main memory into the SPM at once, instead of fetching each of them separately. By doing so, we can reduce number of memory transfers, and eliminate status checking of stack frames between these calls.

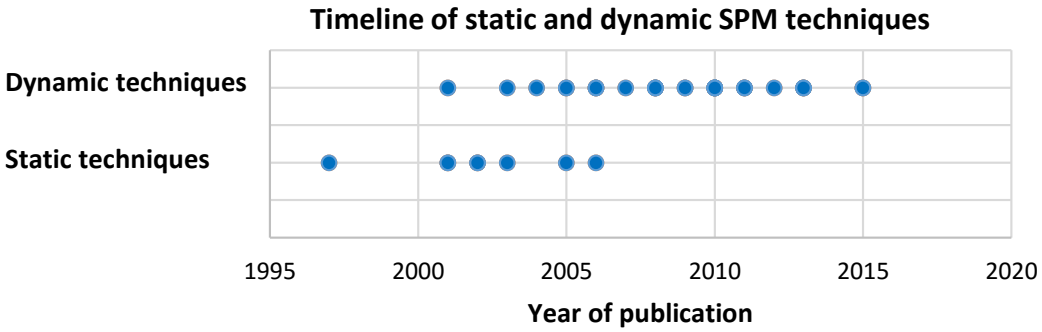

Fig. 4. *The trend of SPM management research is shifting from static to dynamic techniques.*

3. SPM Management for Modern Manycore Architectures

Existing processors with SPMs mostly leave the management of data to compiler or programmers, e.g. TI DSP processors [9], or some ARM-based processors [8]. While the IBM cell processor [7] manages code in the SPM by dividing the SPM space into regions and overlaying all the functions (in a program) to these regions so that application code that cannot fit into the available SPM space can be run by loading a function into its region right before it is called, it does not manage stack, heap or global data. Therefore, we focus on academic research efforts of SPM management in this paper.

The work on enabling efficient execution of tasks on an SPM-based manycore architecture (example illustrated in Figure 1), can be divided into three categories: i) techniques to manage the data of one task per core (each core has one SPM), ii) techniques to partition one SPM among the multiple tasks running on one core, and iii) techniques to distribute the data of multiple tasks on the multiple cores thus multiple SPMs of the system.

3.1. Running One Task on Each core

Research on SPMs started with techniques to manage data of one task per core with one SPM. As is clear from Figure 4, the earliest techniques to do this were static techniques [11, 12, 13, 14, 15, 16, 17] – implying that the placement of data on the SPM is fixed during runtime, meaning that the locations of data can not be changed during execution. Static SPM management techniques attempt to identify parts of data that maximally benefit the runtime performance of applications (e.g. most frequently used data) upon their placement in SPM.

One of the very useful properties of static techniques to manage data on SPM is the determinism in execution that it provides. There is no data miss in SPM based systems. The data is either in the SPM or in main memory, and its locations are known at compile-time. Due to the predictability in execution time, a lot of research has focused on static approaches to manage the SPM for real-time systems to minimize the upper bound of the WCET of applications. Mapping techniques that target WCET reduction statically allocate data to SPM in order to increase the predictability of execution in SPM-based systems [18, 19, 20, 21, 22, 23, 24]. These approaches utilize expensive optimization algorithms for code analysis, which are not amenable to average-case execution time optimization.

Static approaches did not take into consideration dynamic program behavior and therefore were

not able to fully exploit the benefits of SPM for general purpose computing. To solve this problem, as shown in Figure 4, research efforts began investigating dynamic SPM management techniques. Dynamic approaches allow the movement of data at runtime so they can swap in the more frequently accessed data and swap out less-used data over time to achieve greater performance optimization. [25] proposed an approach for managing global and stack variables. It divides the execution of applications into regions (an interval between any two consecutive program points chosen by its proposed approach), brings the most profitable data from the main memory to SPM, and evicts some of the resident data based on a profile-driven cost model. [26] proposes a similar approach to [25] for heap management that employs profiling to divide execution of applications into regions and places variables with high frequency-per-byte accesses in SPM. The novelty of this work is that it always allocates a fixed number of objects in the SPM and leaves the other objects in the main memory, enhancing applicability even when the size of objects is unknown. [27] targets array data management. It divides a SPM into pseudo registers and applies an existing register allocation algorithm to allocate frequently accessed arrays into the SPM that it detects using profiling. [28] uses profiling information and loop transformation techniques (such as tiling) to improve data locality in loop nests with array accesses, and maps array sections to different levels in the memory hierarchy. [29] shows the benefits of exploiting hybrid memory subsystems consisting of SRAM and NVM SPMs. It proposes a dynamic data management algorithm which places the most-written data in SRAM and the most-read data in NVM in order to realize the full potential of the hybrid SPM. The proposed approach automatically moves data at runtime using information from profiling to determine data allocation and movement. Some approaches are proposed specifically for processors with both caches and SPMs. [30] partitions scalar and arrayed variables into off-chip DRAM and on-chip SPM to minimize the execution time of embedded applications. [31] introduced dedicated hardware to speedup memory transfers between the SPM and the main memory, as well as a programming interface for runtime SPM management. [32] proposed a code placement technique that divides and maps code into cacheable and non-cacheable (i.e. SPM and DRAM) memory regions to minimize energy consumption. Data with low temporal locality is also placed in the non-cacheable regions.

As a special kind of data, code is essential for running any programs, and its management has also been studied for SPMs. One strategy divides the SPM into regions, and deploys code overlays to dynamically use these regions. Here the tasks of code management are to i) determine the best mapping between objects and regions, and ii) minimize memory transfers given the mapping. [33] creates procedures for loops, and uses profiling to identify the functions with the highest access frequency—its corresponding number of dynamic instructions executed—and maps these functions into the SPM regions. [34] identifies functions and frequently executed basic blocks (known by profiling) which are laid out in memory contiguously as memory objects or overlays, and uses a first-fit heuristic to map these memory objects into the SPM regions. Both [33] and [34] assume a given division of the SPM into memory regions. [35] determines function mappings to SPM at compile-time for embedded systems in order to reduce energy consumption.

The techniques discussed in the last paragraph can be considered as compiler-based techniques, since the changes are done in the applications, e.g. inserting the DMA instructions. The problem of SPM management can also be solved at the operating-system level as well, through page table management. [36] assumes a horizontal memory architecture with a combination of SPM and cache for instructions. The authors try to identify candidate page clusters of instructions for mapping to SPM using a postprocessing step. They support demand paging of these instructions at runtime, and propose a memory manager to track and prefetch frequently executed instruction

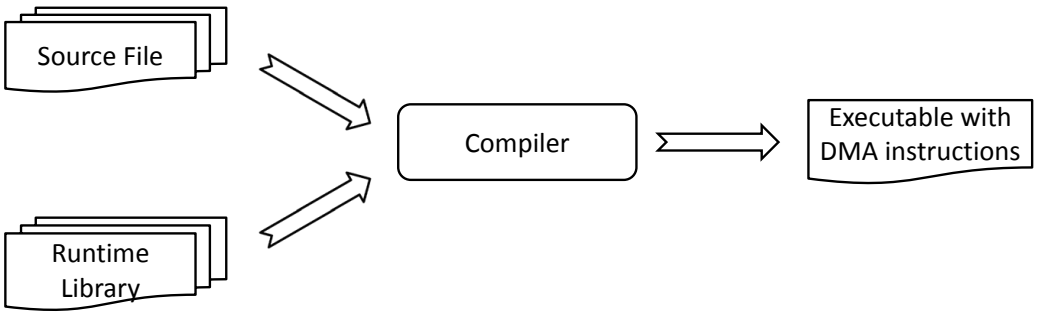

Fig. 5. *The general flow of a compiler-based SPM management on manycore processors.*

pages. [37] proposes a slightly modified version of the same approach, in which all information is extracted from a post-processing step, amending the binary to handle demand paging of designated instruction pages at runtime into a subset of SPM space in lieu of an MMU. [38] explores the opposite scenario, providing an SPM-hidden solution for runtime SPM mapping of multitask workloads that are dynamically created (i.e. unpredictable).

