Multi-compartment: A new architecture for secure co-hosting on SoC

Joël Porquet*† and Christian Schwarz†
†STMicroelectronics
Advanced System Technology
Rousset, France
{firstname.lastname}@st.com

Alain Greiner*
*LIP6-SoC Laboratory
University of Paris-VI, Pierre et Marie Curie
Paris, France
{firstname.lastname}@lip6.fr

***Abstract*—Multi-compartment is a flexible, lightweight architecture for embedded systems that allows multiple protection domains (compartments) to securely share processing, memory and other system resources. Compartments run in physical address space and enjoy direct access to security-critical initiator devices, such as DMA devices, while remaining protected from one another.**

I. Introduction

Co-hosting several standalone software stacks is an upcoming requirement in embedded systems. For example, in multimedia oriented SoCs, recent developments show the growing importance of being able to execute multiple software stacks in parallel (time shared) and to transparently partition available platform resources in a protected way. These stacks can range from applications on top of a Realtime Operating System (RTOS) to small, security-critical conditional access stacks running isolated from a RTOS and up to execution of a rich OS in parallel with a RTOS or a baseband phone stack.

Until recently, embedded systems were not designed to address this secure co-hosting requirement.

Relatively simple platforms without sophisticated (memory) protection mechanisms, are typically equipped with a RTOS executing in supervisor mode and without application separation or separation using a Memory Protection Unit (MPU). The RTOS and its applications run in physical address mode. Often the memory layout is decided by the system integrator and thus remains static throughout the platforms life cycle. More complex platforms can use a rich OS, protecting its applications from one another by means of a Memory Management Unit (MMU). In both cases, the memory protection or isolation between applications is always achieved at processor level. All the memory accesses an application performs are checked and validated inside the processor's boundary: an application is then unable to propagate any illegal memory access across the platform.

This processor-centric property makes the sharing of processing and memory resources efficient for applications running on the same basis (i.e. OS), but is ill-adapted for execution of independent software stacks. Establishing a common basis is especially difficult when confronted with a heterogeneous multi-processor layout. At the platform scale, co-hosting standalone software stacks thus implies a static partitioning of the platform resources without any security.

Yet this secure co-hosting issue has been addressed as "Software Virtualization" first for mainframe servers [1] and later for desktop systems [2][3], and recently for embedded systems [4][5]. In addition to these pure software approaches, hardware support has been added directly within the processor [6][7] and substantially facilitates the virtualization. It is likely this hardware support to appear in embedded processors soon. As a matter of fact, ARM has already begun to impel this mechanism with the TrustZone feature which introduces a "secure world" along the standard "unprotected world" [8]. TrustZone is optimized for and limited to executing a small trusted software stack in the secure world while running untrusted software, typically a rich OS, in the unprotected world.

All these virtualization mechanisms are processor-centric. It means they share the same drawbacks as virtual memory since at platform scale, they also need a common basis to be efficient.

Furthermore, due to performance considerations, an expectation is to allow the Virtual Machines (VMs) direct access to some security-critical devices, such as Direct Memory Access (DMA) devices. Although such a device enjoys full access to the physical address space, it is most of the time hardware only, that is does not embed any trusted software layer. A VM which would directly instruct a DMA device could reach forbidden memory areas. In this perspective of enabling direct access to devices while keeping a strong memory isolation, Intel and AMD have begun to market a mechanism of virtual memory for devices [9][10]. Thanks to an input/output memory management unit (IOMMU), the virtual address space of a VM is extended down to the device. This solution, only intended for desktop systems at this time, requires once again a common basis for the address space virtualization to be coherent.

The μspider [11] project includes an example of a security model coherent with the whole platform. The feature, integrated within a Network-on-Chip (NoC), checks the hardware transactions between physical devices and authorizes them if they comply to the configuration. However, the granularity of the access control is the entire device. This lack of flexibility does therefore not allow different rights to memory areas within the same physical memory bank. Besides, this filtering method is also incompatible with co-hosting, where several software stacks are able to share the same initiator devices.

As a consequence, it is necessary to provide a new co-hosting mechanism which offers a security coherent with the whole platform and which keeps a low-impact on the performance.

In Section II, we introduce our concept of multi-compartment which provides a secure, efficient and flexible co-hosting mechanism. The new hardware and software mechanisms required to build this multi-compartment approach are explained in the following section III. Conclusions and future work are presented in section IV.

