- WHITE PAPER, June 2012 -

EmbeddedXEN: A Revisited Architecture of the XEN hypervisor to support ARM-based embedded virtualization

Prof. Daniel Rossier, PhD

Reconfigurable Embedded Digital Systems Institute

School of Business and Engineering in Canton de Vaud

Yverdon-les-Bains, Switzerland

daniel.rossier@heig-vd.ch

Abstract—**Embedded virtualization is gaining increasing interest in embedded system industry, as microcontrollers are endowed with high performance ARM cores and powerful multi-media peripherals. This paper presents a novel approach in virtualization mechanisms to address particular needs in embedded systems, such as dealing with heterogeneous ARM cores and keeping execution overhead as low as possible. We introduce *EmbeddedXEN*, a revisited virtualization framework based on the existing XEN hypervisor. Para-virtualization of guest operating systems turned out to be the right choice in terms of performance and capabilities to deal with device heterogeneity. Adaptations of guest OSes are kept minimal and low-level events such as IRQs are processed efficiently.**

Keywords: XEN hypervisor, Linux, embedded virtualization, ARM cores

I. INTRODUCTION

Embedded virtualization makes the execution of multiple instances of operating systems on the same microcontroller possible. While virtualization remains of huge interest in the classical desktop/server environment, recent ARM-based microcontrollers [1] are particularly powerful and can afford to run much more complex execution environments than ten years ago. In this context, virtualization can now be considered in embedded systems such as smartphones, gateways, set-top boxes, realtime platforms, etc. in order to provide end users with new features [2] [3]: highly secure execution environments, hardware consolidation and resources partitioning, etc.

Several research projects and company products have been developed in order to provide a virtual machine monitor *or hypervisor*, so as to give end users a complete framework for managing several instances of operating systems, mainly on ARM cores, with particular emphasis on security - or isolation - aspects.

On the desktop side, one powerful *open source* hypervisor is XEN [4]; XEN is perfectly well supported on IA-32/64 SMP platforms and provides the end user with a lot of *userland* tools to manage *virtual machines* and physical resources. XEN has been originally developed to run several *Linux VMs* on servers, and is *de facto* not tailored to embedded platforms[1].

This paper introduces *EmbeddedXEN*, a hypervisor based on XEN source code and devoted to ARM cores with a revisited architecture to achieve a virtualization framework tailored to embedded systems. In particular, *EmbeddedXEN* supports the heterogeneity of ARM-based devices, meaning that cross-compiled OS and applications intended to run on a particular ARM platform can be ported on a different platform with more recent ARM CPUs, without requiring any modification of the user-space applications.

Our framework has been deployed on devices supporting v5, v6 and v7 ARM instruction sets, among which Logitech Squeezebox Touch device [2] and HTC Desire HD Smartphone.

A. *Embedded Hypervisors*

Using virtualization in embedded systems brings about different challenges than the desktop, server or laptop universe. Of course, the main difference resides in the power and capacity of embedded CPUs (microcontrollers) which in most cases are RISC- based architectures and do not have support for virtualization in their instruction set yet. However, ARM have already announced cores (namely Cortex-A15) with such support; they should be available on the market in early 2013.

Hence, embedded virtualization requires the guest operating system to be slightly modified in order to set up interactions with the hypervisor, which is the first underlying software layer close to the hardware[3]. This mechanism is known as para-virtualization and well supported by XEN for *Pentium* architectures.

B. *Porting XEN on ARM*

We started our research investigations in early 2007 with the objective to port the *XEN* hypervisor on ARM platforms. In this context, we started the development of *EmbeddedXEN* with the primary objective to enable the

[1] A port for PPC was done, but is no longer supported.

[2] The SqueezeBox Touch is an advanced multimedia player sold by Logitech which runs SqueezeOS (based upon Linux). Source code is available at http://wiki.slimdevices.com

[3] In that sense, XEN is considered as a *type-1* hypervisor.

execution of a hard realtime domain in parallel with a non-realtime domain. Both domains were based on *Linux*[4]. During the last two years, we have focused on the possibility to let coexist a *native* operating system and a second third-party OS; the latter typically runs on a different device than the host device, therefore with different peripherals and capabilities, as well as a different ARM core.

In this paper, we outline the main differences between the original XEN and *EmbeddedXEN*.

II. EMBEDDEDXEN ARCHITECTURE AND COMPONENTS

A. *The Big Picture*

A major difference with other ARM hypervisors is that *EmbeddedXEN* supports only two guest operating systems. The first *guest* OS is called *dom0* and consists of a para-virtualized version of the *native* OS running on the target device. The second OS is called *domU* and represents the OS running on the third-party device, which has nothing to do with the device running the host OS. The overall architecture including the hypervisor, *dom0* and *domU* is depicted on Fig. 1.

