Virtualizing on-chip distributed ScratchPad memories for low power and trusted application execution

Luis Angel D. Bathen · Dongyun Shin · Sung-Soo Lim ·
Nikil D. Dutt

Received: 9 March 2012 / Accepted: 7 November 2012 / Published online: 10 January 2013
© The Author(s) 2013. This article is published with open access at Springerlink.com

Abstract Emerging multicore platforms are increasingly deploying distributed scratchpad memories to achieve lower energy and area together with higher predictability; but this requires transparent and efficient software management of these critical resources. In this paper, we introduce the concept of ScratchPad Memory virtualization, a hardware/software run-time layer (called SPMVisor) that virtualizes the scratchpad memory space in order to facilitate the use of distributed SPMs in an efficient, transparent and secure manner. We introduce the notion of virtual scratchpad memories (vSPMs), which can be dynamically created and managed as regular SPMs. The SPMVisor exploits policy-driven allocation strategies based on application privilege levels and data level prioritization metrics (e.g., utilization) to efficiently manage the on-chip memory real-estate. Our experimental results on Mediabench/CHStone benchmarks running on various Chip-Multiprocessor configurations and software stacks (RTOS, virtualization, secure execution) showed that SPMVisor enhances performance by 71 % on average and reduces power consumption by 79 % on average with respect to traditional context switching schemes. We showed the benefits of using vSPMs in a various environments (a RTOS multi-tasking environment, a virtualization environment, and a trusted execution environment). Furthermore, we explored the effects of mapping instructions and data onto vSPMs, and showed that sharing on-chip space reduces both execution time and energy by an average 16 % and 12 % respectively. We then compared our priority-driven memory allocation scheme with traditional dynamic allocation

---

L.A.D. Bathen (✉) · N.D. Dutt
Center for Embedded Computer Systems, University of California, Irvine, Irvine, CA, USA
e-mail: lbathen@uci.edu

N.D. Dutt
e-mail: dutt@uci.edu

D. Shin · S.-S. Lim
School of Computer Science, Kookmin University, Seoul, South Korea

D. Shin
e-mail: elsdy@kookmin.ac.kr

S.-S. Lim
e-mail: sslim@kookmin.ac.kr

and showed an average 54 % execution time reduction and 65 % energy savings. Finally, to further validate the SPMVisor's benefits, we modified the initial bus-based architecture to include a mesh-based CMP with up to 4×4 nodes. We were able to observe that SPMVisor's priority-driven allocator was able to reduce execution time by an average 17 % with respect to competing allocation policies, while saving an average 65 % across various architectural configurations. We were also able to observe that SPMVisor reduces execution time by an average 12.6 % with respect to competing allocation policies, while saving an average 63.5 % in total energy for various architectural configuration running 1024 jobs.

Keywords Information assurance · Security · Chip-multiprocessors · Policy · Scratchpad memory · Virtualization · Embedded systems

1 Introduction

The ever increasing demands of the embedded system software stack, limitations in the uniprocessor domain [1], and technology scaling have pushed for the move towards multiprocessor technology ([20, 22, 40]). A byproduct of the multicore phenomena is the rapid integration of distributed scratchpad memories (SPMs) into the memory hierarchy [22] due to their increased predictability, reduced area and power consumption [5]. Moreover, the adoption of multicore platforms further motivate the need for multi-task environments, where system resources such as SPMs need to be shared. Sharing of the SPMs is a critical task as they tend to hold critical data (commonly used data, sensitive data, etc.), and it has been shown that efficient SPM utilization leads to great energy savings and power consumption [5, 10, 23, 31, 41].

Traditional approaches assume that a given application is granted full access to the underlying resources [4, 5, 10, 23, 24, 31, 36, 41], however, in multi-tasking environments such approaches will not work as the state of the system (applications running, memory requirements) will vary. Techniques for sharing the on-chip SPMs have been proposed [16, 37, 38], however, they assume that the applications are known ahead of time. These assumptions were true for closed systems, but as open systems (e.g., Android [17]) start to be widely adopted, what programs are loaded onto the device will not be necessarily known at compile time.

Deploying open environments comes at a price; with the ability to download and run pre-compiled applications, combined with greater on-chip resources, and the ability to share resources opens the door to new threats (e.g., side channel attacks [43]) that were not present in the uniprocessor domain, much less in closed systems. As a result, any one of these vulnerabilities may lead the system to (a) run a malicious application that tries to access sensitive data via software exploits (e.g., buffer overflows [11]), or (b) expose private information via side channel attacks [43]. Virtualization has been proposed as a possible solution to the ever growing threats in open systems, where virtual machine (VM) instances with various privilege levels may run different software stacks [19]; however, such approaches do not address the problem of on-chip memory management.

In this paper, we introduce the concept of *SPMVisor*, a hardware/software layer that virtualizes the scratchpad memory space in order to facilitate the use of distributed SPMs in an efficient, transparent and secure manner. To provide dynamic distributed SPM memory allocation support to any application that is installed in our system (e.g., downloaded applications, launching of VM instances), we introduce the notion of virtual scratchpad memories (vSPMs), which can be dynamically created and managed as regular SPMs. To protect the

on-chip memory space, the SPMVisor supports vSPM-level and block-level access control lists. Finally, in order to efficiently manage the on-chip real-estate, our SPMVisor supports policy-driven allocation strategies based on privilege levels. Our experimental results on Mediabench/CHStone benchmarks running on various Chip-Multiprocessor configurations and software stacks (RTOS, virtualization, secure execution) showed that SPMVisor enhances performance by 71 % on average and reduces power consumption by 79 % on average with respect to traditional context switching schemes. We showed the benefits of using vSPMs in a various environments (a RTOS multi-tasking environment, a virtualization environment, and a trusted execution environment). Furthermore, we explored the effects of mapping instructions and data onto vSPMs, and showed that sharing on-chip space reduces both execution time and energy by an average 16 % and 12 % respectively. We then compared our priority-driven memory allocation scheme with traditional dynamic allocation and showed an average 54 % execution time reduction and 65 % energy savings. Finally, to further validate the SPMVisor's benefits, we modified the initial bus-based architecture to include a mesh-based CMP with up to 4×4 nodes. We were able to observe that SPMVisor's priority-driven allocator was able to reduce execution time by an average 17 % with respect to competing allocation policies, while saving an average 65 % across various architectural configurations. We were also able to observe that SPMVisor reduces execution time by an average 12.6 % with respect to competing allocation policies, while saving an average 63.5 % in total energy for various architectural configuration running 1024 jobs. We also varied the SPM's sizes and were able to see that the SPMVisor is able to reduce execution time by an average 17.7 % with respect to the *Spatial* allocation policy, while saving an average 43.7 % in total energy.

The rest of the paper is organized as follows: Sect. 2 will go over the motivation for our work and the contributions of this paper. Section 3 will give the reader a high level overview of SPMVisor, the target platform and our assumptions. Section 4 will discuss the our ScratchPad Memory (SPM) virtualization layer, which consists of the concept of virtual SPMs, the run-time system (SPMVisor), a programming API that allows programmers to manage the on-chip memory space, and the isolation and security mechanisms of the SPMVisor. Section 5 will briefly go over the integration of our scheme with the OS/Hypervisor layers. Section 6 will discuss the differences between our work and state of the art work and how our scheme can complement existing SPM management approaches. Finally, Sect. 7 will discuss the experimental evaluation of our scheme.

2 Motivation

Two major issues motivate our work: (1) The challenge of providing dynamic distributed SPM memory allocation/de-allocation support in the presence of a multi-tasking environment. (2) The need for on-chip virtualization support for virtualized environments and on-chip memory isolation to support trusted application execution.

2.1 Shared SPMs in multi-tasking environments

Figure 1 shows a set of applications being executed (*App1-App4*) by two CPUs (*CPU0*, *CPU1*) utilizing a total of two SPMs (4 KB space each) with temporal and spatial allocation (hybrid allocation) [38] and pre-defined schedules. Such schemes work well in closed systems, however, it might not be feasible to predict all the combinations of all the applications that will be running concurrently in an open system (e.g., Android).

Fig. 1 Optimal data placement is not guaranteed in open environments (e.g., Android based systems)

Consider the case where a high-priority application is launched (*App5* denoted by the dotted box in Fig. 1); two schemes can be used: (1) Flush the contents of the SPM and grant full SPM access to *App5*, or (2) Strictly adhere to the static placement and map contents for *App5* to off-chip memory as shown in Fig. 1. Assume that *App5* is a critical application with real-time requirements; then mapping the application's data to off-chip memory might not be the best approach as the overhead due to the off-chip accesses might lead it to miss its deadlines. This worsens when designers wish to map their application's instructions to the on-chip SPMs, given that it has been shown that mapping instructions to SPMs may result in greater performance improvements than mapping data alone [4, 13, 14, 33]. As a result, there is a need for dynamic allocation of SPM space considering the priority of the applications in a heterogenous multi-tasking system.

2.2 Trusted application execution

Trusted application execution is needed in an open environment as trusted applications that process sensitive data (e.g., mobile banking) share the same system/hardware resources as untrusted applications. In order to provide a trusted environment various schemes can be deployed: (1) The use of virtualization to isolate resources and run applications inside their own VM instances [19]. (2) The use of small Trusted Computing Bases (TCBs) with dynamic trusted environment generation based on halting the system and granting full system access to the application [29] as shown in Fig. 2.