II. Multi-compartment concept

An ideal mechanism would allow several standalone software stacks to transparently share all the platform's resources, particularly the memory and processing resources. The architecture we present in this paper represents a step towards such a co-hosting system. We call it *multi-compartment* because each standalone application is securely contained in a logical entity called *compartment*. This concept (shown on figure 1) is based on the two following principles.

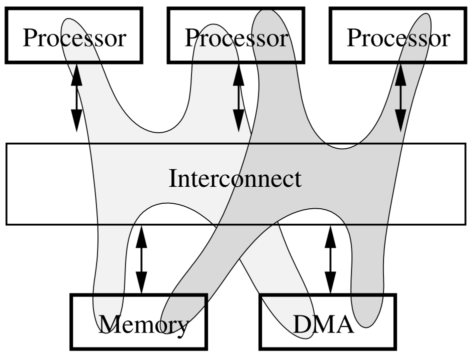

Fig. 1. Co-hosting of two compartments sharing processing and memory resources as well as a DMA device

A. Identification

Co-hosting compartments first requires identifying each compartment accurately at hardware level. From any transaction (in the sense of request/response between an initiator device and a target device through the SoC interconnect), this identification must allow to track back the transmitting compartment. For that, we introduce the *compartment identifier* (CID) which allows to couple any transaction with its associated compartment. Concretely only initiator devices are concerned since they are the only ones capable of initiating transactions. They must be able to provide the exact identity of the compartment on behalf of which they launch any transaction. This CID must be available directly at the device's output, that is to say at the link between the device and the interconnect.

This identification enables then to perform an access control for each transaction.

B. Protection

Protecting compartments from one another means actually protecting their code, data, exclusive uses of peripheral devices, etc against misuses. Since these assets are in fact areas in the shared address space, the access control merely corresponds to access rights on address regions (e.g. read/write/execute). In combination with the CID associated to each transaction, the isolation is achieved by confronting all the transactions against different sets of access rights, one set per compartment. As a result, the granularity of the access control becomes the compartment.

III. Multi-compartment architecture

In this section, we give an overview of a possible implementation for this multi-compartment concept. From this implementation, we detail how the CID can be propagated through the platform, then how the multi-compartment can be managed by initiator devices, i.e. processors and DMA devices, and what flexible memory protection mechanism can be applied.

A. CID propagation

At the heart of multi-compartment is the compartment identifier which tags all the transactions, at hardware level. Our approach consists in extending the interconnect interface with a new signal. Actually, standard communication protocols, such as VCI or OCP, already offer such a feature. The VCI protocol [12] provides a field named TRDID which "can be used as an extension to the SRCID (identifier of the physical device) to create logical, or virtual devices". This addresses exactly our issue where initiator devices must broad-cast different logical identity according to the compartment they are running at a certain time.

B. Processor devices

At hardware level, a new control register is added, to hold the compartment identifier. The value of this register tags every transaction (instruction/data) the processor issues in the platform. Besides, it is preferable to also tag the whole memory chain, i.e. the cache and the write buffer, with the CID value. This avoids memory chain flushing upon a CID register switch, thus reducing performance degradation. Figure 2 illustrates the hardware modifications.

For specialized processors providing only one execution mode and/or processors which only run one compartment, the CID register value can be static. This can be done either at design-time or configured at boot by a trusted software. On processors providing several privilege levels, a *Trusted Software Agent* (TSA) runs at the most privileged level and is in

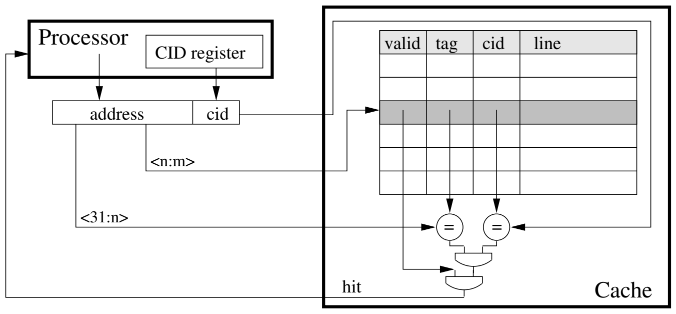

Fig. 2. Example of a request in a simple direct-mapped cache

charge of two tasks: scheduling the compartments which run at least one level of privilege above and updating the CID register value accordingly. If compartments are applications then the TSA can be any trusted rich OS. If complex compartments, such as commodity OS, are targeted then the TSA can be any trusted hypervisor. In both cases, the TSA implementation is significantly simplifies, since it does not have to care about memory protection management.

