Towards Distributed On-Chip Memory Virtualization

Luis Angel D. Bathen, Dongyoun Shin, Sung-Soo Lim, Nikil D. Dutt

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{lbathen,dutt}@uci.edu, {elsdy, sslim}@kookmin.ac.kr

CECS Technical Report #11-08
August, 2011

Towards Distributed On-Chip Memory Virtualization

LUIS ANGEL D. BATHEN, University of California, Irvine
DONGYOUN SHIN, Kookmin University, Seoul, South Korea
SUNG-SOO LIM, Kookmin University, Seoul, South Korea
NIKIL D. DUTT, University of California, Irvine

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 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 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. 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) show that SPMVisor enhances performance by 71% on average and reduces power consumption by 79% on average.

Categories and Subject Descriptors: C.3 [**Special-purpose and Application-based systems**]: Real-time and embedded systems; D.4.6 [**Security and Protection**]: Access Controls; Security Kernels; B.3 [**Design Styles**]: Virtual Memory; D.4 [**Storage Management**]: Distributed memories

General Terms: Design, Management, Performance, Security

Additional Key Words and Phrases: 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 [Agarwal et al. 2000], and technology scaling have pushed for the move towards multiprocessor technology ([IBM 2005; Intel 2009; Tilera 2010]). A byproduct of the multicore phenomena is the rapid integration of distributed scratchpad memories (SPMs) into the memory hierarchy [IBM 2005] due to their increased predictability, reduced area and power consumption [Banakar et al. 2002]. 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 [Panda et al. 1997; Banakar et al. 2002; Verma et al. 2003; Issenin et al. 2006; Cho et al. 2008]. Traditional approaches assume that a given application is granted full access to the underlying resources [Panda et al. 1997; Banakar

This research was partially supported by NSF Variability Expeditions Award CCF-1029783, and SFS/NSF Grant No. 0723955.
Authors' addresses: Luis Angel D. Bathen and Nikil Dutt, Center for Embedded Computer Systems, School of Information and Computer Science, University of California at Irvine, Irvine, CA 92697;

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

et al. 2002; Verma et al. 2003; Issenin et al. 2006; Cho et al. 2008; Jung et al. 2010; Bai and Shrivastava 2010; Suhendra et al. 2006], 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 [Suhendra et al. 2008; Gauthier et al. 2010; Takase et al. 2010], 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 [Google]) 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 [Wang and Lee 2007]) 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 [Coburn et al. 2005]), or (b) expose private information via side channel attacks [Wang and Lee 2007]. Virtualization has been proposed as a possible solution to the ever growing threats in open systems, where VM instances with various privilege levels may run different software stacks [Heiser 2008]; 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) show that SPMVisor enhances performance by 71% on average and reduces power consumption by 79% on average.

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 trusted application execution.

2.1. Shared SPMs in Heterogeneous 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 (4KB space each) with temporal and spatial allocation (hybrid allocation) [Takase et al. 2010] 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). Consider the case where a high-priority application is launched (App5 denoted by the red/dotted box in Figure 1); two schemes can be used: 1) Flush the contents of the SPM and grant full SPM access to App5. 2) Strictly follow the static placement and map contents for App5 to off-chip memory as shown in Figure 1. Assume that App5 is a critical application with real-time requirements; then mapping the data off-chip might not be the best approach as the overhead due to off-chip accesses

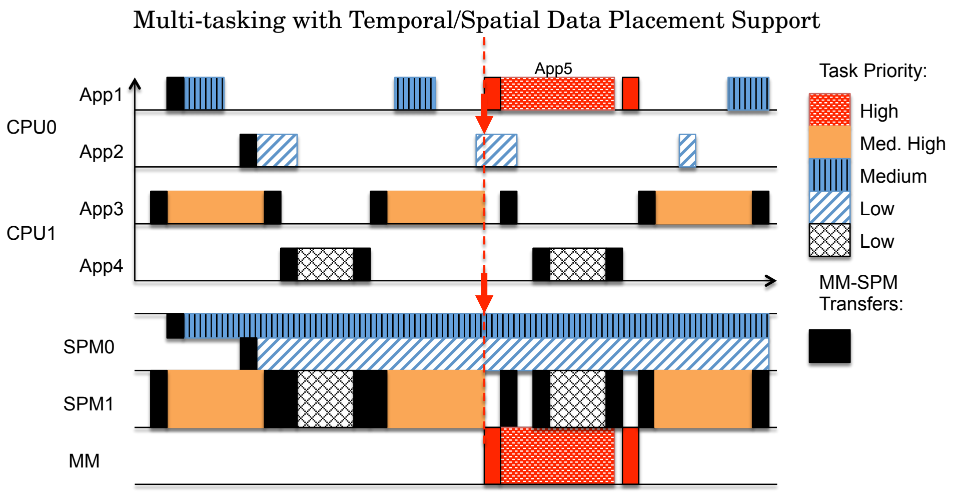


Fig. 1. Optimal data placement is not guaranteed in open environments (e.g., Android based systems)

might lead it to miss its deadlines. Figure 1 shows the need for dynamic allocation of SPM space considering the priority of the applications in a heterogenous multi-tasking system.