To the best of our knowledge, there is no support for on-chip distributed SPM virtualization, so we can either flush all SPM contents and grant full SPM access to the VM instance running *App5* or we can follow the approach presented in [29] and flush the contents of all the executing tasks from the SPMs (*App1-4*), halt the execution of all processes, and grant full system access to *App5*. These approaches incur high power and performance overheads due to the flushing of the SPM contents, hence, in order to provide trusted execution for a given application, there is a need for protecting the on-chip memory resources so that they are not tampered with and guarantee data confidentiality while considering both power and performance.

Fig. 2 Halting all executing processes and flushing SPM contents in order to provide a trusted execution environment

2.3 Transparency for upper layers of software stack

Our goal is to provide an efficient (dynamic, high performance, low power, secure) resource management layer that supports multi-tasking environments and trusted application execution. In order for programmers to adopt our approach there is a critical need for transparency as there is extensive work addressing both SPM management (static and dynamic) [4, 5, 10, 23, 24, 31, 36, 41] as well as scheduling for SPM enabled systems [36, 38]. As a result, we intend to provide developers with a set of high-level APIs (Sect. 4.2) that will allow them to create virtual SPMs (Sect. 4.1), and manage them just like they would manage traditional SPMs.

2.4 Contributions

In this paper we introduce the concept of SPMVisor, a hardware/software layer that allows us to virtualize the on-chip resources (SPMs). This paper's contributions are:

- The concept of virtual ScratchPad Memories (vSPM), allowing software programmers logical and transparent access to SPM resources
- A dynamic and efficient resource-management mechanism built on the idea of policy-driven allocation (based on data/application criticality)
- API for dynamic and transparent on-chip resource management
- Tradeoff analysis between mapping application's data and instructions to on-chip virtual SPMs

To the best of our knowledge, our work is the first to introduce the concept of ScratchPad Memory (SPM) virtualization and policy-driven dynamic allocation for safe, performance-driven, and energy efficient use of on-chip memory resources.

3 SPMVisor overview

Figure 3(a) shows the problem with the traditional SPM-programming model, where traditionally, programmers assume that they have full access to the on-chip memory space. As

Fig. 3 Virtualizing SPMs

a result, if the system wishes to run multiple applications with SPM support, there will be contention for the physical SPM space (Fig. 3(a)). Figure 3(b) shows a high level overview of our proposal; we introduce the concept of virtual (v) SPMs (Sect. 4.1) that allow programmers to assume full access to the on-chip memory space (through virtualized address spaces). By doing so, each application is able to see its own SPM rather than competing with other SPMs for space. Meanwhile, the virtualization layer dynamically prioritizing what data goes on-chip (Fig. 3(c), Sect. 4.4) and what data goes off-chip, to protected memory space (*PEM*—Sect. 4.3).

3.1 Software/hardware virtualization support tradeoff

The SPM virtualization layer can be implemented as a software layer running at the hypervisor/OS level (*SoftSPMVisor*) or as a hardware IP block similar to an arbiter (*HardSPMVisor*). The *SoftSPMVisor* layer should be light-weight, flexible, and modularized in a manner that allows for easy integration into existing OSes/Hypervisors. The *HardSPMVisor* module should have minimal area overheads, and support a simplified API for transparent use by the programmers or the OS/Hypervisor software stacks. The *SoftSPMVisor* has the benefit of being flexible, portable (across various hardware configurations) and requires no extra hardware (except a secure DMA/ability to lock part of off-chip memory). The benefits of the *SoftSPMVisor* comes at the cost of higher power/performance overheads than the *HardSPMVisor*. Ideally, both *SoftSPMVisor* and *HardSPMVisor* should support the same minimal API and should require minimal changes in the programming model. For this paper, we will focus on the *HardSPMVisor*, and leave the *SoftSPMVisor* as future work. Our end goal is to have a tightly coupled SW/HW layer that exploits the benefits of both *SoftSPMVisor* and *HardSPMVisor*. For the remainder of this paper we will refer to the *HardSPMVisor* as SPMVisor.

3.2 Target platform

Figure 4 shows the high-level diagram of our target platform. Our Chip-Multiprocessor (CMP) resembles the platform used in [6, 7, 36], which consists of a set of OpenRISC-

Fig. 4 SPMVisor enhanced chip-multiprocessor

like cores, distributed SPMs, an AMBA AHB [2] on-chip bus, enhanced with a secure DMA (S-DMA), and a cryptographic engine (Crypto) similar to the ones in [21, 28]. The SPMVisor module behaves as an enhanced arbiter that serves requests from its masters (CPUs) to the on-chip distributed SPMs. The CMP can run a simulated RTOS or a simulated microvisor-like [30] virtualization environment.

3.3 Assumptions

We assume that the programmer and/or compiler can statically (or dynamically) define the priority of the data-blocks. Priority can be defined via various metrics: (1) utilization (e.g., number of accesses/cost of bringing data to SPM), (2) confidentiality (e.g., a crypto key value), (3) real-time requirements (e.g., deadline-driven), etc. For this manuscript we assume we have soft real-time requirements since we are in the process of extending our work to support hard real-time systems. As a result, we will not go over real-time analysis at this point (e.g., the deadline-driven allocation metric).

We assume the programmer can partition their application's address space and map their data and instructions to SPMs (similar to the Cell processor programming model) in order to exploit our scheme. For some instances we disable the instruction cache and map instruction blocks to on-chip SPM or off-chip memory. Similarly, we disable the data cache, and map data to SPM space or off-chip memory depending on the allocation policy.

Our S-DMA supports locking of part of main memory to be used as Protected Eviction Memory (*PEM*), which serves as temporal storage for data that the SPMVisor is unable to fit in SPM space.

Finally, we assume the existence of a trusted third party where applications may be downloaded from and installed on the system. The application developers are trusted, and priorities are not exploited (e.g., making all data high priority unnecessarily); thus even if all applications and data sets have the same priority, our approach will behave similarly to traditional context-switching (e.g., RTOS) approaches. This assumption is reasonable as the trustworthiness of trusted sources such as application markets (e.g., Google Play, The App Store, etc.) is beyond the scope of this paper.

4 SPMVisor: ScratchPad memory virtualization

4.1 Virtual SPM (vSPM)

Virtual ScratchPad Memories (vSPMs) are introduced to provide software programmers a transparent view of the on-chip memory space. vSPMs can be created on-demand and

Table 1 vSPM management API

Method	Parameter	Type	Note
<i>v_spm_create</i>	PID	uint	Process ID
	AppPriority	uint	Application priority
	IPA	uint*	Intermediate Physical Address
	ACL	uint	vSPM level access control list
<i>v_spm_delete</i>	PID	uint	
	IPA	uint	
<i>v_spm_malloc</i>	PID	uint	
	IPA	uint	
	BlkSize	uint	Block size (B)
	BlkPriority	uint	Priority of this block
	MallocType	uint	Type of allocation
	BlkACL	uint	Block access control list
<i>v_spm_blk_deleeteete</i>	PID	uint	
	IPA	uint	
<i>v_spm_clear</i>	PID	uint	
	IPA	uint	
<i>v_spm_poll</i>	PID	uint	
	IPA	uint	
<i>v_spm_transfer</i>	PID	uint	
	SrcAddr	uint	Transfer source address
	DstAddr	uint	Transfer destination address
	TxType	uint	Transaction type sync/async/secure

deleted when no longer needed. Table 1 shows a subset of our API, which allow programmers to use vSPMs with minimal changes to their applications. We briefly describe a subset of the methods and their parameters. First, in order to specify the need for a vSPM, a programmer needs to create it, via the *v_spm_create* method, where the PID refers to the process/task ID of the application or process trying to request SPM space. AppPriority refers to the application's priority, which helps our allocation engine make run-time decisions for efficient on-chip resource utilization as we assume an environment consisting of applications with various requirements (soft real-time, security, reliability, etc.).

This work exploits the notion of IPAs, which stands for *intermediate physical address*, and are used to address vSPM space. The idea is that the CPU/hypervisor can still use traditional virtual address (VA) to physical address (PA) translation, where the PA coming out of the CPU refers to the IPA addressing SPMVisor space. The real SPM PA is then obtained by the IPA to PA translation done inside the SPMVisor.

vSPM block allocation can be achieved through the *v_spm_malloc* method, which allows the programmer to specify the priority of the block so that SPMVisor can dynamically choose whether to grant this block SPM space or map it to off-chip memory (or Protected Evict Memory). Priorities can also be defined for a given application by setting the AppPriority field when creating (*v_spm_create*) the vSPM or for a given block within a vSPM by setting the BlkPriority field (*v_spm_malloc*). The MallocType entry refers to

synchronous/asynchronous allocation or if the data block requires extra protection such as encryption. vSPMs support content deletion, meaning that programmers may want to zero the contents of a vSPM for security without deleting the vSPM. It is possible to delete a vSPM via the `v_spm_delete` method, which depending on the vSPM priority, may zero all contents in the physical blocks for security, and then de-allocate the blocks belonging to the vSPM. The `v_spm_blk_delete` method allows for single-block deletion, which allows us to dynamically create and delete blocks. The `v_spm_poll` method can be used to monitor the status of an asynchronous transaction such as asynchronous `v_spm_malloc`, asynchronous `v_spm_transfer`, etc. It is possible to have two types of protection mechanisms for each vSPM: (1) vSPM-level defined at creation (`v_spm_create`) by setting the vSPM's access control list (ACL). (2) Block-level by defining the BlkACL during allocation (`v_spm_malloc`). Finally, the `v_spm_transfer` method allows for (sync/async) secure or regular transfers between physical SPM space and off-chip memory using SPMVisor and its secure DMA engine (S-DMA). The default size of the vSPM blocks is set to be the same size of a mini-page (1 KB), the idea is to allow existing SPM management techniques that work with page tables to still use our vSPMs. Again, our goal is to provide a lightweight virtualization layer for SPMs, while allowing for existing SPM management techniques to work without much change to their programming models.