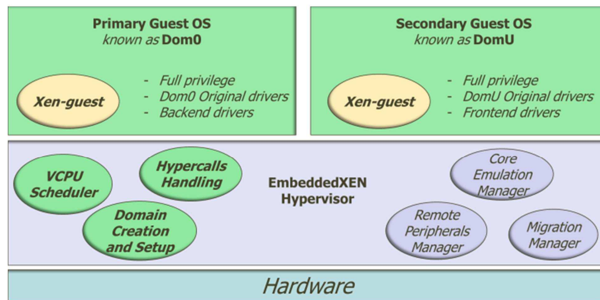

Figure 1. Overall architecture with two guest OSes

Para-virtualization of the guest OS at the kernel level is required for two reasons: on the one hand, ARM CPU does not support virtualized instructions like Intel CPUs with VT-extensions or AMD CPUs yet; the Cortex-A15 announced by ARM, which will have such support, is not commercially available yet. On the other hand, restricting CPU activities in embedded systems is relevant in order to reduce power consumption. In *EmbeddedXEN*, modifications in the guest kernel to support virtualization are minimized.

The *Xen-guest* contains the API and other functions to control the interactions between the hypervisor and the guest OS kernel.

B. *Single Multikernel Binary Image*

The build system of *EmbeddedXEN* produces a single binary image which contains the hypervisor and both *dom0* and *domU* which are concatenated and separated by specific markers, as shown on Fig. 2.

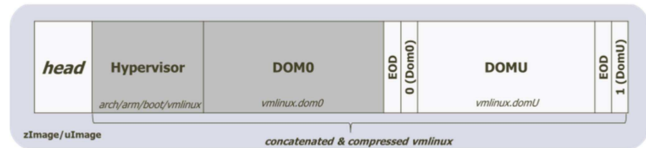

Figure 2. Multi-kernel single binary image

This image is a single file which can be easily deployed into the flash or transferred via NFS. Since the image is built upon similar scripts as those used by the *Linux* build system (as well as the compression/relocation code), the image can be loaded and started from any *Linux*-compatible *bootloader* (such as *U-boot* for instance, or even *HBoot* used by HTC smartphones). The image's bootstrap is therefore similar to *Linux*, except that the first executed code is the *hypervisor* code. For this reason, the code architecture and files & directories organization remain identical to *Linux*.

The hypervisor parses the file to extract the domains (normally *dom0* and *domU*) and put the images on top of the RAM. Those images are stored in ELF-format (*vmlinux*) and are parsed in turn by the ELF loader during the domain setup.

C. *Memory Address Spaces and Isolation*

As already mentioned, one of the main differences between our virtualization framework and others resides in security aspects of memory management. In our environment, the physical RAM is allocated to the guest by the hypervisor, depending on size configuration, but the mapping of virtual memory to this portion is delegated to the guest. Normally, a translation table like a *physical-to-machine* table is used in order to protect the real (physical) memory from the guest, and provide it with intermediate address: the virtual address is therefore translated into an intermediate address (*physical* or *pseudo-physical*), and a second translation is needed to reach the final (*machine*) address. To make these two translations, either a hardware-assisted MMU is necessary, or the page table setup is managed by the hypervisor and the *physical-to-machine* table is used to configure the page table entry with the right frame number.

We do not use such a mechanism in *EmbeddedXEN*; since we trust the guest OS kernel, the manipulations of page tables are made directly by the guest. Only simple linear mapping of the allocated RAM is performed during the domain set-up. Fig. 3 shows the mapping of virtual address spaces onto the physical space.

[4] Hard realtime OS consisted in using the *Xenomai* patch for Linux.

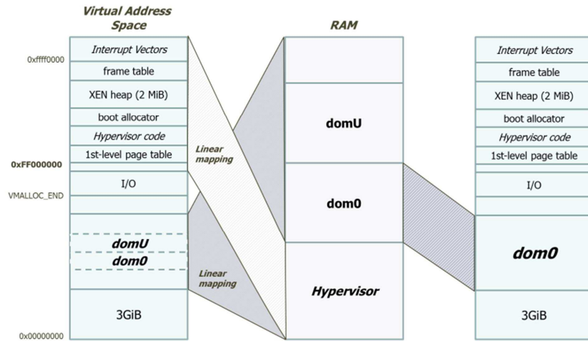

Figure 3. Virtual and physical address mapping

The virtual address space on left is set up during the hypervisor bootstrap. A linear mapping of the available physical RAM is performed at the virtual address *0xc0000000* (where the kernel space starts). The hypervisor itself resides at *0xff000000* (as in the original *XEN*). The hypervisor can therefore have the visibility of the two domains, initialize the guest page tables and prepare them for bootstrap easily.

On the right side, the virtual address space corresponds to the one used when *dom0* executes itself. The "traditional" *Linux* address space configuration is used without any particular modifications; however, only a portion of the RAM is passed to the guest through the RAM descriptor initialized by the hypervisor at the guest boot time. This configuration also shows that there is no possible direct interaction with the other domain (*domU*) and both domains are thus isolated.

The same virtual address space is set up for *domU*.