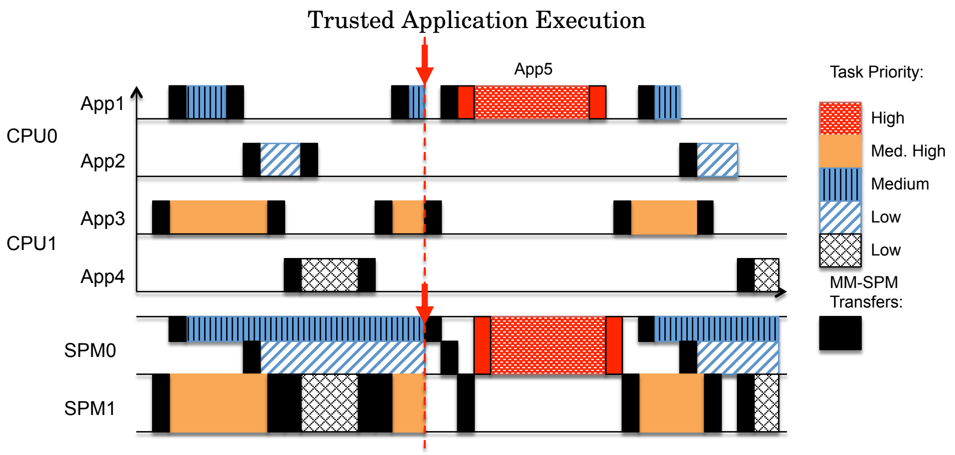


Fig. 2. Halting all executing processes and flushing SPM contents in order to provide a trusted execution environment

2.2. Trusted Application Execution

Trusted application execution is needed in a heterogeneous 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 [Heiser 2008]. 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

[McCune et al. 2008] as shown in Figure 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 [McCune et al. 2008] 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/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/performance.

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 heterogeneous 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) [Panda et al. 1997; Banakar et al. 2002; Verma et al. 2003; Issenin et al. 2006; Cho et al. 2008; Jung et al. 2010; Bai and Shrivastava 2010; Suhendra et al. 2006] as well as scheduling for SPM enabled systems [Suhendra et al. 2006; Takase et al. 2010].

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

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

3.1. Sw/Hw Virtualization Support Tradeoff

Our virtualization layer can be a software layer running at the hypervisor/OS level as a module (*Soft*SPMVisor) or as a hardware IP block similar to an arbiter (*Hard*SPMVisor). The *Soft*SPMVisor layer should be light-weight, flexible, and modularized in a manner that allows for easy integration into existing OSes/Hypervisors. The *Hard*SPMVisor module should have minimal area overheads, and support a simplified API for transparent use by the programmers or the OS/Hypervisor software stacks. The *Soft*SPMVisor 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 *Soft*SPMVisor comes at the cost of higher power/performance overheads than the *Hard*SPMVisor. Ideally, both *Soft*SPMVisor and *Hard*SPMVisor should support the same minimal API and should require minimal changes in the programming model. For this paper, we will focus on the *Hard*SPMVisor, and leave the *Soft*SPMVisor as future work. Our goal is to have a tightly coupled SW/HW layer that exploits the benefits of both *Soft*SPMVisor and

*Hard*SPMVisor. For the remainder of this paper we will refer to the *Hard*SPMVisor as SPMVisor.

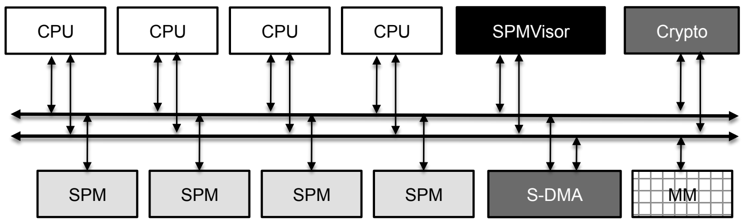


Fig. 3. SPMVisor enhanced Chip-Multiprocessor

3.2. Target Platform

Figure 3 shows the high-level diagram of our target platform. Our Chip-Multirprocessor (CMP) resembles the platform used in [Suhendra et al. 2006; blind a; b], which consists of a set of RISC cores, distributed SPMs, an AMBA AHB [ARM 1999] on-chip bus, enhanced with a secure DMA (S-DMA), and a cryptographic engine (Crypto) similar to the ones in [Huang et al. 2009; Mayhew and Muresan 2009].