To support legacy code that operates over physical SPMs, our approach allows the OS to create vSPMs prior to launching an application. The application gets the SPM's physical address offset from a base register (accessed through legacy APIs). In this case, the OS does not have to report the physical address of the SPM, rather, it only needs to report the IPA obtained from the SPMVisor when the vSPM for the given application was created. When the application finishes its execution, the OS must release the on-chip memory resources by deleting the vSPM. In this mode, the entire vSPM space is pre-allocated, and the AppPriority is used to determine where to map the application's blocks. For instance, if the application being launched has hard real-time requirements, then the SPMVisor may map all of the application's vSPM blocks to physical SPMs rather than mapping them to *PEM* space.

4.2 vSPM programming model

Function 1 shows the traditional programming model for SPMs. We first see in Line 5 the assignment of the offsets for the target SPM, this offset is used throughout the code. We program the DMA engine and request it to transfer the contents of the `zigzag_idx` buffer to the SPM offset (Line 9). We then wait for DMA to complete the transfer (Line 11), we do the same for the `input_matrix` buffer.

We also point our pointers (`zz`, `imatrix`, `omatrix` as shown in Lines 12, 19, and 22 respectively) to the SPM offsets holding the data we want to access. We then execute our kernel as shown in Lines 24 through 25. Function 2 shows the vSPM programming model, which depicts the *minimal* changes needed to use our vSPMs. First, we create the vSPM as shown in Function 2 Lines 8 through 9, we provide the method with the process ID, the application's priority, the pointer that will hold the vSPM IPA, and set the ACL for the vSPM. Lines 13–14 show the vSPM block allocation call, where we pass the process ID, the IPA (stored in `m_offset`), the block size in Bytes, the priority which is the same as the application in this case, the allocation type as blocking/synchronous, and the block ACL, which is set to the same value as the vSPM. Once the vSPM has been created and the block to be used has been allocated, we can then proceed to use the vSPM as a traditional SPM as shown in Lines 16 through 34, where the same source code is executed as in Function 1.

Figure 5 shows the memory view of a newly created vSPM (vSPM 3), after invocation to the `v_spm_create` method (Fig. 5(a)), we obtain the IPA for the vSPM from the SPMVisor.

```

void i_zig_zag () {
    int i;
    int *omatrix, *imatrix, *zz;
    unsigned int m_offset = SPMBASEADDR;

    // point zig zag matrix to SPM
    m_lock = 0;
    status = init_dma_put(get_pid(),
        &zigzag_idx, m_offset);
    wait_dma_complete(&m_lock);
    zz = m_offset;

    // point input matrix to SPM
    m_lock = 0;
    status = init_dma_put(get_pid(),
        &input_matrix, m_offset + 64 * sizeof(int));
    wait_dma_complete(&m_lock);
    imatrix = m_offset + 64 * sizeof(int);

    // point output matrix to SPM
    omatrix = m_offset + 128 * sizeof(int);

    for (i = 0; i < DCTSIZE2; i++)
        *(omatrix++) = *(imatrix + *(zz + i));
    ...

```

Function 1 Traditional programming model for SPM based systems

```

void i_zig_zag () {
    int i;
    int *omatrix, *imatrix, *zz;
    unsigned int m_offset;

    // create v spm
    v_spm_create(get_pid(), MIN_PRIO,
        &m_offset, ((get_pid() << 6) | RW_ACL));

    // allocate the block with min priority,
    // same acl as vspm and synchronous allocation
    v_spm_malloc(get_pid(), m_offset,
        1024, MIN_PRIO, SYNC_REG, V_SPM_DEF);

    // point zig zag matrix to vSPM
    m_lock = 0;
    status = init_dma_put(get_pid(),
        &zigzag_idx, m_offset);
    wait_dma_complete(&m_lock);
    zz = m_offset;

    // point input matrix to vSPM
    m_lock = 0;
    status = init_dma_put(get_pid(),
        &input_matrix, m_offset + 64 * sizeof(int));
    wait_dma_complete(&m_lock);
    imatrix = m_offset + 64 * sizeof(int);

    // point output matrix to SPM
    omatrix = m_offset + 128 * sizeof(int);

    for (i = 0; i < DCTSIZE2; i++)
        *(omatrix++) = *(imatrix + *(zz + i));
    ...

```

Function 2 vSPM programming model

Fig. 5 Process of SPMVisor vSPM creation/allocation with view of memory space

The various blocks within vSPM space are mapped to different physical blocks (SPM or PEM) based on their priority as shown by the dashed arrows from SPMVisor space to SPM/PEM space. We then proceed to allocate the block as shown in Function 2, where the checkered block is mapped to PEM space (dashed arrow from SPMVisor address space to PEM) as it was given low priority and there aren't enough on-chip resources to hold on to the content (Fig. 5(b)). Note that the black block pointed by SPMVisor refers to the configuration memory that serves as storage for the metadata needed by the vSPMs and their blocks. Finally, we proceed to use the vSPM by pointing our various pointers to the vSPM memory regions (via IPAs) in a transparent manner, since the users are oblivious to exactly where the data is mapped (Fig. 5(c)). Figure 5 shows a high level view of the IPA and its breakdown. Programmers do not have to worry about IPA, as the back-end (SPMVisor) decodes the IPA, and extracts the vSPM offset (14-bits) which is used to point to the physical block being accessed, as well as the Byte Offset (10-bits), which address data within a block.

4.3 Protected evict memory (PEM)

In order to extend the ability to virtualize more vSPMs than there are physical SPMs, we define the notion of Protected Evict Memory (PEM) space. Since SPM space is very precious, we exploit the idea of block based priorities in order to determine exactly what data goes on-chip and what data can be mapped to off-chip. 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. Blocks with large utilizations are better off being placed in SPM space as this would yield better energy and performance. PEM space is protected by locking the memory space and restricting access to it. The only master that should be able to access PEM space is the SPMVisor, and thus, any attempt to access it by any other master

Fig. 6 Data-driven priority allocation

would trigger an invalid access flag. *PEM* access control can be implemented by the secure DMA or the arbiter where the ACL for *PEM* contains the hardware ID (HW_ID) of the SPMVisor, and is validated against it. We assume that the HW_ID cannot be spoofed.

4.4 Policy driven allocation

It is possible to have various allocation policies for the on-chip resources, be it based on priorities, first-fit, or fairness (e.g., Round Robin). In this paper, we will focus on two types: (1) Data-driven, where data blocks may have individual priorities, hence, allowing the SPMVisor to decide at run-time where each block should be mapped. (2) Application-driven, where each application has real-time requirements or needs trusted execution, etc., and SPMVisor decides how to allocate physical resources for entire vSPMs. The main difference between the two approaches is the granularity and guarantees each offer.

The data-driven approach has block level priorities and block-level ACLs, thereby allowing for various degrees of performance/protection for each of the vSPM blocks. This is very useful in case a programmer wants to define memory regions within his/her vSPM with different performance/protection requirements. The application-driven approach has vSPM level ACL and vSPM level priority, this is useful when the programmer may want have dedicated space of a given type. vSPM level policies are given much higher priority as they reflect the criticality of the application using them. The block-level/vSPM-level priorities allow us to efficiently utilize the on-chip real-estate.

Traditional approaches [15] do not take application/data priority and are thus unable to allocate SPM space to an application once the entire SPMs are fully allocated, leading to energy inefficiencies and performance degradation. Our allocation engine currently supports fixed-block allocation, however, it is possible to use variable-block allocation and exploit some of the concepts introduced in [15]. Of course, the more complex the back-end allocation, the higher the overheads introduced into the system, so we must be careful when deciding which allocation mechanism to use.