The research efforts mentioned so far, offer only a somehow incomplete solution to the application data management problem of one task on one SPM, e.g. only manage code, stack or global variables. Also these techniques suffer from several limitations, e.g. do not support pointers, recursive functions, etc. These limitations dramatically reduce the programmability of the target architectures. Finally, some of these approaches are only applicable to a specific memory hierarchy, e.g. with both SPMs and caches presented. More importantly, none of these techniques consider communication between different tasks, which makes these techniques applicable to uni-processors only. In the rest of Section 3.1, we will study some of the newer work done in this research area that offer a complete solution to enable efficient execution of a task. The work also covers the topic of inter-task communication, which fits perfectly into SPM management on multi-/many-core processors.

Figure 5 shows the general flow of compiler-based SPM management approaches. Such an approach takes a regular program written for cache-based processors as input, together with library files that define runtime management functions, and performs analyses to generate the executable that can be run on SPM-based systems. Compiler-based SPM management mainly focuses on solving two problems—the management of private data for a single task on each core, and management of shared data between communicating tasks.

3.1.1. Stack Data Management: A call stack is a stack data structure that stores information about the active functions of a program, and is one of the most often accessed memory segments. The accesses to stack account for about 64% of the overall memory accesses in multimedia applications[39]. High-frequency accesses to the stack segment makes the design of stack data management critical for the runtime performance of applications in SMM architectures.

[40] and [41] first proposed a runtime stack data management scheme called Circular Stack Management (CSM) which operates at the level of function frames. The basic idea is to move a function frame from the main memory to the SPM when control flow goes to the corresponding function call, and evict stack frames that are not immediately used if there is not enough space. CSM views the stack space in SPM as a queue, and brings in stack frames in a First-In-First-Out manner. Figure 6 shows an example of how CSM works. In this example, there is a lack of stack space in the SPM at the function call from *F1* to *F2*, so CSM finds the oldest stack frame of *main*

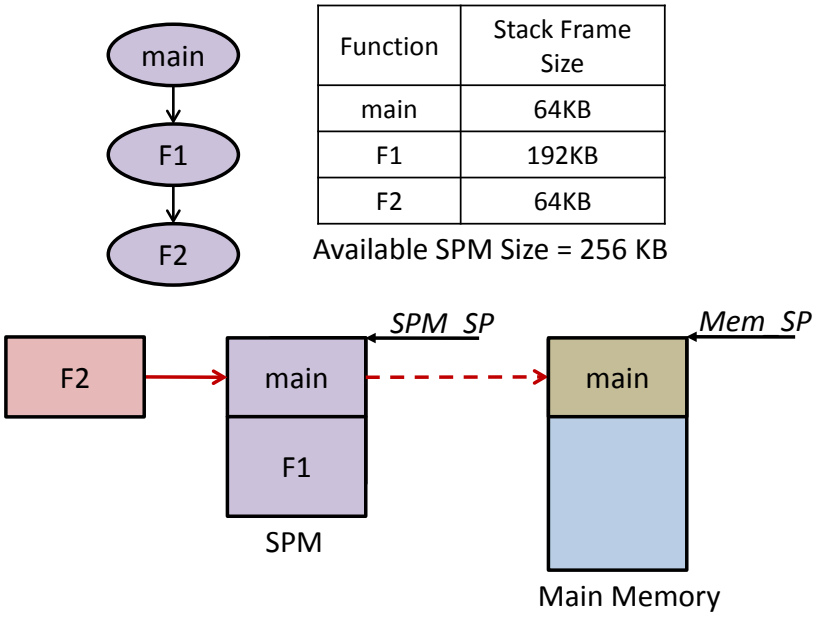

Fig. 6. *Introduction to the Circular Stack Management scheme (CSM). Given the SPM size and stack frame sizes, when* ***F1*** *calls* ***F2*** *function, the stack frames of* ***main*** *will be evicted to make space, following the FIFO rule of a queue. On the return of* ***F1****, the stack frames of* ***main*** *will be brought back in order.*

function and evicts it. Now that there is enough space after the eviction, CSM stops evicting more frames and brings in the stack frame of *F2*. When control flows returns from *F2*, *F1* is in the SPM, therefore no management is required. However, after the execution of *F1*, the stack frame of *main* has to be brought back from the main memory to the SPM.

While CSM is functionally correct, it introduces significant runtime overhead. [42] proposed a heuristic called Smart Stack Data Management (SSDM) to cut down the overhead of managing stack frames. The proposed approach takes in a call graph where each (function) node is annotated with its stack frame size, and models the problem of identifying the function calls that require stack frame management as placing *cuts* on the edges corresponding to these calls. For example, in the case of Figure 6, SSDM will place a cut on the edge corresponding the function call from *F1* to *F2*, since when *F2* is called, the runtime stack manager has to manage the stack space in SPM to accommodate the stack frame of *F2*. SSDM initially places a cut on each edge, and then goes through the call graph and merges edges along paths, with the constraints that the size of nodes between any two consecutive cuts along any path should not exceed the size of the stack segment in the SPM. This approach seeks placing no more than necessary cuts, and successfully reduces the overhead related to transferring stack frames between the SPM and the main memory. Figure 7 illustrates the difference of applying CSM and SSDM on the same call graph. For the same program, SSDM places significantly fewer cuts—in other words, less management—and therefore has much superior runtime performance for stack management. In particular, CSM can be viewed as an extreme case of SSDM which places a cut on every function call. Experimental results show that SSDM, even with the extra instruction overhead can outperform hardware caching.

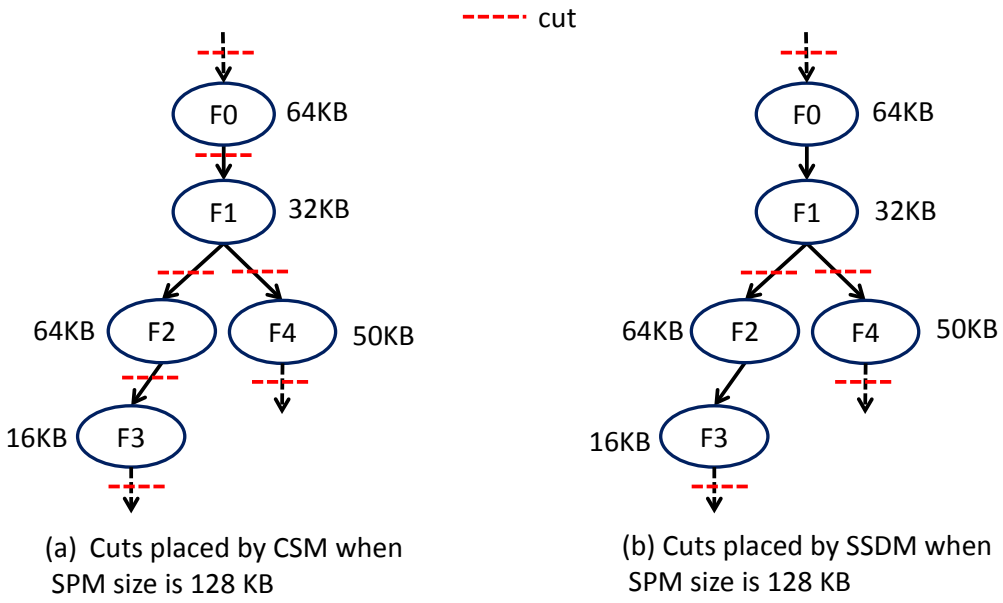

Fig. 7. *Comparison of management functions insertion by the Circular Stack Management (CSM) scheme and the Smart Stack Data Management (SSDM) scheme. While CSM conservatively places a* ***cut*** *on every function call, SSDM goes through the call graph and realizes some cuts are not necessary as SPM has enough space, e.g. between* ***F0*** *and* ***F1****, or between* ***F2*** *and* ***F3****.*

3.1.2. Heap Data Management: The existence of heap segment provides a way to dynamically allocate portions of memory to programs on demand, and free it for reuse when no longer needed. This is critical to any manycore system where more than a single process might be active at any time.