Those modifications at hardware and the CID register management at software levels, are quite inexpensive. On the processor side, there is only one register to add which value is modified by the TSA when switching. On the cache side, the increase due to the new CID tag does not impact its whole size (only 1.5% increase on a simple 4K cache with a 8-bits CID field) and the combinatory logic is slightly modified to take the CID into account for the "hit" computation.

C. DMA devices

In the case of a DMA device, we apply an inheritance concept. When a compartment directly instructs a DMA device, its identity is included in the programming transaction via the new CID signal. The device is able to reuse the same CID at the time of the effective data transfer. The memory accesses performed by the DMA device on behalf of the requesting compartment will inherit of the same access rights than those initially defined for this compartment. Therefore through a minor modification, we benefit of the compartment identity propagation into DMA devices.

The DMA device is also able to support concurrent access from several compartments. Let us consider a simple DMA device with three configuration registers (one for the start address, one for the end address and one for the length of the transfer). If this register set is replicated into several virtual sets, each compartment is able to configure its own set independently. The CID value tagging the programming transactions is used as an efficient demultiplexer. Next there are different possible implementations but basically the DMA device can decide what transfer to perform thanks to any arbitration policy. See figure 3. Consequently, multi-compartment allows building an inexpensive self-virtualized device.

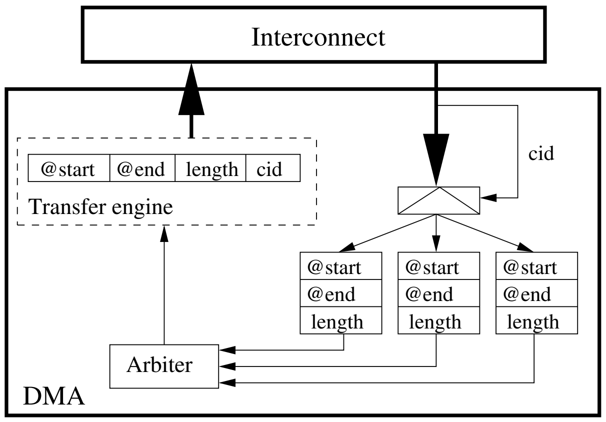

Fig. 3. Self-virtualized DMA device

D. Protection mechanism

Several compartments are able to run concurrently on the platform. In fact, this means that each transaction (issued by an initiator device) travelling across the platform belongs to a compartment. For a proper protection and in order to reach a maximal control on all memory accesses, the filtering mechanism is thus positioned at the heart of the platform, the communication network. This mechanism is thus represented by a hardware module located in the network interface controller as illustrated on figure 4.

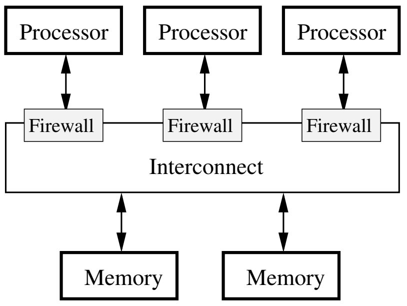

Fig. 4. Example of a platform equipped with firewall modules

Abstractly, the filtering algorithm is represented by a permission table which is indexed by the memory address of the transaction and the compartment identifier, and which contains the defined access rights for each combination of these two entries. Since one of the requirement is flexibility, that is many compartments sharing the address space in a multitude of parts with different access rights each, we assume this table is stored outside of the firewall. It is recommended to store it in on-chip memory, for performance and security reasons. In this perspective, the firewall module is thus composed of a small cache (called *permission lookaside buffer*, PLB) and dedicated Finite State Machine (FSM) to fetch the required entry from the permission table in case of PLB miss. As shown on figure 5, when a transaction arrives, the PLB is looked up. If the PLB has the permission information concerning the tuple (address, CID) the transaction is granted or dismissed accordingly. Otherwise, the hardware FSM walks the permission table in memory and refill the PLB with the missing information.