3.3. Assumptions

We assume that the programmer/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. We focus on SPM data, however, our approach can be used for instructions as well since we provide an API for transparent utilization of the vSPMs (e.g., vSPM management can follow the same policies as regular SPMs). For this work we bypass the data-cache and data that is not mapped onto SPM space is mapped onto off-chip memory since we wanted to focus purely on the benefits of SPMVisor. 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.

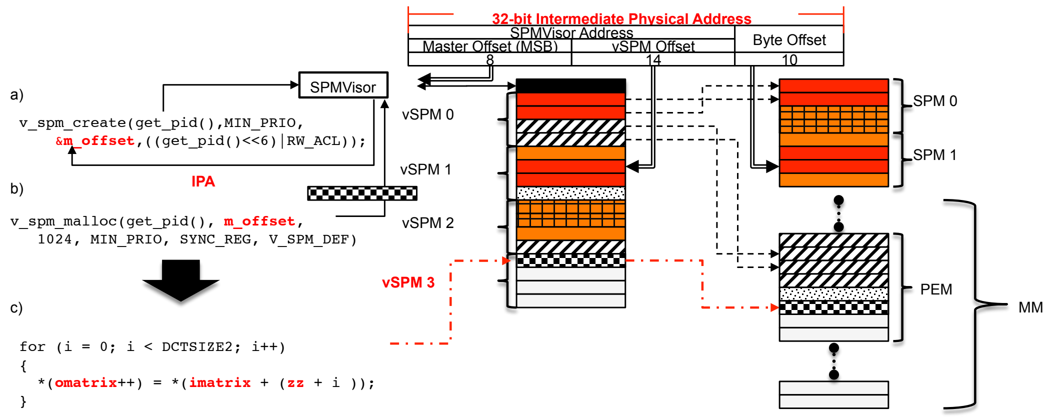
4. SPMVISOR: SCRATCHPAD MEMORY VIRTUALIZATION

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 deleted when no longer needed. Table I 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 real-time decisions for efficient on-chip resource utilization as we

Table I: vSPM Management API

Method	Parameter	Type	Note
v_spm_create	PID	uint	Process ID
	AppPriority	uint	Application priority
	IPA	uint*	Intermediate Physical Address
	ACL	uint	vSPM level access control list
v_spm_delete	PID	uint	
	IPA	uint	
v_spm_malloc	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
v_spm_blk_del	PID	uint	
	IPA	uint	
v_spm _clear	PID	uint	
	IPA	uint	
v_spm _poll	PID	uint	
	IPA	uint	
v_spm _transfer	PID	uint	
	SrcAddr	uint	Transfer source address
	DstAddr	uint	Transfer destination address
	TxType	uint	Transaction type sync/async/secure



```
v_spm_create(get_pid(),MIN_PRIO,
    &m_offset,((get_pid()<<6)|RW_ACL));
```

```
v_spm_malloc(get_pid(), m_offset,
   1024, MIN_PRIO, SYNC_REG, V_SPM_DEF)
```

```
for (i = 0; i < DCTSIZE2; i++)
{
  *(omatrix++) = *(imatrix + (zz + i ));
}
```