[43] first introduced the semi-automatic heap data management (SHDM) scheme for SPM-based multicore architectures. The work implements a runtime heap manager that transparently manages data transfers of heap objects between the SPM and the main memory. It acts like a fully associative software cache in SPM. When the runtime manager receives any request for allocating a heap object, it allocates space in the main memory, and returns the address of the allocated space which will be subsequently used to uniquely identify the heap object. The manager then checks and evicts existing heap objects from the SPM to the main memory based on Least Recently Used (LRU) policy if necessary, to make sure there will be enough space for the newly allocated object in the SPM. To access the heap object, the manager translates the main memory address to the SPM address, since it knows both. Finally, upon the reception of the request to deallocate this object, it simply frees the allocated main memory space. The runtime heap manager in SHDM is implemented as an API, and users are required to manually insert these API functions.

[44] proposes a fully-automatic heap data management (FHDM) scheme. This compiler-based heap-management approach is totally automated—users are not required to do anything more than compiling the applications, and the compiler will transform the applications properly. Proposed as an efficient enhancement to the SHDM approach, FHDM deploys a set-associative software cache with low associativity instead of a fully associative software cache. The new data structure saves the expensive table lookup at every heap access, and instead only compares the tags of the requested memory address with the blocks within the same set. In particular, tag comparison can

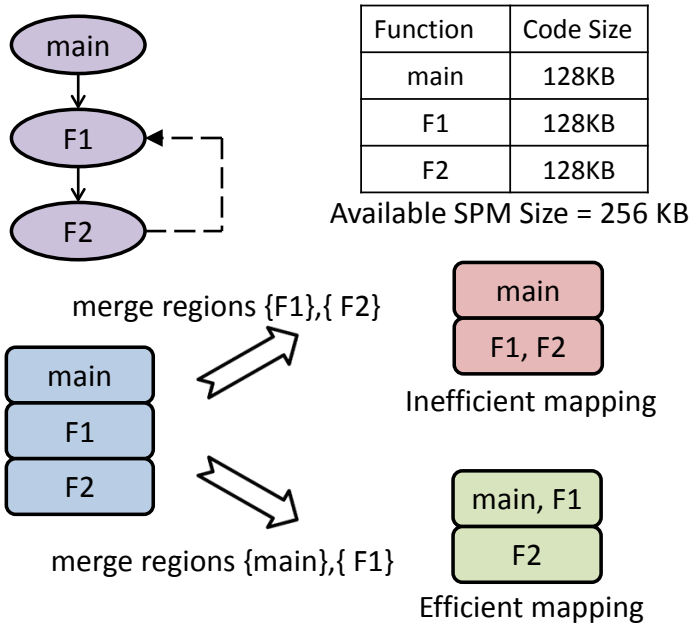

Fig. 8. *The general steps CMSM takes. The dashed line indicates* ***F2*** *is called in a loop at* ***F1****. CMSM initially places each function in a separate region, and merge regions* ***main*** *and* ***F1*** *to the same region to accommodate the SPM space. Function* ***F1*** *and* ***F2*** *are placed separately to avoid thrashing as the loop executes.*

be done in parallel if the architecture supports SIMD instructions, which is not uncommon in modern manycore processors. To further cut down the software overhead, FHDM uses a simple round-robin policy (to decide the victim to save software overhead), instead of the more expensive LRU used in SHDM. FHDM also builds a victim buffer as an optional performance improvement.

A subtle yet important benefit of FHDM is the fixed size of the data structure used for management. In both the SHDM and FHDM scheme, the data structure used to implement the software cache always resides in the SPM. In SHDM, the size of the data structure is proportional to the number of heap objects, so itself needs to be managed if it becomes large. On the other hand, in FHDM, once the number of sets, associativity, and the size of each data block are decided by the users, the size of the data structure is determined and unchanged during the execution. This guarantees constant SPM memory space usage and therefore will not cause memory overflow.

3.1.3. Code Management: On desktops or clusters with general purpose processing units, the system loads the program into the main memory and then executes it. Virtual memory enables a processor to load instructions on demand so that it can execute large programs that can not fit in the main memory. Such a memory management technique is critical to SPM management, which usually deals with limited SPM capacity. Traditionally, memory virtualization requires hardware support in the form of a memory management unit (MMU) to translate virtual memory to physical memory. Typically MMUs are not deployed in SPM-based manycore architectures to simplify the hardware and save power and thus this work has to be done in software.

A software code management technique typically divides the available SPM space into *regions*, and maps different functions into these regions [45]. Each function is mapped to exactly one region. At runtime, any function being called must be loaded to the region it is mapped to, and

later moved out to make space for calls to other functions that are going to be mapped to the same region. When the control flow returns, the runtime library will check if the caller function is available in SPM, and brings it back if it is not present in SPM.

Mapping conflicting functions (e.g. two functions when one function calls the other one frequently) to the same region of SPM would considerably degrade the performance of the application. The quality of mapping is decided by the way we estimate the cost of management overhead. We want to have a mapping that will minimize such overhead.

The previous models of code management cost are calculated statically—they do not correctly account for the interference caused by mapping a function into a region. The interference indicates the overhead related to the replacement of each other when two functions are mapped to the same region during execution, analogous to conflict misses in cache. This is measured by the total size of memory transfers introduced. Code mapping for Software Managed Manycores (CMSM) [45] first takes into consideration the interference to other functions every time it maps a function to a region while modeling the cost, by adjusting the cost function to reflect the interference dynamically while the mapping algorithm runs. It also exploits control flow information such as execution paths and branches. Overall, it improves the accuracy of the cost estimation and greatly reduces the management overhead.

CMSM starts by mapping each function to a separate region, and then greedily finds two regions that incur the maximum reduction on the management overhead after merging, until the sum of the remaining regions can fit into the available SPM space. Figure 8 illustrates CMSM using a simple example. Assume the available SPM space is 256 KB and there are three functions—*main*, *F1* and *F2*—in the program whose code sizes are all 128 KB. Function *main* calls function *F1*, which further calls function *F2* in a loop (indicated by the dashed line). Initially each function is placed in a separate region. Since the sizes of the three regions is larger than the available SPM, CMSM tries to merge regions. If the region $\{F1\}$ and $\{F2\}$ are merged, then every time *F1* calls *F2* or *F2* returns to *F1*, they need to replace the other function currently in the region, yielding an inefficient mapping. On the other hand, if function *F1* and *F2* are mapped into separate regions, then no conflicts due to competition for the same region will happen during the execution of these two functions. Therefore, CMSM will get two new regions with *F1* and *F2* being placed separately, i.e. $\{main, F1\}$ and $\{F2\}$. Now that the new regions (both are 128 KB) can fit into the SPM (256 KB), CMSM will stop at this point. More detailed examples and the underlying algorithm can be found in [45]. Experimental results demonstrate that even with the extra instructions, CMSM can deliver better performance than instruction caches.

3.1.4. Management of Inter-task Communication: Multiple tasks running on manycore processors may need to communicate for some use cases of manycore processors. For example, a large workload can be broken into smaller subproblems and distributed into multiple tasks so each task can solve a subset of the problem in parallel to yield the outcome more efficiently. In such an environment, it is sometimes unavoidable to have shared data, e.g. multiple cores may need to read and write to the same memory locations for communication. Any modifications to such shared memory have to be propagated to the other cores that subsequently access the same locations. In the presence of hardware cache coherence, the problem is automatically solved—the hardware logic will make sure the modification is publicized. However as mentioned earlier, cache coherence has been removed from recent multicore processors to simplify the hardware design and save power as well as area [4, 9, 5, 7]. Therefore, the consistency of the shared data has to be done by software.