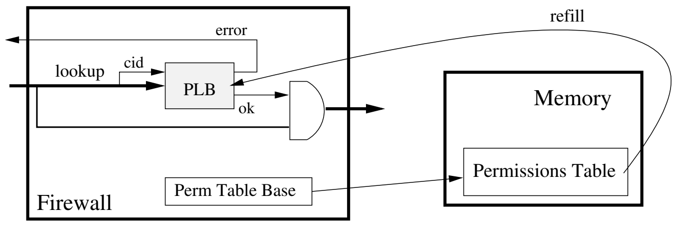

Fig. 5. Internal architecture of the firewall module

[13] discussed alternative layouts of the entries in the permissions tables. Basically, there are two approaches. Firstly the segmentation approach is a "linear array of segments ordered by segment start address". Although this approach supports variable segment sizes, the lookup time can turn out to be tremendous, as well as the update. Secondly, the paged approach is a "forward mapped page table". The worst-case lookup is deterministic and the management is quite easy, while defining permissions for variable-sized pages is not supported. Intended for other purposes, some projects such as Mondrian Memory Protection [13] or Guarded Page Table [14] have already addressed this kind of problematic and present efficient, fined-grained protection schemes that could definitely be adapted to our architecture.

The location of the firewall modules in the platform can be relevant. The major advantage of being located at the initiators side (figure 4) is that modules are able to prevent any denial of service in the interconnect, if ever an initiator device keeps performing unauthorized accesses. Otherwise, the location must be chosen according to the platform layout. In order to reduce the number of modules, it is advised to locate them on the side which counts fewer devices.

E. Platform management

In our approach, the processing resource is locally managed through *Trusted Software Agents* (renamed Local TSA or LTSA), in order to schedule compartments and keep the CID updated. However, this local management is typically complemented by a *Global Trusted Software Agent* (GTSA) which deals with the whole platform. The GTSA especially supervises the address space partitioning management through the "on-the-fly" definition of the permission table. Moreover, it dynamically handles the creation/destruction of compartments and the firewall modules configuration (definition of the permission table base pointer).

Yet this layered trust is conceptual and can be implemented differently according to the context. For a platform which is entirely covered by a common basis, such as a RTOS, the RTOS is able to act as the LTSA on all processors and as the GTSA for the platform management. For a platform shared by two RTOSes, they are able to act as LTSAs on the processors they have under control, while one of them acts as the GTSA.

IV. Conclusion

We have presented a proposal for securely co-hosting several protection domains (compartments), called multi-compartment. By representing logical entities at the hardware level, the protection policy turns out to be much more flexible, lightweight and coherent with respect to embedded systems aspects, in particular heterogeneous multi-processing. Compared with other approaches, compartments are able to run in physical address space and enjoy direct access to security-critical initiator devices, such as DMA devices, while remaining protected from one another.

Our upcoming work is to complete the prototyping of this architecture. Along with the concept validation, it will allow to get accurate performance results. We will also get into details concerning other peripheral devices management.

References

[1] R. J. Creasy, “The Origin of the VM/370 Time-Sharing System,” *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981.
[2] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, “Xen 3.0 and the Art of Virtualization,” in *Proceedings of Linux Symposium 2005*, July 2005.
[3] VMWare. [Online]. Available: http://www.vmware.com
[4] VirtualLogix. [Online]. Available: http://www.virtuallogix.com
[5] O. K. Labs. [Online]. Available: http://www.ok-labs.com
[6] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, “Intel Virtualization Technology,” *Computer*, vol. 38, no. 5, pp. 48–56, 2005.
[7] Advanced Micro Devices, Inc., *AMD64 Virtualization Codenamed “Pacifica” Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.
[8] T. Alves and D. Felton, “ARM TrustZone: Integrated Hardware and Software Security,” July 2004. [Online]. Available: http://www.arm.com
[9] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, “Intel Virtualization Technology for Directed I/O,” *Intel Technology Journal*, vol. 10, no. 3, August 2006. [Online]. Available: http://www.intel.com/technology/itj/2006/v10i3/
[10] Advanced Micro Devices, Inc., *AMD I/O Virtualization Technology (IOMMU) Specification*, February 2009, PID 34434 Rev 1.26. [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf
[11] J.-P. Diguet, S. Evain, R. Vaslin, G. Gogniat, and E. Juin, “NOC-centric Security of Reconfigurable SoC,” in *NOCS ’07: Proceedings of the First International Symposium on Networks-on-Chip*, 2007, pp. 223–232.
[12] VSI Alliance, *Virtual Component Interface Standard*, April 2001, version 2, OCB 2 2.0.
[13] E. Witchel, J. Cates, and K. Asanović, “Mondrian Memory Protection,” in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002, pp. 304–316.
[14] J. Liedtke, “Address space sparsity and fine granularity,” in *EW 6: Proceedings of the 6th workshop on ACM SIGOPS European workshop*, 1994, pp. 78–81.