Figure 6 shows the same sequence of tasks being executed as in Fig. 1, where a new critical task is introduced into the system with high priority (dotted block). The main difference is the diagram which shows the status of the memories as vSPMs are created, and blocks are allocated (States *S1* through *S6*). On arrival of the first application (*App3*), the vSPM is

Fig. 7 Application-driven priority allocation

created and the SPMVisor maps the *App3*'s blocks to *SPM0*, and the process continues up until *S3*. When *App4* arrives, the SPMVisor looks at the priorities of the blocks belonging to *App4* and decides to map them to *PEM* space. When *App5* (dotted block) needs to execute, rather than evicting all of *App1* and *App2*'s contents from SPM, the SPMVisor looks at the priorities of the various blocks, and makes the decision to evict *some* of the lower priority blocks from SPM space (*App1*-*3*), and allocating the space to *App5* blocks as shown in *S5*. After *App5* completes and destroys the vSPM, the SPMVisor then re-loads the contents it had evicted prior to *App5*'s execution. This example shows how our data-driven allocation policy works, as blocks may have different priorities (*P1*–*P3*) and its possible that applications with lower priority may have higher priority blocks than applications with higher priority. This is useful when an application such as audio playback can request SPM space because it will greatly benefit from it, whereas an image processing application with higher priority may not benefit as much from the SPM space.

Figure 7 depicts the various states undergone by the applications and their contents when following our application-driven allocation policy. Just like in Fig. 2, *App5* requires trusted execution, but in our case, rather than halting all processes and flushing the contents of the on-chip memory, we exploit the benefits of our vSPMs and their ability to isolate address spaces and enforce access control lists. States *S1* through *S4* go through the same allocation process as in Fig. 6. When *App5* is loaded, it is to be executed by *CPU0* in isolation, and a secure vSPM is created, where the ACL and priority is at the vSPM granularity rather than block based. As we can see, *S5* depicts the state of the memory space after allocating the vSPM for *App5*. Notice that vSPM is given highest priority, and as a result, its blocks have priority *P1*, whereas the blocks for the other vSPMs have lower priorities (*P2*–*P3*). Both *App1* and *App3* data blocks have the same priority (*P2*), however, *App1* is lower priority than *App3*, and will not benefit as much from holding the SPM space (since it will not run while *App5* runs), therefore, *App1*'s blocks are evicted from SPM space.

4.5 vSPM data protection

For this work, we assume that on-chip SPM is trusted/secure and can store sensitive data in plain-text. Any piece of sensitive data placed in off-chip memory must be encrypted.

Fig. 8 Data protection schemes

Figure 8(a) shows the traditional approach (Full Encryption) for protecting sensitive information, where transactions between on-chip and off-chip must undergo encryption/decryption and transactions between CPU and SPM space can be assumed to be secure (e.g., no tampering). Figure 8(b) shows configuration #1 of SPMVisor, where we assume that any transaction between CPU and SPMVisor is secure, and any transaction between on-chip and off-chip memory space must undergo encryption/decryption (Partial Encryption). Note that even data mapped to *PEM* space (denoted as a shaded box next to MM) must also undergo encryption/decryption. The communication between SPMVisor and *PEM* incurs high performance and power overheads due to the encryption/decryption steps each transaction must undergo. In order to reduce this overhead, our approach (Fig. 8(c)) makes use of secure DMA (S-DMA) which locks part of main memory (referred to as *PEM* space), and grants full access to only one master in the system, in our case, the SPMVisor. This allows us to bypass the extra encryption/decryption transactions we would have to perform when transferring data between SPMVisor and *PEM* space. Of course, any piece of data that is mapped to off-chip (not in *PEM* space) will still have to go through the encryption/decryption step.

Our vSPMs allow programmer to protect their memory space and exploit on-chip access control lists in order to guarantee data confidentiality. Moreover, side-channel attacks that monitor the memory subsystem by selectively evicting data for other tasks are unable to do so as vSPM content can only be evicted by SPMVisor. Applications that require trusted execution (e.g., SHA) may exploit application-driven allocation, and thus their data may not be evicted at run-time by another task. Finally, it is possible to provide data obfuscation by switching between SPMVisor data protection schemes (use of S-DMA and Partial Encryption) in a randomized manner, thereby reducing the chances of an attacker deriving any side information from the application at the cost of both power and performance overheads.

4.6 SPMVisor hardware module

Figure 9 shows a high level block diagram of our SPMVisor, which includes its configuration memory, which holds the metadata for vSPMs and their blocks. SPMVisor provides a vSPM address space of (2^{14}); however, the number of vSPMs is limited by the block size used (can be 64 B, 128 B, 256 B, 512 B, 1024 B, 4 KB, etc.), the total amount of physical memory managed by the SPMVisor (# SPMs and *PEM* space), and the amount configuration memory storage. Each block metadata requires 8 Bytes, and is stored in SPMVisor's configuration memory (can store between 512 B and 256 KB). So there is a tradeoff between the block size and the total size of the SPMVisor's configuration memory. In short, to minimize fragmentation we have decided to do block-based mapping, so the allocation is done via block bit-maps. The more blocks to keep track the more configuration memory is

Fig. 9 Block diagram of SPMVisor module

Table 2 SPMVisor block metadata

PID	HW_ID	S	Priority	ACL_ptr
12	6	1	4	9
IPA	PA	On/Off	Sec. Setting	Status
14	13	1	1	3

necessary. One way to reduce the necessary storage would be to do first-fit allocation (e.g., continuous blocks) [7], but this may lead to fragmentation.

The breakdown of the block metadata is shown in Table 2. The *HW_ID* flag refers to the owner of this block, and is used to validate any changes to the metadata. The *S* bit is used to determine if there is a need to protect the block and the *ACL_ptr* is used to validate the transaction in case the *HW_ID* of the request is not the block's owner. Each ACL entry has capacity for up to four (*HW_ID:rights*) entries. The *On/Off* bit is used to decide which offset (SPM or *PEM* base address) is used to translate IPA address PA. The *Priority* field is used to prioritize the given block and set at allocation time. The *Sec. Settings* field is used to decide whether to block's ACL should be enforced. Finally, the *Status* field is used when asynchronous methods are used to monitor the status of the block creation, deletion, etc.

On every read/write transaction, the SPMVisor (Fig. 9) will fetch the corresponding block's metadata, and validate it against the ACL. Based on the address, we decide if it is a control transaction or an access transaction (read/write). If it is an access transaction, then after ACL validation and IPA/PA translation (through the address translation layer), the SPMVisor performs a slave transaction to SPM or *PEM* space (SLV_RD or SLV_WR). If the transaction is a control transaction, then depending on whether it is a vSPM creation/deletion or block creation/deletion, we invoke the Allocator/Evictor or the De-allocator modules. The Allocator/Evictor and De-allocator blocks have access to SPMVisor's secure DMA interface, which is used to transfer data between SPMVisor and main memory or *PEM* space.

4.7 Mapping data and instruction to vSPMs

Traditional SPM allocation schemes have focused primarily on mapping data to SPMs [16, 26, 31, 36–38, 41], however, it has been shown that mapping instructions and data (e.g., functions, heap/stack management) onto SPMs yields better performance and energy utilization than mapping data alone [4, 24]. As a result, the SPMVisor's APIs allow programmers

Fig. 10 Full application sandboxing

to prioritize what to bring onto SPM space regardless of whether it is data or instruction blocks. To map instruction blocks onto vSPMs, just like in the Cell processor, programmers must load the blocks onto a vSPM, which requires the creation of the vSPM prior to the execution of the new task/thread. Once the instruction blocks have been mapped to the vSPM, the new thread may start execution with its instruction memory base pointer mapped to the task's vSPM (given by the IPA provided by the SPMVisor).

Programmers need to be careful and choose the right allocation policies and priority metrics to be used and apply them accordingly (be it block level or application level, and utilization or real-time, etc.). The SPMVisor is capable of allowing programmers to partition their vSPM space into two level of performance guarantees; for instance, instructions are accessed every cycle, whereas data may be accessed every other few cycles, so programmers may need high-speed on-chip memory for their instructions to increase throughput, while highly-utilized data may be mapped to high-speed on-chip memory or off-chip memory (*PEM* space).

Our work serves as a framework for programmers to study how to efficiently map their application's data and instructions. Most state-of-the-art schemes would assume that the given application has full access to the physical resources, and do not consider what happens when more than one application try to run simultaneously while competing for SPM resources. In [8] we primarily focused on mapping data onto vSPMs, in Sect. 7.7 we will study this tradeoff and explore the effects of mapping both data and instructions to vSPMs.

This type of tradeoff is particularly important as we move away from applications with soft real-time requirements (this work) and move towards supporting applications with hard real-time requirements, where timing is critical, and designers need to map their applications so that their timing is predictable and their deadlines are met. The SPMVisor is capable of supporting the same predictable timing of SPMs (e.g., single-cycle address translation and predictable on-chip memory access timing) for applications that need it. Of course, this will affect how the applications running in the rest of the system run as on-chip memory space is very limited and very precious.

4.8 On-chip application sandboxing

By supporting the mapping of both data and instructions onto vSPMs we provide support for full on-chip application sandboxing (as shown in Fig. 10), where the application is capable of executing in complete isolation from the outside world. Moreover, to prevent any timing information leakage, the SPMVisor can support time-sharing of the bus (e.g., TDMA-based

arbitration). Though this might affect the available bandwidth (e.g., with respect to round robin or static priority based arbitration), the system will allow various sandboxes to execute in the same CMP, similar to the isolated execution (cell-vault) feature in the cell processor [35]. Though in this case, we would need to evaluate two things: (1) the context switching window and (2) the time sharing window (TDMA period).