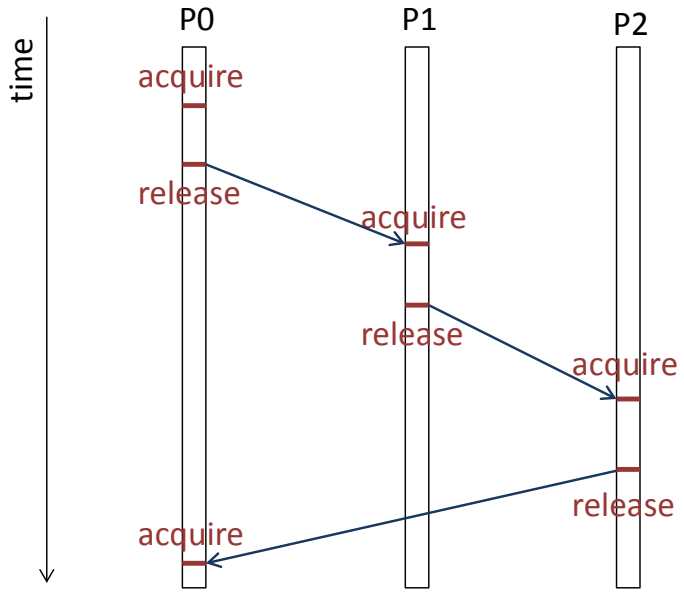

Fig. 9. *The general flow of the proposed byte-grain coherence management. Each core records its writes to shared memory locations, which is passed down to the subsequent cores that access the same memory locations.*

Designing an efficient coherence management is important for the overall system performance, as otherwise a poorly designed coherence management scheme may incur overwhelming communication and computation overheads that may exceed the gains from parallel computing. Traditional coherence designs for multiprocessor systems typically employ coarse-grain management granularity (e.g. page). Such techniques usually sacrifice computation for communication, which makes lots of sense in traditional multiprocessors with computationally powerful processors yet much slower inter-processor commutation [46]. However, in modern manycore processors, the computing power of each core is relatively weak compared to traditional processors, while the communication speed is much higher [47]. Therefore, these design choices must be reviewed and techniques modified for multi-/many-core processors.

A byte-grain coherence management technique [48] can be used for managing shared data in SPM-based manycore processors. Figure 9 shows the general flow of the approach. Each writing core first requests the access to shared memory locations. After obtaining the permission, the core starts to write to the shared memory locations exclusively, and record the modification. When the core is finished, the subsequent core takes over and reads the record of writes, and applies the change accordingly. To track the exact information of writes, a write notice [49] is maintained. Instead of recording the values related to the write, it records the description of the write, e.g. location, size. Write notices decide the granularity of coherence management: in a page-grain coherence management, a write notice records which page is modified, while in the byte-grain approach, a write notice records the starting address and the size of the write.

The main advantage of a byte-grain technique is the resulting savings in computation. To avoid false sharing caused by multiple writers simultaneously writing to the same page, a page-grain technique creates a duplicate of the page of interest, and then compares the duplicate with the modified copies sent back by the writers to extract and apply the differences. It is not hard to imagine that such comparisons will be computationally expensive. On the other hand, byte-grain

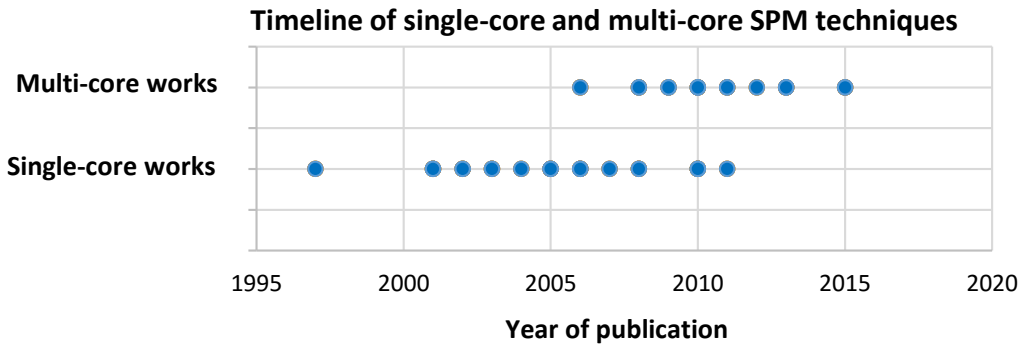

Fig. 10. *The trend of SPM management research is shifting from single-core processors to multi-/many-core processors.*

management is able to avoid such comparisons completely. Instead of maintaining coherence of a shared page, the byte-grain coherence management maintains the coherence of each write to shared data—e.g. a memory object, or a data member of a class instance in object-oriented languages—which zeros out the chance of false sharing.

3.2. Running Multiple Tasks On Each Core

Several research efforts have considered a different scenario where multiple tasks are simultaneously running on a single-core architecture and sharing a single physical SPM. [50] proposes a compile-time analysis approach to support concurrent execution of tasks sharing the same SPM resource. It assumes all working sets are known at compile time. To avoid the overhead of flushing the whole content of the SPM during context switches, it uses an integer linear programming formulation to find the best placement of data objects that minimizes the overlaps of the data object placement between subsequent context switches. [51] 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. [52] integrates a scratchpad memory manager into the operating system. In this work, after defining memory objects, a profit value is assigned to them by profiling the access patterns. At runtime, different heuristics and methods are used to change the SPM content after any context switch with the goal of having high SPM utilization for the set of currently active tasks. [53] proposed three different strategies to allocate SPM space for instructions—a spatial method that allocates each task its private space in the SPM; a temporal method that allows each task to use the entire SPM during its time slice; a hybrid method that combines the previous two methods.

3.3. Running Multiple Tasks on Multiple Cores

With the adoption of multi-/many-core platforms, the focus of SPM management research has shifted from single-core to manycore processors (Figure 10). As a result, a slew of adaptive approaches have been proposed to make allocation decisions at runtime for multi-core architectures with multiple tasks sharing and contending for SPM space. [54] specifies an ILP formulation that integrates task scheduling with SPM partitioning and allocation at compile time to produce both a schedule and static data allocation that optimize performance by profiling the set of tasks. [55]

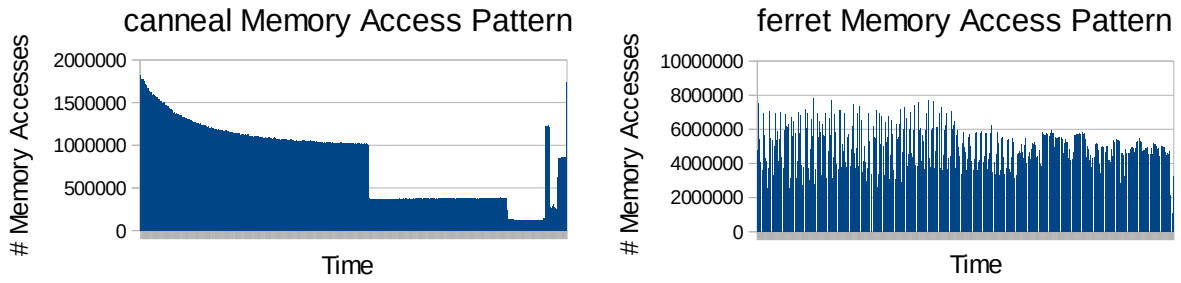

Fig. 11. *Number of memory accesses measured periodically for the execution of the canneal and ferret benchmarks.*

generates SPM allocation of arrays using static analysis at compile time for a fixed set of tasks executing on a MPSoC sharing distributed SPMs in order to both improve performance and minimize energy consumption in embedded systems. Similarly, [56] defines an OpenMP extension and compiler optimization to allocate parts of data arrays to distributed SPM for parallel programs executing on an MPSoC. They improve performance by locating data near the processing elements that access it most frequently. [57] 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. [58] outline another integrated approach to task scheduling and SPM allocation, but only using static analysis. [59] 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. [60] defines a framework that allows programmers to guide runtime decisions for allocating heap data to SPM in order to reduce energy consumption. [61] and [62] both propose hybrid memory hierarchies (i.e. caches + SPM) that support globally addressable and coherent address spaces.