Fig. 4. Process of SPMVisor vSPM creation/allocation with view of memory space

assume a heterogeneous environment consisting of applications with various requirements (real-time, security, reliability, etc.). IPA means *intermediate physical address*, and is 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). 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*).

```
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

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_del* 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. 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 (1KB), 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.

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 4 shows the memory view of a newly created vSPM (vSPM 3), after invocation to the *v_spm_create* method (Figure 4(a)), we obtain the IPA for the vSPM from the SPMVisor. 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 (Figure 4(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 (Figure 4(c)). Figure 4 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 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 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 in real-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 [Francesco et al. 2004] 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 [Francesco et al. 2004]. 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 5 shows the same sequence of tasks being executed as in Figure 1, where a new critical task is introduced into the system with high priority (dotted red 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 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 (red 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,

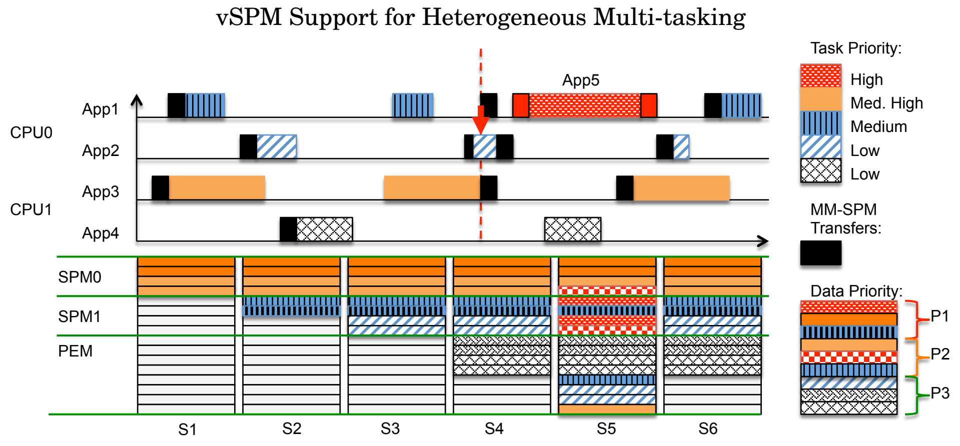


Fig. 5. Data-driven priority allocation

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.

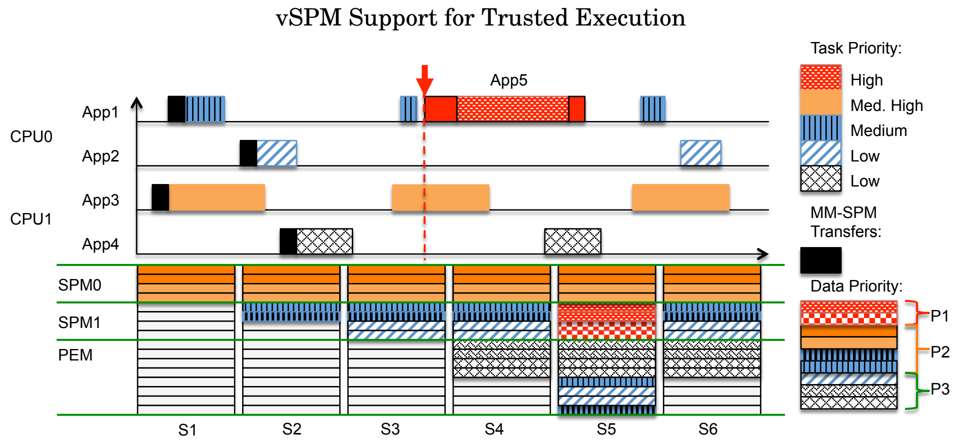


Fig. 6. Application-driven priority allocation

Figure 6 depicts the various states undergone by the applications and their contents when following our application-driven allocation policy. Just like in Figure 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 Figure 5. 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

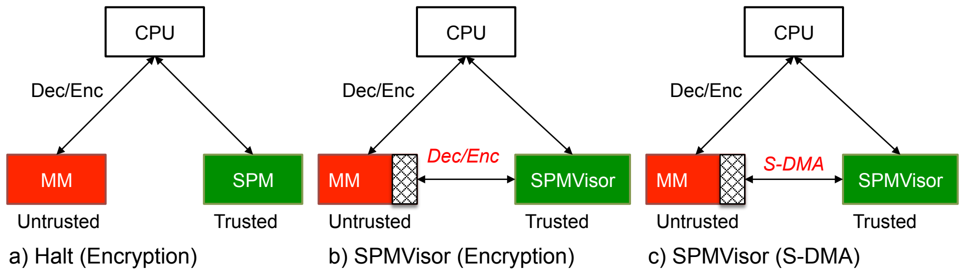


Fig. 7. Data protection schemes

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. Figure 7(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 7(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 (Figure 7(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.

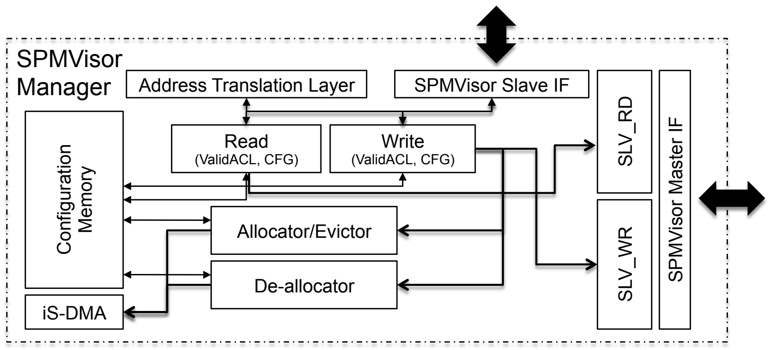


Fig. 8. Block diagram of SPMVisor Module

4.6. SPMVisor HW Module

Figure 8 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 512B, 1024B, 4KB, 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 512B and 256KB).

Table II: vSPM Management API

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

The breakdown of the block metadata is shown in Table II. 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. 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 (Figure 8) 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.

5. OS/HYPERVISOR INTEGRATION