5 OS/hypervisor integration

The OS/Hypervisor Layers can benefit from exploiting vSPMs since one benefit of using the SPMVisor is much lower context switching times. Since our focus in this manuscript is the introduction of the SPMVisor and use the *HardSPMVisor* as a proof-of-concept implementation, we will not give details on the *SoftSPMVisor* implementation and how it can be integrated into a hypervisor/OS layer. Instead, we will briefly discuss some of the key benefits of using vSPMs and *HardSPMVisor* in a virtualized environment. To the best of our knowledge, we are the first to introduce a virtualization layer and virtualization support for on-chip distributed memories, so the hypervisor needs to do two things when a context switch at the application level or OS level happens: (1) flush absolutely all contents from SPM space thereby incurring high overheads, or (2) keep page tables for SPM data, and flush the contents only for the preempted application (or applications in the case of OS level context switching). Thus, the benefits of our approach are: First, the hypervisor/OS does not have to worry about managing the on-chip memories as the SPMVisor will handle it, all the OS/hypervisor layers have to do is make calls for vSPM creation/deletion or block updates; Second, on a context switch (application level or OS level), the hypervisor does not need to flush SPM contents as long as vSPMs are used and vSPM IPAs are used to address SPM content (e.g., the application's page tables that keep track of SPM data use IPA instead of PA).

Our approach may co-exist with cache-based memories as well since SPM memory space is traditionally physically addressed. In our case, the SPMVisor provides a similar style of access. The main reason why we have IPAs and not VAs is because IPAs are exposed to the application as if they were real physical addresses much like SPMs and we wanted to clearly separate the concept of IPAs from traditional virtual memory management.

6 Related work

SPMs have through the years become a critical component of the memory hierarchy [4, 24], and are expected to be the memories of choice for future many-core platforms (e.g., [20, 22, 40]). Unlike cache-based platforms where data is dynamically loaded into the cache with hopes of some degree of reuse due to access locality, SPM based systems depend completely on the compiler to determine what data to load. Placement of data onto memory is often done statically by the compiler through static analysis or application profiling, the location of data is known a priori which increases the predictability of the system. Panda et al. [31] profiled the application and tried to allocate all scalar variables onto the SPMs. They identified candidate arrays for placement onto the SPMs based on the number of accesses to the arrays and their sizes. Verma et al. [41] look at an application's arrays, and identify candidates for splitting with the end goal of finding an optimal split point in order to map the most commonly used area of the array to SPM. Kandemir et al. [26] use loop transformation techniques such as tiling to improve data locality in loop nests with array accesses, and map

array sections to different levels in the memory hierarchy. Issenin et al. [23] proposed a data reuse analysis technique for uniprocessor and multiprocessor systems that statically analyses the affine index expressions of arrays in loop nests in order to find data reuse patterns. They derive buffer sizes to hold these reused data sets, and could be implemented on the available SPMs in the memory hierarchy. Suhendra et al. [36] proposed an ILP formulation to find out the optimal placement of data onto SPMs. Jung et al. [24] looked at dynamic code mapping through static analysis in order to maximize SPM utilization for functions. Bai et al. [4] looked at heap-data management for SPMs. Shalan et al. [34] looked at dynamic memory management for global memory through the use of a hardware module.

In order to support concurrent execution of tasks sharing the same SPM resources [16, 37, 38] propose various static analysis approaches, which assume all working sets are known at compile time. Poletti et al. [15] proposed a memory manager that supports dynamic allocation of SPM space, which supports block-based allocation (fixed and variable). Verma et al. [42] proposed three different SPM sharing strategies to support application mapping on MPSoCs (e.g., temporal, spatial, and hybrids). Egger et al. proposed SPM management techniques for MMU supported [13] and MMU-less embedded systems [14], where code was divided into cacheable code and pageable (SPM) code, and the most commonly used code is mapped onto SPM space. Pyka et al. [33] introduced an OS-level management layer that exploited hints from static analysis at run-time to dynamically map objects onto SPMs.

Our approach is different from [15] in that we exploit policy driven allocation mechanisms (Application or Data), which allows us to better utilize the SPM space and the Dynamic Memory Manager (DMM) presented in [15] stops allocating SPM space to an application as soon as the SPM space is fully allocated. Our work is different from other SPM management schemes in that we provide a means for transparent dynamic allocation of the SPM space. To the best of our knowledge, we are the first to propose a virtualization layer for on-chip distributed memories that supports true multi-tasking. Since transparency is one of our main goals, rather than competing with existing approaches, our work can be complemented by many of the existing SPM data allocation schemes [16, 26, 31, 33, 36–38, 41], where the static allocation policies can be used by our run-time system as hints, and our dynamic allocator can decide how to best enforce them. Our allocation engine can even exploit some of the allocation mechanisms presented in [15]. Since using our vSPMs require little effort on the programmer's end, techniques such as [4, 13, 14, 24] can exploit vSPMs for code/function/instruction management. Consider the work introduced by [13, 14], on a context switch, the page table information mapped onto SPM space would have to be flushed, where as if they mapped the content to vSPMs, the content would remain despite the context switch.

Our scheme is different from [13, 14] in that we allow programmers to explicitly (though hints) tell the SPMVisor what data to prioritize and map to on-chip memory space, moreover, the scheme in [13, 14] targets primarily single SPMs. Similarly, the Cell [22] processor supports transparent management of data transfers between local stores and main memory through DMA instructions, however, each SPE runs a single thread, and does not support true multi-tasking (e.g., PPC tells the SPE what to do, and on completion, the SPE tells the PPC it is done).

Finally, our scheme differs from [16, 36–38, 42] in that we do not assume the number of applications running on the system are fixed.

7 Experimental evaluation

7.1 Experimental goals

Our goal is to show that the benefits of our approach greatly offset the overheads introduced by our virtualization layer. First, we show the overheads of our approach in an ideal world where resources are not an issue. Second, we show the benefits our approach in a multi-tasking environment. Third, we wanted to show the benefits of using vSPMs in a virtualized environment running a light weight hypervisor. Fourth, we show the benefits of using SPMVisor in order to provide a trusted environment for secure software execution. Fifth, we explore the effects of mapping instructions and data onto vSPMs. Sixth, we explore the effect of varying the context-switching window in the scheduler on the SPMVisor. Sixth, we explore the effects of the task priority in the SPMVisor's allocation decisions. Finally, to further validate the benefits of the SPMVisor, starting with Sect. 7.10, we change the architectural template from a bus-based CMP to a Network-on-Chip [9]-based Mesh architecture, which is used to evaluate different allocation policies under different circumstances: (1) Mesh size, (2) Workload size, and (3) Memory size.

7.2 Experimental setup

Figure 11 shows our experimental setup where we implemented our SPMVisor module as a SystemC TLM/CCATB [32] block and integrated it into our simulation framework [7], which interfaces with SimpleScalar [3] and CACTI [39]. Table 3 shows in more detail the different parameters of our simulated architectural template.

We assume 65 nm process technology for our memories and a 512 MB off-chip main memory. We cross-compiled a set of applications from the CHStone [18] and Mediabench II [27] benchmark suites and analyzed them to obtain SPM mappable data sets. The applications we used are (ADPCM, AES, BLOWFISH, GSM, H.263, JPEG, MOTION, and SHA) as they are representatives of various security, multimedia, and communication applications

Fig. 11 Experimental setup

present in mobile systems. In order to support multiple applications running concurrently we used page tables (1 KB mini-pages).

The application's virtual addresses are translated by the CPU's MMU unit and generates physical addresses which point to physical SPMs, or intermediate physical addresses which then point to vSPMs. Our environment can simulate a lightweight RTOS environment with context switching enabled as well as a light-weight hypervisor. Each OS/CPU instance can run anywhere between 1–4 OSeS and 1–8 applications.

7.3 SPMVisor overheads

Figure 12 shows the power and latency overheads due to our virtualization layer. For this experiment, we compiled and ran each of the eight applications on a single CPU using a single 8 KB SPM. On average we see 14 % power consumption increase and 13 % latency (execution time) increase. This is due to the fact that in an ideal world, after the VA/PA conversion done by MMU, the CPU read/write to the SPM using the PA with no additional noise (e.g., waiting for arbiter, contention at the SPM, etc.). In our case, in order to access vSPM data, each transaction goes through the MMU (like in the base case) and performs the VA/IPA translation. We then use the IPA to talk to the SPMVisor, look up the vSPM's metadata, do a second level address translation IPA/PA, and make the request to the physical SPM (after validating the request against the ACL if needed). We expect to see much higher overheads if the virtualization layer is implemented in software, so it is important to have hardware support in order to minimize the virtualization overheads.