3.3.1. Virtualization Strategies for SPMs in Manycores: SPM virtualization aims to design effective memory hierarchies for modern manycore platforms, and provide applications with the convenience of a transparently managed address space, while simultaneously using developers' guidance to mitigate the complexity of managing shared memory, as well as utilizing the non-uniform characteristics of the underlying hardware. These techniques are designed to wisely allocate SPM space among tasks in the multitasking environment, which manifest their complexity in the number of tasks that are concurrently executing and the variety in their resource utilization, especially with memories—utilization of data memory can vary not only between tasks in a workload, but also within a single task over the course of its execution (Figure 11).

When software-programmable on-chip memory is contained in an additional layer of virtualization, application developers can specify when and what data to store near the core executing its instructions without knowing the intimate details of the underlying memory architecture. The runtime software (as part of the operating system) can map the data to any physical location on- or off-chip. Additional address translations to this intermediate virtualization layer must be stored to enable accesses to the data in the address space. This can be done with a translation table that maps virtual addresses of the task executing on the core to intermediate physical addresses (IPA), which is essentially a physical address in on-chip SPM space.

In the SPM hierarchy outlined Figure 12, the Runtime Memory Manager can account for un-

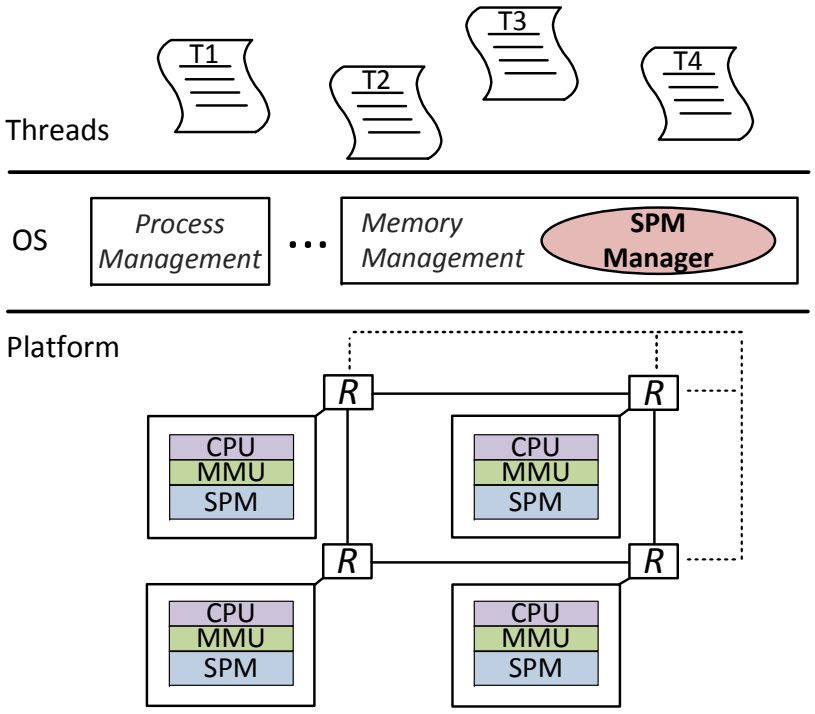

Fig. 12. *Example architecture with data SPMs controlled by a Runtime Memory Manager that monitors running tasks and performs SPM data mapping.*

predictable workloads and support oversubscription of SPM space. Additionally, we can expose characteristics of the memory banks to the runtime manager to further inform mapping decisions.

3.3.2. SPM Virtualization in Bus-based Manycores: [63] introduced the concept of virtual SPMs (vSPMs) that allows programmers to assume access to the entire on-chip memory space through virtualized address spaces. Each thread can therefore assume it has access to a dedicated contiguous memory (Figure 13 (b)) without considering interference from competing threads(Figure 13 (a)). A virtualization layer is responsible for determining what data is placed on-chip and what data is placed off-chip.

This layer can be implemented as a piece of software running at the operating system level as a kernel module (SoftSPMVisor) or as a hardware IP block (HardSPMVisor) acting like an arbiter that serves requests from its masters (processing cores) to the on-chip distributed SPMs. The SoftSPMVisor has the advantage of being flexible, portable (across various hardware configurations), and requires no extra hardware. However, it comes at the cost of higher power/performance overheads than the HardSPMVisor.

SPMVisor requires the programmer and/or compiler to define the priority of the data blocks. Moreover, specifying task-level priorities can help the allocation engine make runtime decisions for efficient on-chip resource utilization when many tasks contend for the limited physical SPM space.

SPMVisor provides programmers with APIs they can use to create vSPMs on-demand and delete them when they are no longer needed. Task-level priorities are also passed to the virtualization layer through the APIs. The virtualization layer receives all the vSPM creation requests from all threads, and based on the specified priorities, decides which will be mapped to on-chip SPM

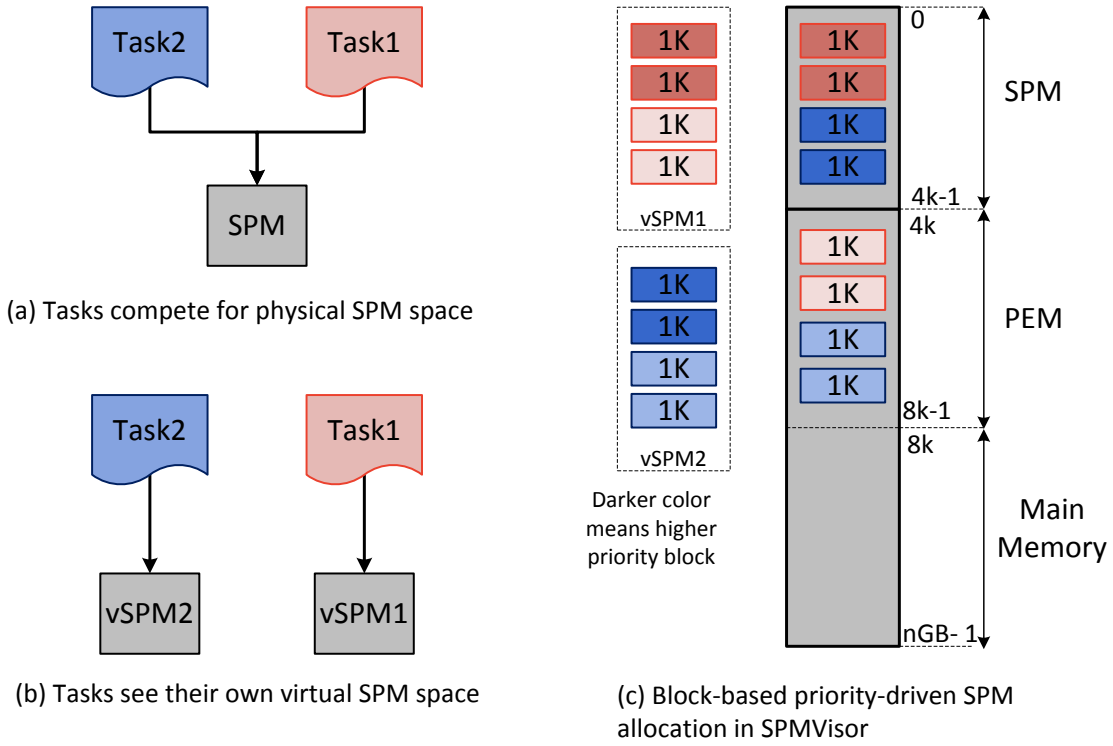

Fig. 13. *Virtualizing SPMs in SPMVisor approach*

and which will stay off-chip due to limited on-chip memory capacity.