The OS/Hypervisor Layers can benefit from exploiting vSPMs since a benefit of using the SPMVisor is much lower context switching times. As our focus is the introduction of the SPMVisor and use the *Hard*SPMVisor as a proof-of-concept implementation, we will not go into details into the *Soft*SPMVisor implementation and how it can be integrated into a hypervisor/OS layer. However, we will briefly discuss some of the key benefits of using vSPMs and *Hard*SPMVisor 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).

6. RELATED WORK

SPMs have through the years become a critical component of the memory hierarchy [Jung et al. 2010; Bai and Shrivastava 2010], and are expected to be the memories of choice for future many-core platforms (e.g., [IBM 2005; Intel 2009; Tilera 2010]). 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. [Panda et al. 1997] 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. [Verma et al. 2003] 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. [Kandemir et al. 2001] 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. [Issenin et al. 2006] 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. [Suhendra et al. 2006] proposed and ILP formulation to find out the optimal placement of data onto SPMs. Jung et al. [Jung et al. 2010] looked at dynamic code mapping through static analysis in order to maximize SPM utilization for functions. Bai et al. [Bai and Shrivastava 2010] looked at heap-data management for SPMs. Shalan et al. [Shalan and Mooney 2000] looked at dynamic memory management for global memory through the use a hardware module.

In order to support concurrent execution of tasks sharing the same SPM resources [Suhendra et al. 2008; Gauthier et al. 2010; Takase et al. 2010] propose various static analysis approaches, which assume all working sets are known at compile time. Francesco et al. [Francesco et al. 2004] proposed a memory manager that supports dynamic allocation of SPM space, which supports block-based allocation (fixed and variable). Egger et al. proposed SPM management techniques for MMU supported [Egger et al. 2008] and MMU-less embedded systems [Egger et al. 2010], 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. [Pyka et al. 2007] 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 [Francesco et al. 2004] 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 [Francesco et al. 2004] 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 our knowledge, we are the first to propose a virtualization layer for on-chip distributed memories. 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 [Panda et al. 1997; Kandemir et al. 2001; Verma et al. 2003; Suhendra et al. 2006; Suhendra et al. 2008; Gauthier et al. 2010; Takase et al. 2010; Pyka et al. 2007]. Our allocation engine can even exploit some of the allocation mechanisms presented in [Francesco et al. 2004]. Finally, since using our vSPMs require little effort on the programmer's end, techniques such as [Jung et al. 2010; Bai and Shrivastava 2010; Egger et al. 2008; Egger et al. 2010] can exploit vSPMs for code/function/instruction management. Consider the work introduced by [Egger et al. 2008; Egger et al. 2010], 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.

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 our approach in a heterogeneous 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.

7.2. Experimental Setup