Fig. 12 Overheads in an ideal world

Table 3 Simulation configuration parameters

Core Type	Frequency	Pipeline Stages	Issue	Order
OpenRISC 1200	250 MHz	5	Single	In-Order
Address Translation Cycles	Arbitration	Bus Frequency	Bus Cycle Count	On/Off-Chip Memory Access Latency
1	STATIC	250 MHz	4–16	1 (on-chip)/80 (off-chip)

Fig. 13 RTOS SPMVisor savings for various configurations

Table 4 Virtualized environment configurations

Configuration	# of Applications	# of Guest OSes
e1	2	1
e2	4	2
e3	8	2
e4	8	4

7.4 SPMVisor support for heterogenous multi-tasking environments

Figure 13 shows the energy savings and latency reductions for a set of configurations: (a) 8 Apps/CPU, 1 × CPU, and 1 × 16 KB SPM (1 CPU × 16 KB). (b) 4 Apps/CPU, 2 × CPUs, and 2 × 16 KB SPMs (2 CPU × 16 KB). (c) 2 Apps/CPU, 4 × CPUs, and 4 × 16 KB SPMs (4 CPU × 16 KB). We use the same configurations for 8 KB and 16 KB SPMs. We compare two approaches: (1) base case which consists of a lightweight RTOs with context-switching enabled and (2) our approach which exploits the idea of the SPMVisor. As we can see in Fig. 13, our approach achieves 85 % power consumption reduction and 82 % latency reduction for a CMP with 16 KB distributed SPMs and 77 % power consumption reduction and 72 % latency reduction for a CMP with 8 KB distributed SPMs. For this experiment we use *Data-driven* priority allocation and use data utilization as a metric for priority. We set the context-switching window to 100K instructions (rather than cycles to keep the comparison consistent). The reason for the savings is that the context-switch requires swapping the contents (flushing the page tables) of the SPM being used by the CPU. As a result, the context-switch with no SPM virtualization (vSPMs) incurs high power/performance overheads due to the flushing/loading of the SPM contents. The % reduction is smaller for the 16 KB SPMs than the 8 KB SPMs because the amount of contention for SPM space between the applications is lower.

7.5 Benefits of vSPMs in a virtualized environment

For this experiment we wanted to show the benefits of using vSPMs in a virtualized environment. Table 4 shows a set of configurations where we varied the number of Applications per Guest OSes, and the number of Guest OSes. In the virtualized environment, we assume application context switch costs and OS context switch costs similar to the ones presented

Fig. 14 Virtualized environment running on CMP with vSPM support

Fig. 15 Comparison between various security schemes and their normalized energy utilization and latencies

in [12]. The baseline approach assumes that on every OS/Application context switch, the contents of the SPM where the OS/Application is running are flushed. The benefits of our approach (reduced latency and better energy utilization) are depicted in Fig. 14, where each of the configurations shown in Table 4 were run on top of varying number of CPUs. For this experiment we assumed that each CPU could access 8 KB of SPM space (and 8 KB of vSPM space) and kept page tables for SPM mapped data in order to flush only contents belonging to the preempted application(s). On average we see 76 % reduction in energy utilization and 49 % in latency across all the configurations. As a result, a hypervisor exploiting our vSPMs will have much lower context switch time.

7.6 Performance comparison among various secure approaches

Figure 15 shows the energy efficiency and performance comparison between the three schemes discussed in Sect. 4.5. The *Halt* approach evicts the contents of all processes from SPM space and halts all tasks, thereby granting full access to the underlying hardware to the given architecture. The *Encryption* scheme refers to the SPMVisor approach with no Secure-DMA support (e.g., no protection of main-memory (*PEM* space) via locking of the address space). In the *Encryption* scheme, any request addressing any a vSPM block mapped to *PEM* space needs to go through the encryption/decryption process, thereby introducing extra power consumption/latency overheads. The *S-DMA* scheme refers to SPMVisor with

Fig. 16 Comparison between mapping instructions only, data only, and a mix of the two onto vSPMs

Secure-DMA support, which locks part of the main-memory and grants restrictive access to SPMVisor. In the X-axis we have three different platforms in the following format: (# of applications per CPU $\times$ # of CPUs $\times$ size of SPM assigned to each CPU), so $2 \times 2 \times 8$ means two applications per CPU (total of 2 CPUs) and 2×8 KB SPMs. We ran up to 8 applications concurrently with multi-tasking enabled. We defined SHA, AES, and BLOWFISH as secure applications which require trusted execution. For this experiment we used *Application-driven* priority allocation of blocks, so SHA, AES and BLOWFISH were given the highest priority when allocating SPM space. We set the context-switching window to 100K instructions same as 7.4. As expected, SPMVisor with *S-DMA* support provides much better performance and energy utilization than the halt approach as it does not have to halt processes nor evict data every time a trusted application needs to run (77 % and 81 % on average). The reason why SPMVisor with *S-DMA* and SPMVisor with full *Encryption* are close to each other is that any data mapped by the compiler to off-chip memory will be encrypted/decrypted and any data loaded/flushed to/from SPM is also encrypted/decrypted. The main difference in the schemes is when the SPMVisor maps vSPM blocks to *PEM* space. In this case, any access to a *PEM* mapped vSPM block will have to be decrypted/encrypted by the *Encryption* scheme, whereas the *S-DMA* does not have to go through the extra encryption/decryption step. As a result, the *S-DMA* scheme will always outperform (be more efficient than) the *Encryption* scheme. In this experiment we do not observe much difference between the *S-DMA* and *Encryption* schemes because the SPMVisor exploited *Application-driven* priority allocation, hence, for the trusted applications, there were very few blocks mapped to *PEM* space.

7.7 Effects of mapping instructions and data onto vSPMs

Like in [38], we partitioned the application into its data (e.g., global arrays, local arrays, etc.), and instruction/code components. We then evaluate the difference between mapping data and instructions onto vSPMs. In the case where only data (DATA label in Fig. 16) was mapped to on-chip SPMs, instructions were all mapped to off-chip memory (e.g., no cache). Similarly, in the case where only instructions (INS label in Fig. 16) were mapped to on-chip SPMs, data elements were mapped to off-chip memory. The mixed workload (MIX label in Fig. 16) mapped both data and instruction onto SPM space (most commonly used data and instruction blocks/functions).

As we can see in Fig. 16, mapping instructions only (INS) to vSPMs greatly reduces the energy and execution time of the system for both cases (8 applications using 2 CPUs (4×2) and 8 applications using 4 CPUs (2×4). We observe similar behavior as in other scenarios where we vary the number of applications and CPUs in the system, resulting in an

Fig. 17 Improvements over temporal sharing of SPM space [38]

Fig. 18 Effects of context switch window

average 33 % reduction in execution time and 43 % energy savings. As expected, sharing SPM space carefully between both data and instructions shows greater energy savings and reduced execution times (MIX label). We see further 12 % energy savings by sharing on-chip memory space between both instructions and data, while further reducing execution time by an average 16 %.

Now that we have seen the effects of mapping both data and instructions to vSPMs, we want to explore how the SPMVisor fares against traditional schemes. For this purpose, we took the traditional context switching scheme with temporal allocation of SPM space ([38]) and compared it with our approach. Figure 17 shows that across the various configurations, on average, vSPMs allow us to reduce execution time by 39 % while achieving an average 59 % energy savings across the different configurations. This is due to the fact that temporal sharing of SPM space has to swap contents of SPM space on every context switch to guarantee that the applications can resume their normal execution.

Moreover, in this mode, we can guarantee that the applications run in their own sand-boxes (Sect. 4.8). The only information being leaked is possibly the time it spends over the bus, but this can easily be resolved by enforcing time-sharing of the bus (e.g., TDMA-based arbitration protocols).

7.8 Effects of the context switching window

The next issue we want to address is how the context-switching window will affect SPMVisor's performance/energy savings over the more traditional temporal allocation [38]. Note

Fig. 19 Improvements over fixed allocation [15]

that the longer the context-switching window, the more time a CPU dedicates to executing an application. For this experiment, we ran the 2 Applications per CPU, with a total of 4 CPUs, and 4×16 KB SPM space with mixed workloads (e.g., allowing instructions and data to share the SPM space). As shown in Fig. 18, we vary the context switching window from 150,000 to 2 Million cycles. As expected, the longer the window (e.g., 1M to 2M), the smaller the overheads incurred by context switching. In this case, SPMVisor goes from saving 76 % energy (1M window) to saving 47 % energy. Similarly, execution time reduction goes down from 55 % (1M) to 29 % (2M). For larger context switching windows, the more the system converges to the uni-processor mode, and where the user will not be taking advantage of the multi-tasking capabilities of the CMP. For results showing overheads of running applications with no context switching please refer to Sect. 7.3.

7.9 Effects of dynamic allocation policy

The final issue we wanted to address is the impact of the allocation policy. Our scheme supports policy-based priority-driven allocation, which allows us to efficiently utilize the on-chip memory space. This however, comes at the cost of longer allocation decision times (e.g., if not enough space, the block with lowest priority/utilization is evicted), which is in the order of hundreds of cycles by exploiting DMA bursts (e.g., dominated by the time to transfer a block from SPM to DRAM). The fixed dynamic allocation scheme (referred to as fixed) [15] maps data to SPM space as long as there is space and does not prioritize what goes on-chip and what goes off-chip. If there is no space on-chip (in SPM space), then data is mapped off-chip (DRAM). We observe that our approach, despite evicting blocks to grant access to blocks with higher priority is able to outperform the fixed allocation scheme in most cases. For applications with small footprint, both the fixed allocation scheme and our scheme achieve the same performance/energy utilization (see the 2×1 data point in Fig. 19). For scenarios where the amount of SPM space is not enough for all applications we clearly see the advantage of our scheme (see configurations 4 applications with a single CPU and 16 KB SPM space (4×1 in Fig. 19) to 2 applications per CPU, 4 CPUs total, and 4×16 KB SPMs (2×4 in Fig. 19)). We observe an average 54 % execution time reduction and 65 % energy savings over the fixed allocation scheme.