In order to support the ability to virtualize more vSPMs than there are physical SPMs, [63] defines the notion of Protected Evict Memory (PEM) space. Since SPM space is very precious, SPMVisor utilizes block-based priorities in order to determine exactly what data should be copied to the on-chip memory. Other vSPMs will be mapped to the PEM in off-chip memory (Figure 13 (c)). A sample priority mechanism would be data utilization given by the ratio: (# of accesses to a block / cost of bringing the block to on-chip SPM). The utilization metric determines the impact of mapping a given block to SPM/PEM memory space on the energy consumption of the memory subsystem and the performance of the thread accessing that block. It is more beneficial to place blocks with high utilization (darker blocks in Figure 13 (c)) in the SPM space as this would yield better energy efficiency and performance.

Various policies can be defined for the SPMVisor that specify how to use priorities and make allocation decisions: first-fit, or fairness (e.g. Round Robin). Of course, the more complex the back-end allocation mechanism, the higher the overheads introduced into the system. More details can be found in [63].

Experimental results show that SPMVisor reduces execution time by 71% on average and reduces the on-chip memory power consumption by 79% on average.

[64] extends the idea of SPMVisor and presents a Variability-aware Memory Virtualization (VaMV) approach that allows programmers to exploit memory variability in order to reduce power consumption through memory virtualization, without exposing the underlying variability to the programmers. VaMV considers two types of SPM memories: 1) Normal SRAM and 2) Voltage-scaled SRAM including side-effects (e.g. process variations, higher access latency, etc.). The device signatures are available to VaMVVisor through sensing. VaMVVisor uses the device signa-

tures and prioritizes the memory resources according to their characteristics (e.g. power consumption), and selectively maps data to the best-fitting memory resource based on the defined policy (e.g. high-utilization data to low-power physical memory). The same approach can be applied to variability affected DRAM banks for off-chip memory as well. VaMV is capable of reducing dynamic power consumption by 63% on average while reducing total execution time by an average of 34% by exploiting both kinds of variabilities.

SPMVisor does not account for the different characteristics of the hybrid memory technologies consisting of SRAMs combined with emerging NVMs. HaVOC [65] extends the idea of SPMVisor and introduces the concept of virtual NVMs (vNVMs), which behave similarly to vSPMs, meaning that the runtime environment transparently allows each thread to manage its own set of vNVMs through a set of minimalistic APIs. Programmers (through annotations) and compilers (through static analysis) can then specify hybrid memory-aware mapping policies for their data/instruction blocks at compile-time. Each policy attempts to map data to virtual SPMs/NVMs. The HaVOC runtime system enforces these policies and decides how to best utilize the underlying memory resources in order to optimize SPM energy consumption. [65] also introduces a new metric called data volatility to facilitate simpler definition of mapping policies. Data volatility is defined as the write frequency of a piece of data over its accumulated lifetime. This metric is useful when deciding whether data is worth (cost effective) being mapped onto NVM. Highly volatile data implies that at some point the cost of keeping that data in NVM during its entire lifetime might be greater than leaving it in main memory. As a result, when two competing threads request NVM space, the estimated cost function (e.g. energy savings) will be used to prioritize allocation of on-chip space, while volatility can be used as a tie breaker and prediction metric of cost fluctuation. HaVOC is able to reduce execution time and energy by 60.8% and 74.7% respectively on a chip-multiprocessor with hybrid NVM/SPMs compared to traditional multitasking based SPM allocation policies.

3.3.3. SPM Virtualization in Large-Scale Manycores: The concept of storage clouds has been previously explored for data centers.[66] has adapted a similar concept to propose a possible solution (called SPMCloud) for managing software programmable memories in scalable manycore platforms. In the SPMCloud scheme, each task can access memories from a distributed memory subsystem through virtualization. On-demand allocation of virtual memories, introduced in SPMVisor, is adapted to manycore platforms and enhanced by priority-driven allocation of the software-programmable memory space.

SPMCloud uses a hierarchical approach to divide the entire platform into multiple regions. Two different enterprise-network-inspired configurations for SPMs are proposed: (1) embedded Network Attached Storage (eNAS), which provides a single standalone reliable on-chip memory subsystem for each region; and (2) embedded Storage Area Network (eSAN), which distributes memories to every single node of the region.

SPMCloud introduces a conceptual way of managing memory in manycore platforms. Experimental results on Mediabench/CHStone benchmarks show that by using SPMCloud's fully distributed memory subsystem, 48% energy savings and 70% latency reduction can be achieved on average over state-of-the-art NoC memory techniques.

SPMPool builds on this concept in order to demonstrate a detailed solution for dynamic management of SPM-based manycore platforms. In manycore platforms, cores are often under-utilized, as are the intra-core private on-chip memories. Furthermore, concurrently executing tasks exhibit high variability of memory intensity during their execution, which makes on-chip memory re-

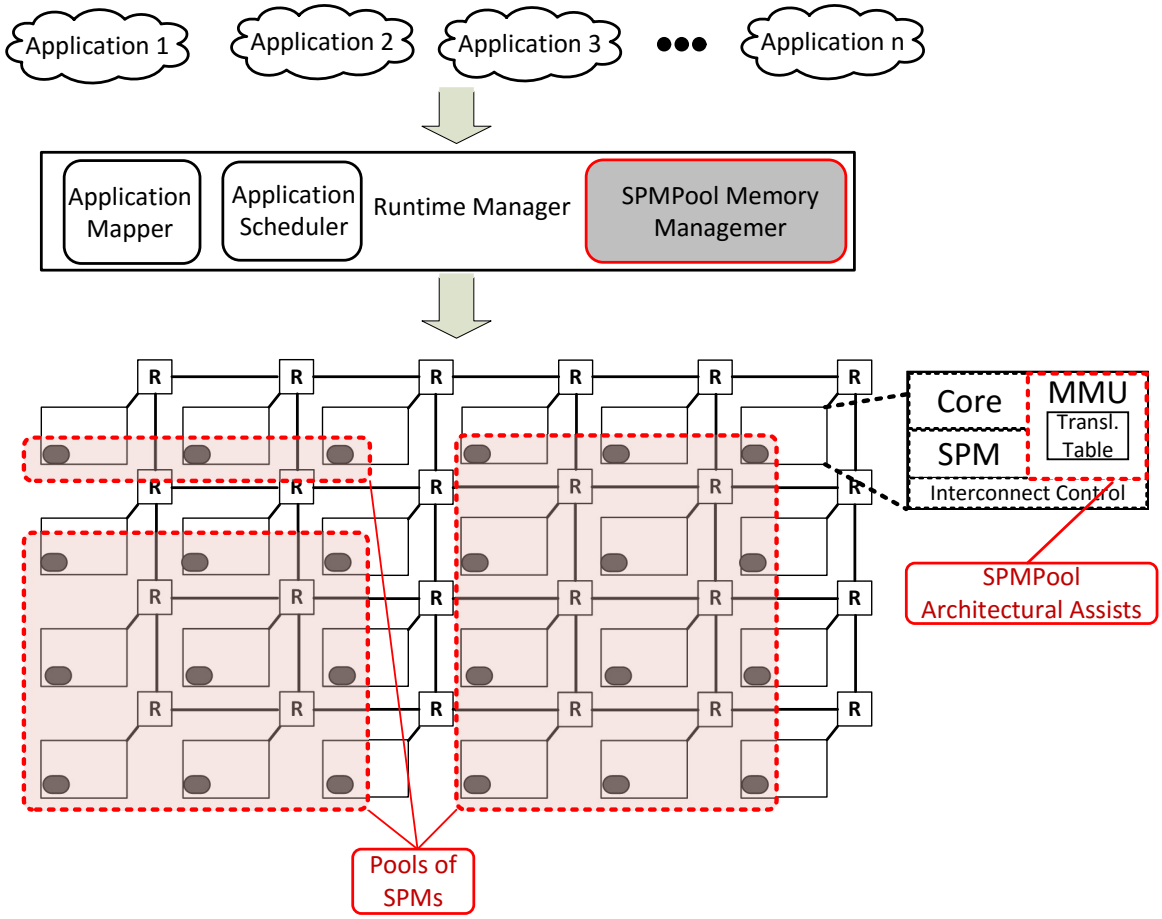

Fig. 14. *System-level view of SPMPool*

sources more valuable for some tasks than others. Limiting tasks to their local memory can often result in poor resource utilization.