Figure 9 shows our experimental setup where we implemented our SPMVisor module as a SystemC TLM/CCATB [Pasricha et al. 2008] block and integrated it into our simulation framework [blind b], which interfaces with Simplescalar [Austin et al. 2002] and CACTI [Thoziyoor et al. 2004]. We assume 65nm process technology for our memories and a 128 MB off-chip main memory. We cross-compiled a set of applications from the CHStone [Hara et al. 2008] and Mediabench II [Lee et al. 1997] 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). In order to support multiple applications running concurrently we used page tables (1KB 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

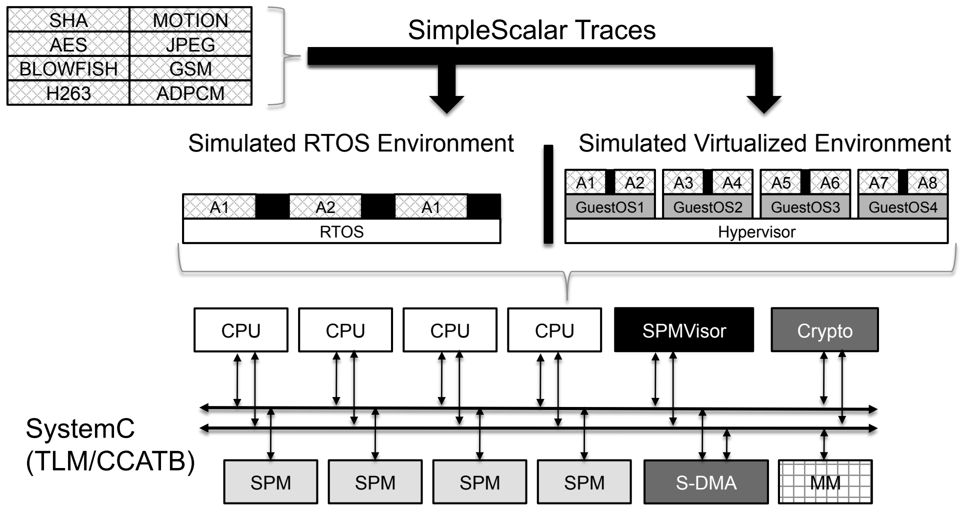


Fig. 9. Experimental Setup

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.

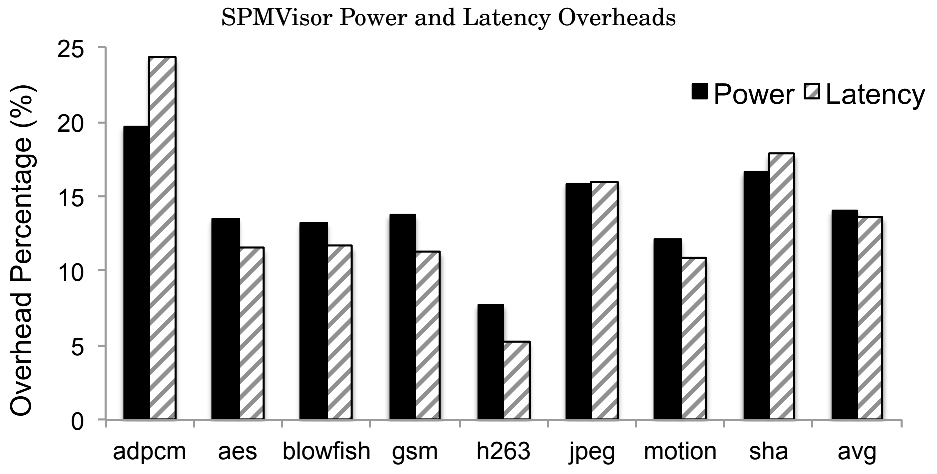


Fig. 10. Overheads in an ideal world

7.3. SPMVisor Overheads

Figure 10 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 8KB 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.

7.4. SPMVisor Support for Heterogenous Multi-tasking Environments

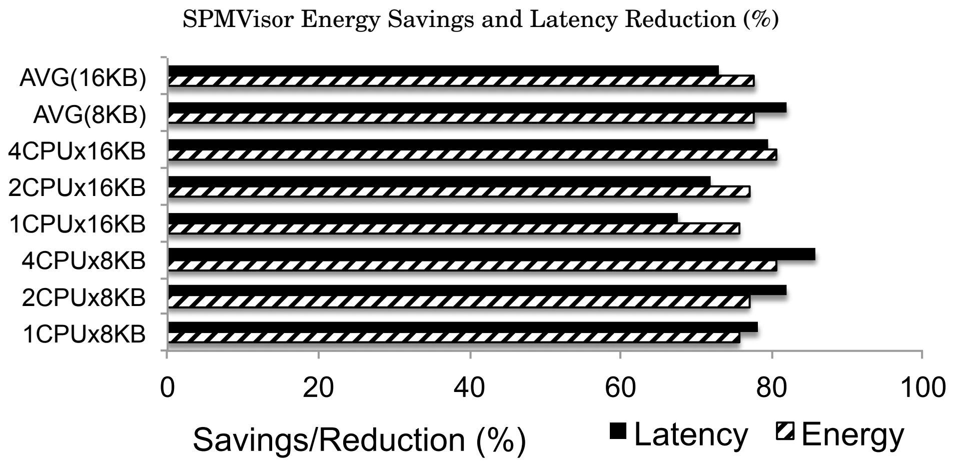


Fig. 11. RTOS SPMVisor savings for various configurations

Figure 11 shows the energy savings and latency reductions for a set of configurations: a) 8 Apps/CPU, 1xCPU, and 1x16KB SPM (1CPUx16KB). b) 4 Apps/CPU, 2xCPUs, and 2x16KB SPMs (2CPUx16KB). c) 2 Apps/CPU, 4xCPUs, and 4x16KB SPMs (4CPUx16KB). We use the same configurations for 8KB and 16KB 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 Figure 11, our approach achieves 85% power consumption reduction and 82% latency reduction for a CMP with 16KB distributed SPMs and 77% power consumption reduction and 72% latency reduction for a CMP with 8KB 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 16KB SPMs than the 8KB SPMs because the amount of contention for SPM space between the applications is lower.

7.5. Benefits of vSPMs in a Virtualized

Table III: Virtualized Environment Configurations

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

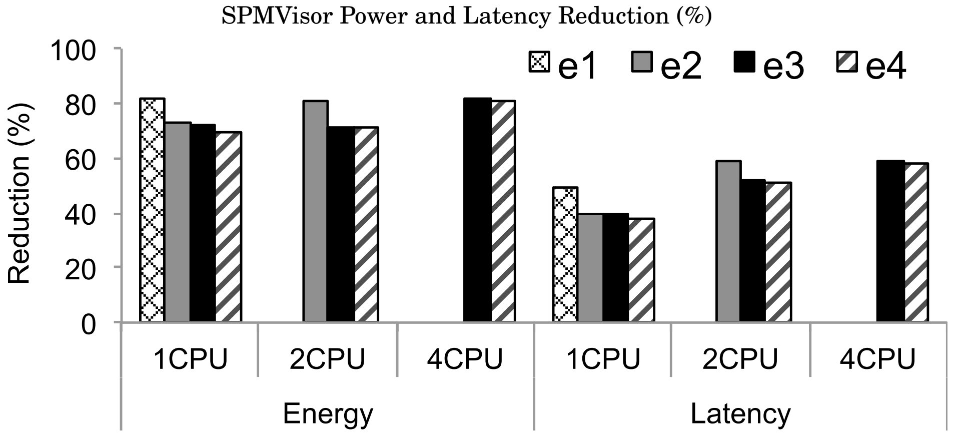


Fig. 12. Virtualized environment running on CMP with vSPM support

Environment

For this experiment we wanted to show the benefits of using vSPMs in a virtualized environment. Table III 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 in [David et al. 2007]. 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 Figure 12, where each of the configurations shown in Table III were run on top of varying number of CPUs. For this experiment we assumed that each CPU could access 8KB of SPM space (and 8KB 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 13 shows the energy efficiency and performance comparison between the three schemes discussed in Section 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 Secure-DMA support, which locks part of the main-memory and grants restrictive access to SPMVisor. In the X-axis we have three

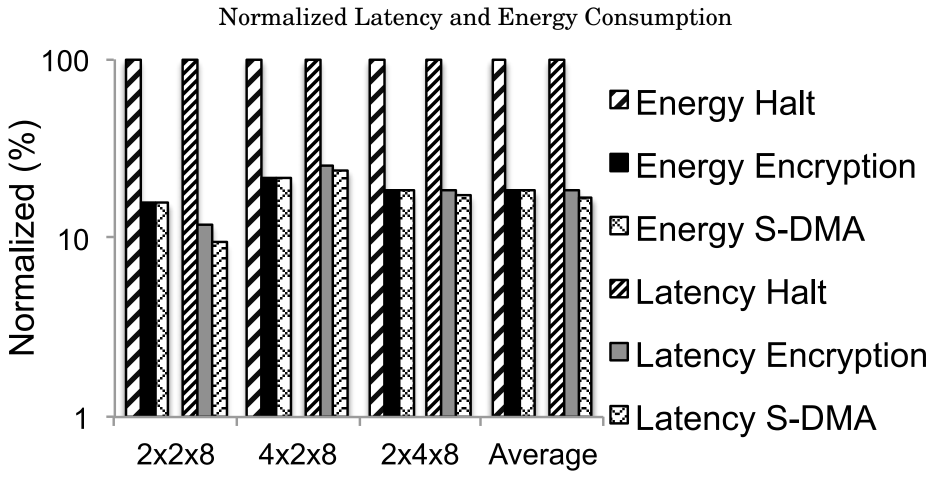


Fig. 13. Comparison between various security schemes and their normalized energy utilization and latencies

different platforms in the following format: (# of applications per CPU x # of CPUs x size of SPM assigned to each CPU), so 2x2x8 means two applications per CPU (total of 2 CPUs) and 2x8KB SPMs. We ran up to 8 applications concurrently with multitasking 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 other 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.