7.10 Comparing traditional SPM allocation policies

For the next set of experiments we have changed the architectural template to resemble the template shown in Fig. 20. In this architectural template, each core has access to its local SPM (1 cycle latency), and can access its neighbor SPMs through the router. The core frequency and the NoC frequency are clocked at 1 GHz. The router configuration is similar to the Intel SCC [20, 25]. We interfaced our framework with ORION 2.0 [25] to obtain the

Fig. 20 Mesh-based CMP with distributed SPMs

Table 5 Network-on-chip configuration [25]

Voltage	Frequency	Transistor Type	Number of Ports
1.1 V	1 GHz	HVT	5
Number of VC	Input Buffers	Output Buffers	Flit Width
1	2	1	32

Network-on-Chip's energy and cycle counts. More details on the architectural template can be found in Table 5. Figure 20 shows how the SPMVisor distributes blocks of data across the different SPMs in the system. For the sake of illustration, in this example the CPU at (0, 0) created an 8 KB vSPM, and allocated four 2 KB blocks. These blocks were then distributed by the SPMVisor (as shown as dark-purple shaded boxes labeled 2 KB).

We define *four* block-based allocation policies: (1) *Temporal* allocation [38], which consists of temporally sharing the SPM space among the various applications. To share the SPM space, this policy swaps the contents of the SPM on a context-switch to avoid conflicts with other tasks. (2) *Dynamic* allocation, a dynamic-spatial SPM allocation ([15]) scheme, which maps the data block to the preferred memory type as long as there is space, otherwise, data is mapped to off-chip memory (DRAM). (3) *Spatial* allocation [38], which partitions the SPM space equally among the different applications running on the system. Finally, (4) our *Priority-driven* allocation policy, which is what drives the allocation of the memory space in the SPMVisor. For each of the different allocation policies *Temporal*, *Spatial*, *Dynamic*,

Fig. 21 Software-controlled memory allocation policies

Fig. 22 Mesh-based CMP configurations

the assumption is that all memory transfers go *directly* to their respective local memory (e.g., they all share a local SPM). These allocation policies can be visualized in Fig. 21(a) to (d). In this context we can see two applications, *A* and *B*, where each has two types of block priorities, *Low* and *High*, each referring to how utilized the memory block will be. *Temporal* (Fig. 21(a)) allocation will dedicate the full SPM to each application, for instance, in this example, every time *A* or *B* execute, they are given full access to the SPM space, however, such scheme incurs very high overheads due to flushing the contents of the memory. *Spatial* (Fig. 21(b)) allocation will allocate part of the SPM space to each application, in this case, the larger blocks for *A* or *B* would not fit into their assigned SPM space, so the allocation policy was forced to allocate less critical blocks onto SPM space. *Dynamic* (Fig. 21(c)) will allocate SPM space as long as it exists, otherwise, it will allocate data in off-chip memory. In this example, we see that *B* was forced to allocate its data to off-chip memory. Finally, Fig. 21(d) shows the *SPMVisor*'s priority-driven memory allocator, which tries to map highly utilized data to on-chip memory. For the sake of illustration, the darker shade blocks have higher priorities than the lighter shaded blocks, and *B* has higher priority than *A*, so *B*'s larger block was mapped to SPM space along with *A*'s smaller dark-shaded block.

Figure 22 shows three different mesh-based CMPs (2×2 , 3×3 , and 4×4). The dark box marked by the *M* letter shows the position of the node where the *SPMVisor* resides. Each node (box) consists of a total of 32 KB SPM space.

Fig. 23 Mesh-based CMP configurations

Figure 23 shows the normalized execution time (top) and energy (bottom) with respect to the *Spatial* allocation policy and executing 1024 jobs in the 2×2 Mesh shown in Fig. 20. For this experiment, we took the same 8 applications as in the previous sections and replicated them along with their SPM allocation policies. As a result, we generated a pool of 1024. To stress the memory system, we then ran all jobs and demonstrated how each allocation policy fared in terms of execution time (time it took all 1024 jobs to finish), and the total energy utilized by the system. Jobs were all processed in batches of 8 jobs per CPU, so the memory state of the system would reset after completing the execution of 8 jobs per CPU. Each CPU runs a context-switching OS as was done in the previous versions. The SPMVisor's allocation policy exploits the notion of block priority, in this case, the utilization of each memory block was used to prioritize allocation. In this example we can observe that SPMVisor's priority-driven allocation (the *Priority* line in the graphs) is able to reduce execution time by an average 17 % with respect to the *Spatial* allocation policy, while saving an average 65 % in total energy. Note that the reason why the *Dynamic* allocation policy fares the worst is that when instances of applications with high memory requirements (e.g., JPEG or H263) need to run, the on-chip memory space may have already been taken by the applications with least requirements. The *Spatial* allocation policy and the *Temporal* allocation policies perform almost the same, with *Spatial* edging *Temporal* allocation. *Spatial* allocation guarantees some SPM space to each job, but the total amount may be small as the number of applications grow, where as *Temporal* guarantees the entire SPM space for each application, but suffers the overhead of flushing the contents of the memory space.

7.11 Effects of varying the number of jobs

Like in the previous section, we generated a pool of jobs ranging from 32 to 512 jobs by replicating the initial 8 applications and their SPM allocation policies. The goal of this experiment is to observe the effects of varying the number of jobs and allocation policy on the system's performance and energy. As shown in Fig. 24, our *Priority*-based allocation policy is able to reduce execution time by an average 23 % with respect to the *Spatial* allocation policy, while saving an average 58.4 % in total energy.

Fig. 24 Normalized execution time and energy for a 2×2 mesh CMP

Fig. 25 Normalized execution time and energy for a 4×4 mesh CMP

Figure 25 shows a similar trend as Fig. 24, however, in this example we see how the extra transactions on the NoC incurred by the SPMVisor start taking their toll on performance. In this scenario, SPMVisor is able to reduce execution time by an average 1.4 % while saving an average of 60 % energy. One of the limitations of our scheme is how scalable the SPMVisor may be as we start migrating towards many-cores, thus we need to be careful about how to efficiently distribute these memory managers in a mesh-based architecture. Since this work focuses on CMPs, we leave the exploration into scalability as future work.

Fig. 26 Normalized execution time and energy for a 2×2 mesh with varying memory sizes

7.12 Effects of varying memory size

Like in the previous section, we generated a pool of 256 jobs by replicating the initial 8 applications and their SPM allocation policies. The goal of this experiment is to observe the effects of varying SPM size and allocation policy on the system's performance and energy. As shown in Fig. 26, our *Priority*-based allocation policy is able to reduce execution time by an average 17.7 % with respect to the *Spatial* allocation policy, while saving an average 43.7 % in total energy. Es expected, as the memory size increased, the system performance improved for most policies. Moreover, we can observe that the sweet spot for the *Temporal* allocation scheme is around 32 KB, which is enough space to support the various tasks, while minimizing the effects of the flushing in of the SPM contents.

8 Conclusion

In this paper, we introduced the concept of *SPMVisor*, a hardware/software layer that virtualizes the scratchpad memory space in order to facilitate the use of distributed SPMs in an efficient, transparent and secure manner. We introduce the notion of virtual Scratch-Pad Memories (vSPMs) to provide software programmers a transparent view of the on-chip memory space. To protect the on-chip memory space, the SPMVisor supports vSPM-level and block-level access control lists. Finally, in order to efficiently manage the on-chip real-estate, our SPMVisor supports policy-driven allocation strategies based on privilege levels.

Our experimental results on Mediabench/CHStone benchmarks running on various Chip-Multiprocessor configurations and software stacks (RTOS, virtualization, secure execution)

showed that SPMVisor enhances performance by 71 % on average and reduces power consumption by 79 % on average with respect to traditional context switching schemes. We showed the benefits of using vSPMs in a various environments (a RTOS multi-tasking environment, a virtualization environment, and a trusted execution environment). Furthermore, we explored the effects of mapping instructions and data onto vSPMs, and showed that sharing on-chip space reduces both execution time and energy by an average 16 % and 12 % respectively. We then compared our priority-driven memory allocation scheme with traditional dynamic allocation and showed an average 54 % execution time reduction and 65 % energy savings. Finally, to further validate the SPMVisor's benefits, we modified the initial bus-based architecture to include a mesh-based CMP with up to 4×4 nodes. We were able to observe that SPMVisor's priority-driven allocator was able to reduce execution time by an average 17 % with respect to competing allocation policies, while saving an average 65 % across various architectural configurations. We were also able to observe that SPMVisor reduces execution time by an average 12.6 % with respect to competing allocation policies, while saving an average 63.5 % in total energy for various architectural configuration running 1024 jobs. We also varied the SPM's sizes and were able to see that the SPMVisor is able to reduce execution time by an average 17.7 % with respect to the *Spatial* allocation policy, while saving an average 43.7 % in total energy.