To address these issues, [67] proposes SPMPool, which dynamically shares SPM resources between concurrently running tasks. Sharing is feasible through virtualization: each task has a private virtual address space and a runtime manager maps virtual addresses to physical on-chip memories by exploiting underutilized memory resources and adapting to the memory needs of workloads.

Figure 14 shows a system level view of SPMPool that consists of three components:

1) Pools of SPMs: in a tiled many core platform equipped with NoC, tasks inside a pool can share SPMs based on their dynamic memory requirements.

2) SPMPool Memory Manager: The SPMPool runtime memory manager is in charge of determining the data placement for the entire set of tasks running concurrently. The tasks are assumed to be profiled beforehand and all of the virtual memory pages that are accessed throughout the execution of that task are tagged with a weight metric. This weight metric is meant to represent the priority of each virtual page. The more a virtual page is accessed, the higher the calculated weight metric will be and hence the priority of that page during SPM allocation.

The memory manager is triggered anytime a new task enters the system or a task finishes its execution. Following the trigger, it updates its internal bookkeeping list of live memory pages as

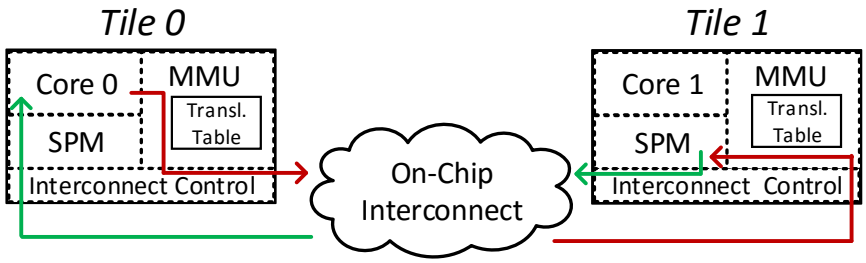

Fig. 15. *Remote SPM access in SPMPool*

well as their priorities. This list along with the task-to-core mappings can then be used to update the data placement. However, it is infeasible to find the optimal data placement at runtime due to time and memory constraints. To alleviate that, [67] proposes heuristics to find a near-optimal placement in a reasonable amount of time. The general intuition behind all of the presented heuristics is to keep the highest weighted pages on-chip and close to the core where they are accessed from.

3) SPMPool Architectural Assists: SPMPool enables tasks to address and access remote SPMs, as well as store information about the physical location of their data. Each tile has an augmented translation table that contains virtual memory to physical memory mapping information for any of the task's pages that are stored on-chip (Figure 15). If an entry for the page exists in the TLB, a local or remote SPM access is initiated. If a task's page is not in its local translation table, the memory access defaults to a standard main memory access. The MMU on each tile contains hardware to handle direct SPM access requests to and from remote tiles. Figure 15 illustrates a sample remote SPM access by an executing task.

For workloads with varying inter-application memory-intensity and configurations ranging from 16 to 256 cores, SPMPool can achieve up to 76% reduction in memory access latency compared to the approach that limits executing cores to use their local SPMs.

4. Conclusions and Future Work

Software Programmable Memories (SPMs) are those that—in contrast to caches—have to be explicitly controlled by the software through data movement instructions. Owing the simpler, power-efficient design, and promise of high scalability, these memories are being considered for modern manycore architectures. Since the SPMs have to be explicitly controlled in software, two kinds of capabilties must be added in the software: i) manage the data of a task on a single SPM, and ii) partition the data of multiple tasks on the multiple SPMs of the manycore system. This article surveyed a sampling of previous work on using SPMs in this context, and also discussed some new research developments in this direction. While the earlier work on managing task data on SPM proposed static techniques (where the mapping of the data on the SPM does not change), later work researched dynamic techniques that could further improve system performance by changing the data that is mapped to the SPM at runtime. Many early efforts suffered from limitations such as the inability to handle applications with pointers, and recursion. We outlined new techniques enabling efficient execution of any task on an SPM based core, that manage all task components (stack, heap, global and code) comprehensively, which also provide support for inter-task communication. With regard to the sharing of multiple SPMs in a many-core many-task environment, early research focused on the sharing of one SPM by multiple tasks executing on a core. Tech-

niques were then developed to spread the heap data among globally addressed SPM space. Recent research has attempted to solve the data distribution problem for multiple tasks executing on multiple cores, managing data on multiple SPMs through SPM virtualization. These runtime techniques are typically managed by a runtime monitor that operates at the higher level between applications and the operating system, where this monitor manages the space on the SPMs, and even promotes oversubscription.

While there is a lot of ongoing work aiming to highlight feasibility and advantages of an SPM based manycore system, much more is needed to demonstrate the feasibility and advantages of executing the entire software stack—including the system software—on a many-core SPM based architecture. Equally important are techniques that address the multi-dimensional constraints faced by emerging embedded systems: power, energy, resilience, aging, etc. The emergence of newer memory technologies and interconnection schemes provide both opportunities as well as challenges for the efficient design of SPM-based architectures. Finally, research needs to address the holistic combination of cache and SPM based manycore architectures, so that applications can seamlessly benefit from the inherent benefits of hardware caching as well as software-managed SPMs in an adaptive manner.

This work was partially supported by the NSF Variability Expedition award CCF-1029783, CCF 1055094 (CAREER), and CNS 1525855, and CCF-0916652. We would also like to acknowledge Dr. Ke Bai, and Dr. Luis Angel D. Bathen for their contributions.

5. References

[1] M. A. Heinrich, "The Performance and Scalability of Distributed Shared-memory Cache Coherence Protocols," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1999, aAI9924431.

[2] D. Abts, S. Scott, and D. J. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[3] T. Li and L. K. John, "ADir_pNB: A Cost-Effective Way to Implement Full Map Directory-Based Cache Coherence Protocols," *IEEE Trans. Comput.*, vol. 50, no. 9, pp. 921–934, Sep. 2001.

[4] Intel Lab, "The SCC Programmer's Guide," http://www.intel.com, Mar 2014.

[5] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the kalray mppa-256 integrated manycore processor," *Procedia Computer Science*, 2013.

[6] C. Ebert and C. Jones, "Embedded Software: Facts, Figures, and Future," *Computer*, vol. 42, no. 4, pp. 42–52, April 2009.

[7] B. Flachs, S. Asano, S. H.Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. Brokenshire, M. Peyravian, V. To, and E. Iwata, "The microarchitecture of the synergistic processor for a cell processor," *Solid-State Circuits, IEEE Journal of*, vol. 41, no. 1, pp. 63–70, Jan 2006.

[8] ARM, "ARM1176JZF-S Technical Reference Manual," http://infocenter.arm.com/, Jul 2004.

[9] Texas Instrument, "TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. E)," http://www.ti.com, Jan 2012.

[10] 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 *Proc. of CODES*, 2002.

[11] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," in *Proceedings of the 1997 European Conference on Design and Test*, ser. EDTC '97. IEEE Computer Society, 1997, pp. 7–.

[12] 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.