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 ScratchPad 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. We showed the benefits of using vSPMs in a various environments (a RTOS/heterogeneous multi-tasking environment, a virtualized environment, and a trusted execution environment). 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, and exploring the scalability of the SPMVisor in a Network-on-Chip platform.

REFERENCES

AGARWAL, V., HRISHIKESH, M. S., KECKLER, S. W., AND BURGER, D. 2000. Clock rate versus ipc: the end of the road for conventional microarchitectures. *SIGARCH Comput. Archit. News 28*, 248–259.

ARM. 1999. Amba specication rev 2.0. In *IHI-0011A*.

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. Simplescalar: an infrastructure for computer system modeling. *Computer 35,* 2, 59 –67.

BAI, K. AND SHRIVASTAVA, A. 2010. Heap data management for limited local memory (llm) multi-core processors. In *Proceedings of the eighth IEEE/ACM/IFIP Int. Conf. on Hardware/software codesign and system synthesis*. CODES/ISSS '10. 317–326.

BANAKAR, R., STEINKE, S., LEE, B.-S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth Int. Sym. on Hardware/software codesign*. CODES '02.

BLIND. for review.

BLIND. for review.

CHO, D., PASRICHA, S., ISSENIN, I., DUTT, N., PAEK, Y., AND KO, S. 2008. Compiler driven data layout optimization for regular/irregular array access patterns. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conf. on Languages, compilers, and tools for embedded systems*. LCTES '08. 41–50.

COBURN, J., RAVI, S., RAGHUNATHAN, A., AND CHAKRADHAR, S. 2005. Seca: security-enhanced communication architecture. In *Proceedings of the 2005 Int. Conf. on Compilers, architectures and synthesis for embedded systems*. CASES '05. ACM, New York, NY, USA.

DAVID, F. M., CARLYLE, J. C., AND CAMPBELL, R. H. 2007. Context switch overheads for linux on arm platforms. In *Proceedings of the 2007 workshop on Experimental computer science*. ExpCS '07. ACM, New York, NY, USA.

EGGER, B., KIM, S., JANG, C., LEE, J., MIN, S. L., AND SHIN, H. 2010. Scratchpad memory management techniques for code in embedded systems without an mmu. *Computers, IEEE Trans. on 59,* 8.

EGGER, B., LEE, J., AND SHIN, H. 2008. Dynamic scratchpad memory management for code in portable systems with an mmu. *ACM Trans. Embed. Comput. Syst. 7*.

FRANCESCO, P., MARCHAL, P., ATIENZA, D., BENINI, L., CATTHOOR, F., AND MENDIAS, J. M. 2004. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st annual Design Automation Conf.* DAC '04.

GAUTHIER, L., ISHIHARA, T., TAKASE, H., TOMIYAMA, H., AND TAKADA, H. 2010. Minimizing inter-task interferences in scratch-pad memory usage for reducing the energy consumption of multi-task systems. In *Proceedings of the 2010 Int. Conf. on Compilers, architectures and synthesis for embedded systems*. CASES '10. 157–166.

GOOGLE. Android. *Google, http://www.android.com/*.

HARA, Y., TOMIYAMA, H., HONDA, S., TAKADA, H., AND ISHII, K. 2008. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE Int. Sym. on*. 1192 –1195.

HEISER, G. 2008. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*. IIES '08.