To show the benefits of our virtualization layer we bypassed the data cache, and the instruction cache (some instances), however, even with the presence of both instruction and data caches, we feel that our layer can still outperform traditional SPM management schemes in a multi-tasking environment.

Our scheme does indeed have some limitations: (1) The scalability of the SPMVisor as we migrate towards many-core platforms. (2) The overheads of the purely-software virtualization layer. (3) The area, performance, and power overheads of the hardware SPMVisor. Finally, (4) The granularity of the SPM dynamic allocator. Thus, these limitations motivate the need for further research in the area.

Our current and future work includes the integration of the SPMVisor into the OS/hypervisor layer as software module, tightly coupling the SPMVisor into a fully functional hardware/software virtual layer, exploration of the overheads due to full application sandboxing, exploring how to modify the SPMVisor to support real-time on-chip memory management, and exploring the scalability of the SPMVisor in a many-core architecture configuration platform.

Acknowledgements This work was partially supported by NSF Variability Expedition Grant Number CCF-1029783. And in part by MKE, Korea, under the ITRC support program (NIPA-2011-C1090-1131-0009) and in part by Basic Science Research Program through the NRF funded by the Ministry of Education, Science and Technology (2011-0004852).

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Agarwal V, Hrishikesh MS, Keckler SW, Burger D (2000) Clock rate versus ipc: the end of the road for conventional microarchitectures. SIGARCH Comput Archit News 28
2. ARM (1999) Amba spec rev 2.0. In: IHI-0011A
3. Austin T, Larson E, Ernst D (2002) Simplescalar: an infrastructure for computer system modeling. Computer 35(2)

4. Bai K, Shrivastava A (2010) Heap data management for limited local memory (llm) multi-core processors. In: Proc of the eighth IEEE/ACM/IFIP int conf on hardware/software codesign and system synthesis, CODES/ISSS '10
5. Banakar R, Steinke S, Lee BS, Balakrishnan M, Marwedel P (2002) Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: Proc of the tenth int symposium on hardware/software codesign, CODES '02
6. Bathen LAD, Dutt N (2010) Polimake: a policy making engine for secure embedded software execution on chip-multiprocessors. In: Proc of the 5th workshop on embedded systems security, WESS '10
7. Bathen L, Dutt N (2011) E-roc: embedded raids-on-chip for low power distributed dynamically managed reliable memories. In: Design, automation test in Europe conference exhibition (DATE)
8. Bathen LAD, Dutt ND, Shin D, Lim SS (2011) 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, CODES+ISSS '11. ACM, New York, pp 79–88
9. Benini L, Micheli GD (2002) Networks on chips: a new soc paradigm. *IEEE Comput* 35(1)
10. Cho D, Pasricha S, Issenin I, Dutt N, Paek Y, Ko S (2008) Compiler driven data layout optimization for regular/irregular array access patterns. In: Proc of the 2008 ACM SIGPLAN-SIGBED conf on languages, compilers, and tools for embedded systems, LCTES '08
11. Coburn J, Ravi S, Raghunathan A, Chakradhar S (2005) Seca: security-enhanced communication architecture. In: Proc of the 2005 int conf on compilers, architectures and synthesis for embedded systems, CASES '05
12. David FM, Carlyle JC, Campbell RH (2007) Context switch overheads for Linux on arm platforms. In: Proc of the 2007 workshop on experimental computer science, ExpCS '07
13. Egger B, Lee J, Shin H (2008) Dynamic scratchpad memory management for code in portable systems with an mmu. *ACM Trans Embed Comput Syst* 7
14. Egger B, Kim S, Jang C, Lee J, Min SL, Shin H (2010) Scratchpad memory management techniques for code in embedded systems without an mmu. *IEEE Trans Comput* 59
15. Francesco P, Marchal P, Atienza D, Benini L, Catthoor F, Mendias JM (2004) An integrated hardware/software approach for run-time scratchpad management. In: Proc of the 41st annual design automation conf, DAC '04
16. Gauthier L, Ishihara T, Takase H, Tomiyama H, Takada H (2010) Minimizing inter-task interferences in scratch-pad memory usage for reducing the energy consumption of multi-task systems. In: Proc of the 2010 int conf on compilers, architectures and synthesis for embedded systems, CASES '10
17. Google: Android. Google, <http://www.android.com/>
18. Hara Y, Tomiyama H, Honda S, Takada H, Ishii K (2008) Chstone: a benchmark program suite for practical C-based high-level synthesis. In: IEEE int symp on circuits and systems, 2008. ISCAS 2008
19. Heiser G (2008) The role of virtualization in embedded systems. In: Proc of the 1st workshop on isolation and integration in embedded systems, IIES '08
20. Howard J et al (2010) A 48-core ia-32 message-passing processor with dvfs in 45 nm cmos. In: ISSCC. doi:[10.1109/ISSCC.2010.5434077](https://doi.org/10.1109/ISSCC.2010.5434077)
21. Huang W, You K, Zhang S, Han J, Zeng X (2009) Unified low cost crypto architecture accelerating rsa/sha-1 for security processor. In: IEEE 8th int conf on ASIC, 2009. ASICON '09
22. IBM The cell project. <http://www.research.ibm.com/cell/>
23. Issenin I, Brockmeyer E, Durinck B, Dutt N (2006) Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In: Proc of the 43rd annual design automation conf, DAC '06
24. Jung Sc, Shrivastava A, Bai K (2010) Dynamic code mapping for limited local memory systems. In: 21st IEEE int conf on application-specific systems architectures and processors (ASAP)
25. Kahng A, Li B, Peh LS, Samadi K (2012) Orion 2.0: a power-area simulator for inter-connection networks. *IEEE Trans Very Large Scale Integr (VLSI) Syst* 20(1):191–196. doi:[10.1109/TVLSI.2010.2091686](https://doi.org/10.1109/TVLSI.2010.2091686)
26. Kandemir M, Ramanujam J, Irwin J, Vijaykrishnan N, Kadayif I, Parikh A (2001) Dynamic management of scratch-pad memory space. In: Proc of the 38th annual design automation conf, DAC '01
27. Lee C, Potkonjak M, Mangione-Smith WH (1997) Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In: Proc of the 30th annual ACM/IEEE int symposium on microarchitecture, MICRO '30
28. Mayhew M, Muresan R (2009) Low-power aes coprocessor in 0.18 um cmos technology for secure microsystems. In: 2nd microsystems and nanoelectronics research conf 2009, MNRC 2009
29. McCune JM, Parno BJ, Perrig A, Reiter MK, Isozaki H (2008) Flicker: an execution infrastructure for tcb minimization. *SIGOPS Oper Syst Rev* 42
30. OpenKernelLabs: Okl4. In: IHI-0011A

31. Panda PR, Dutt ND, Nicolau A (1997) Efficient utilization of scratch-pad memory in embedded processor applications. In: Proc of the European conf on design and test, EDTC '97
32. Pasricha S, Dutt N, Ben-Romdhane M (2008) Fast exploration of bus-based communication architectures at the ccatb abstraction. *ACM Trans Embed Comput Syst* 7
33. Pyka R, Faßbach C, Verma M, Falk H, Marwedel P (2007) Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications. In: Proc of the 10th int workshop on software & compilers for embedded systems, SCOPES '07, pp 41–50
34. Shalan M, Mooney VJ (2000) A dynamic memory management unit for embedded real-time system-on-a-chip. In: Proc of the 2000 int conf on compilers, architecture, and synthesis for embedded systems, CASES '00, pp 180–186
35. Shimizu K, Hofstee HP, Liberty JS (2007) Cell broadband engine processor vault security architecture. *IBM J Res Dev* 51(5):521–528
36. Suhendra V, Raghavan C, Mitra T (2006) Integrated scratchpad memory optimization and task scheduling for mp soc architectures. In: Proc of the 2006 int conf on compilers, architecture and synthesis for embedded systems, CASES '06
37. Suhendra V, Roychoudhury A, Mitra T (2008) Scratchpad allocation for concurrent embedded software. In: Proc of the 6th IEEE/ACM/IFIP int conf on hardware/software codesign and system synthesis, CODES+ISSS '08
38. Takase H, Tomiyama H, Takada H (2010) Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In: Proc of the conf on design, automation and test in Europe, DATE '10
39. Thoziyoor S, Muralimanohar N, Ahn JH, Jouppi NP (2004) Hp labs cacti v5.3. CACTI 5.1, TR. <http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html>
40. Tiler Tile gx family. http://www.tiler.com/products/processors/TILE-Gx_Family
41. Verma M, Steinke S, Marwedel P (2003) Data partitioning for maximal scratchpad usage. In: Proc of the 2003 Asia and South Pacific design automation conf, ASP-DAC '03
42. Verma M, Petzold K, Wehmeyer L, Falk H, Marwedel P (2005) Scratchpad sharing strategies for multiprocess embedded systems: a first approach. In: 3rd workshop on embedded systems for real-time multimedia, 2005, pp 115–120. doi:[10.1109/ESTMED.2005.1518087](https://doi.org/10.1109/ESTMED.2005.1518087)
43. Wang Z, Lee RB (2007) New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput Archit News* 35