[13] O. Avissar, R. Barua, and D. Stewart, "Heterogeneous memory management for embedded systems," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[14] ——, "An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems," *Trans. on Embedded Computing Sys.*, vol. 1, no. 1, pp. 6–26, 2002.

[15] N. Nguyen, A. Dominguez, and R. Barua, "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, pp. 115–125.

[16] M. Verma, S. Steinke, and P. Marwedel, "Data Partitioning for Maximal Scratchpad Usage," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2003.

[17] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, 2002, p. 409.

[18] I. Puaut and C. Pais, "Scratchpad Memories vs Locked Caches in Hard Real-time Systems: A Quantitative Comparison," in *Proceedings of the Design, Automation Test in Europe Conference*, 2007.

[19] H. Wu, J. Xue, and S. Parameswaran, "Optimal WCET-aware Code Selection for Scratchpad Memory," in *Proceedings of the 10th ACM International Conference on Embedded Software*, 2010.

[20] Y. Kim, D. Broman, J. Cai, and A. Shrivastava, "WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores," in *In proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[21] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET Centric Data Allocation to Scratchpad Memory," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, 2005.

[22] Q. Wan, H. Wu, and J. Xue, "WCET-aware Data Selection and Allocation for Scratchpad Memory," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, 2012.

[23] J.-F. Deverge and I. Puaut, "WCET-Directed Dynamic Scratchpad Memory Allocation of Data," in *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, 2007.

[24] P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig, "Fast, Predictable and Low Energy Memory References Through Architecture-aware Compilation," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2004.

[25] S. Udayakumaran and R. Barua, "Compiler-decided Dynamic Memory Allocation for Scratch-pad Based Embedded Systems," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2003.

[26] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap Data Allocation to Scratch-pad Memory in Embedded Systems," *J. Embedded Comput.*, vol. 1, no. 4, pp. 521–540, 2005.

[27] L. Li, L. Gao, and J. Xue, "Memory Coloring: A Compiler Approach for Scratchpad Memory Management," in *Proc. of PACT*, 2005.

[28] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic Management of Scratch-Pad Memory Space," in *Proceedings of the Design Automation Conference*, 2001, pp. 690–695.

[29] J. Hu, C. Xue, Q. Zhuge, W.-C. Tseng, and E.-M. Sha, "Towards Energy Efficient Hybrid On-chip Scratch Pad Memory with Non-volatile Memory," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, 2011.

[30] P. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. Off-chip Memory: the Data Partitioning Problem in Embedded Processor-based Systems," pp. 682–704, 2000.

[31] F. Poletti, 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 Design Automation Conference*, 2004.

[32] Y. Ishitobi, T. Ishihara, and H. Yasuura, "Code and data placement for embedded processors with scratchpad and cache memories," *Signal Processing Systems*, vol. 60, no. 2, pp. 211–224, 2010.

[33] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions," *ACM TECS*, vol. 5, no. 2, pp. 472–511, 2006.

[34] M. Verma and P. Marwedel, "Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 8, pp. 802–815, Aug. 2006.

[35] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee, "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, M. Parashar, R. Badrinath, and V. Prasanna, Eds., 2008, vol. 5374, pp. 569–582.

[36] B. Egger, J. Lee, and H. Shin, "Dynamic Scratchpad Memory Management for Code in Portable Systems with an MMU," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 11:1–11:38, Jan. 2008.

[37] B. Egger, S. Kim, C. Jang, J. Lee, S. L. Min, and H. Shin, "Scratchpad Memory Management Techniques for Code in Embedded Systems Without an MMU," *IEEE Trans. Comput.*, vol. 59, no. 8, pp. 1047–1062, 2010.

[38] B. Egger, J. Lee, and H. Shin, "Scratchpad Memory Management in a Multitasking Environment," in *Proceedings of the 8th ACM International Conference on Embedded Software*, 2008.

[39] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the IEEE International Workshop on Workload Characterization*, 2001.

[40] A. Kannan, A. Shrivastava, A. Pabalkar, and J. Lee, "A Software Solution for Dynamic Stack Management on Scratchpad Memory," in *Proceedings of the Conference on Asia and South Pacific Design Automation*, 2009, pp. 612–617.

[41] K. Bai, A. Shrivastava, and S. Kudchadker, "Stack Data Management for Limited Local Memory (LLM) Multi-core Processors," in *Proceedings of the International Conference on Application Specific Systems, Architectures and Processors (ASAP)*, 2011, pp. 231–234.

[42] J. Lu, K. Bai, and A. Shrivastava, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)," in *Proceedings of the 50th Design Automation Conference (DAC)*, 2013.

[43] 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 (CODES+ISSS)*. New York, NY, USA: ACM Press, 2010, pp. 317–326, iSBN.

[44] ——, "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.

[45] K. Bai, J. Lu, A. Shrivastava, and B. Holton, "Cmsm: An efficient and effective code management for software managed multicores," in *Proceedings of the international symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[46] AMD, "HPC Processor Comparison," July 2012. [Online]. Available: http://sites.amd.com/us/Documents/49747D_HPC_Processor_Comparison_v3_July2012.pdf

[47] IBM Technical Library, "Cell Broadband Engine Architecture and its First Implementation." [Online]. Available: http://www.ibm.com/developerworks/power/library/pa-cellperf/

[48] J. Cai and A. Shrivastava, "Software Coherence Management on Non-coherent Cache Multicores," in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, Jan 2016, pp. 397–402.

[49] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[50] L. Gauthier, T. Ishihara, H. Takase, H. Tomiyama, and H. Takada, "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.

[51] 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*, 2004.

[52] R. Pyka, C. Faßbach, M. Verma, H. Falk, and P. Marwedel, "Operating System Integrated Energy Aware Scratchpad Allocation Strategies for Multiprocess Applications," in *Proceedingsof the 10th International Workshop on Software &Amp; Compilers for Embedded Systems*, 2007.

[53] H. Takase, H. Tomiyama, and H. Takada, "Partitioning and Allocation of Scratch-pad Memory for Priority-based Preemptive Multi-task Systems," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, 2010.

[54] V. Suhendra, C. Raghavan, and T. Mitra, "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.

[55] D. Cho, S. Pasricha, I. Issenin, N. Dutt, Y. Paek, and S. Ko, "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.

[56] A. Marongiu and L. Benini, "An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs," *Computers, IEEE Transactions on*, vol. 61, no. 2, pp. 222–236, 2012.

[57] V. Suhendra, A. Roychoudhury, and T. Mitra, "Scratchpad Allocation for Concurrent Embedded Software," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 4, pp. 13:1–13:47, Apr. 2010.

[58] L. Zhang, M. Qiu, W.-C. Tseng, and E.-M. Sha, "Variable Partitioning and Scheduling for MPSoC with Virtually Shared Scratch Pad Memory," *Journal of Signal Processing Systems*, vol. 58, no. 2, pp. 247–265, 2010.

[59] D. Cho, S. Pasricha, I. Issenin, N. Dutt, M. Ahn, and Y. Paek, "Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 4, pp. 554–567, 2009.

[60] N. Deng, W. Ji, J. Li, and Q. Zuo, "A Semi-automatic Scratchpad Memory Management Framework for CMP," in *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, 2011.

[61] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero, "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.

[62] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have Your Scratchpad and Cache It Too," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[63] L. A. D. Bathen, N. D. Dutt, D. Shin, and S.-S. Lim, "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.

[64] L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta, "VaMV: Variability-aware Memory Virtualization," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012.

[65] 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.

[66] 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.

[67] H. Tajik, B. Donyanavard, J. Jahn, J. Henkel, and N. Dutt, "SPMPool: Runtime SPM Management for Embedded Many-Cores," Center for Embedded Computer Systems, University of California, Irvine, Tech. Rep. CECS TR 14-08, July 2014. [Online]. Available: http://cecs.uci.edu/files/2014/07/CECS-TR-14-08.pdf