HUANG, W., YOU, K., ZHANG, S., HAN, J., AND ZENG, X. 2009. Unified low cost crypto architecture accelerating rsa/sha-1 for security processor. In *ASIC, 2009. ASICON '09. IEEE 8th Int. Conf. on*. 151 –154.

IBM. 2005. The cell project. *IBM, http://www.research.ibm.com/ cell/*.

INTEL. 2009. Single-chip cloud computer. *Intel, http://techresearch.intel.com/ ProjectDetails.aspx?Id=1*.

ISSENIN, I., BROCKMEYER, E., DURINCK, B., AND DUTT, N. 2006. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *Proceedings of the 43rd annual Design Automation Conf.* DAC '06. 49–52.

JUNG, S. C., SHRIVASTAVA, A., AND BAI, K. 2010. Dynamic code mapping for limited local memory systems. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE Int. Conf. on*. 13 –20.

KANDEMIR, M., RAMANUJAM, J., IRWIN, J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th annual Design Automation Conf.* DAC '01.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE Int. Sym. on Microarchitecture*. MICRO 30. IEEE Computer Society, Washington, DC, USA, 330–335.

MAYHEW, M. AND MURESAN, R. 2009. Low-power aes coprocessor in 0.18 um cmos technology for secure microsystems. In *Microsystems and Nanoelectronics Research Conf., 2009. MNRC 2009. 2nd*. 140 –143.

MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. 2008. Flicker: an execution infrastructure for tcb minimization. *SIGOPS Oper. Syst. Rev. 42*, 315–328.

PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 1997. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European Conf. on Design and Test*. EDTC '97.

PASRICHA, S., DUTT, N., AND BEN-ROMDHANE, M. 2008. Fast exploration of bus-based communication architectures at the ccatb abstraction. *ACM Trans. Embed. Comput. Syst. 7*, 22:1–22:32.

PYKA ET AL., R. 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.

SHALAN, M. AND MOONEY, V. J. 2000. A dynamic memory management unit for embedded real-time system-on-a-chip. In *Proceedings of the 2000 Int. Conf. on Compilers, architecture, and synthesis for embedded systems*. CASES '00.

SUHENDRA, V., RAGHAVAN, C., AND MITRA, T. 2006. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the 2006 Int. Conf. on Compilers, architecture and synthesis for embedded systems*. CASES '06. 401–410.

SUHENDRA, V., ROYCHOUDHURY, A., AND MITRA, T. 2008. Scratchpad allocation for concurrent embedded software. In *Proceedings of the 6th IEEE/ACM/IFIP Int. Conf. on Hardware/Software codesign and system synthesis*. CODES+ISSS '08. 37–42.

TAKASE, H., TOMIYAMA, H., AND TAKADA, H. 2010. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *Proceedings of the Conf. on Design, Automation and Test in Europe*. DATE '10.

THOZIYOOR, S., MURALIMANOHAR, N., AHN, J. H., AND JOUPPI, N. P. 2004. Hp labs cacti v5.3. *CACTI 5.1, TR, http://www.hpl.hp.com/ techreports/2008/ HPL-2008-20.html*.

TILERA. 2010. Tile gx family. *Tilera, http://www.tilera.com/ products/processors/TILE-Gx_Family*.

VERMA, M., STEINKE, S., AND MARWEDEL, P. 2003. Data partitioning for maximal scratchpad usage. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conf.* ASP-DAC '03. 77–83.

WANG, Z. AND LEE, R. B. 2007. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News 35*, 494–505.

Received July 2010; revised November 2010; accepted March 2011