D. Para-virtualization and Xen-Guest API

The para-virtualization of guest domains consists in modifying files in the guest kernel source tree in order to support the interactions between the hypervisor and the other domain correctly. There are approximately 73 files to be slightly modified (or added). Table I outlines a summary of modified files to give an idea about the patches to be made.

TABLE I. SUMMARY OF PATCHED FILES

File	# files	*Para-virtualized functions*
arch/arm/	16	Bootstrap, IRQs, setup, traps, memory init
drivers/	27	Block, input, serial, video
include/	15	Memory, irqs, system, mach-related
init/	2	Main entry point (retrieving of start info from the hypervisor, slightly modified init)
sound/	9	Support for the sound in *domU*
kernel/	4	IRQs and time-related adaptations

In addition, a new directory called *xen-guest* must be added in the guest (a symbolic link is actually added) which enhances the guest with convenient APIs in order to perform *hypercalls*, process IRQs (which are known as event channels) as well as manage all interactions on *xenbus,* which enables messages and events exchange between the guest domains (see also Section IV.A). No user space tools are required in *EmbeddedXEN,* since the domain images are included in the single binary; furthermore, *xen-guest* includes all necessary *init* code such as *xenbus* device initialization code.

As shown on Fig. 4, all *xen-guest* code runs at the same level as the rest of kernel code. Following a monolithic architecture, the insertion of this code within *Linux* kernel makes calls to other subsystems straightforward and efficient. For instance, the processing of IRQs relies on a polling mechanism which interrogates an event bitmap; whenever an event occurs, the IRQ is propagated along the guest kernel by calling the corresponding *asm_do_IRQ()* function known in the *Linux* kernel which is normally called by the interrupt vector as defined in *entry-armv.S*.

It has to be noted that the *xen-guest* code – which basically corresponds to *pvops* of XEN - is common to *dom0* and *domU*.

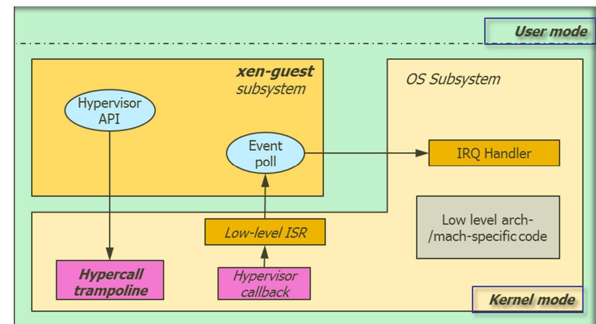

Figure 4. Architecture of a para-virtualized guest OS as *dom0* or *domU*

Normally, a guest kernel should run in a *semi-privileged* mode while the hypervisor runs in a full privileged mode. Architectures based on *x86* for example use the notion of rings to give the necessary protection levels to the hypervisor, guest kernel and *userland* applications. On ARM architectures however, such mechanisms do not exist yet[5]. As we will see in Section III.A, the guest kernel is running in supervisor mode as for the hypervisor.

Therefore, the *hypercall* mechanism at the guest level should normally lead to use a *syscall*-specific instruction such as *SWI* to execute some code at the hypervisor level. In *EmbeddedXEN*, we do not use this approach since the kernel is already running in *privileged* mode when the *hypercall* is initiated: jumping to a pre-defined address in the hypervisor (corresponding to the *hypercall* entry point) is sufficient. Typically, such address can be acquired in the *start info* structure retrieved at the early stage of the kernel bootstrap.

[5] Next generation of ARM Cortex-A15 based on ARMv7+ virtualization support will have such facilities, but not commercially available yet.

III. INTERACTION SCHEME

The presence of a user mode and *pseudo-user* mode at the kernel level constitutes another fundament of *EmbeddedXEN,* which is detailed in the following sections.

A. Double stack and pseudo-user mode

In the early implementation of our framework, the code was mainly based on the Samsung *Xen-arm* project[5]. However, we faced an important problem when we had to deploy the framework on a *Freescale i.MX25* based device: accessing the I/O address from the user mode was complicated, or even impossible. We therefore devised a novel approach based on the assumption that the guest kernel code was mature and secure enough to grant the guest kernel with privileges to run in supervisor (kernel) mode, as it is normally the case without virtualization layer. Besides the possibility to perform direct read/write access to I/O addresses, the advantages of such an approach are multifold: the use of *hypercalls* can be considerably reduced and therefore avoid some complicated call paths between the hypervisor and the guest. The overall performance is therefore improved. Another advantage is the small number of files to be modified; I/O accesses are kept unchanged in the original Linux and upgrading to new versions is facilitated.

One particular problem of using the same mode at the hypervisor and guest levels concerns the use of the stack pointer; on ARM CPUs, different stack pointers are used according to the execution mode (USR, IRQ, SVC, etc.) [6]. If the execution mode is restricted to the *SVC* (kernel) mode, only one stack pointer – therefore one stack – is used which leads to some complications in the stack handling, especially during domain context switches. The introduction of two different stacks as well as a *virtual pseudo-user* execution mode is required as shown on Fig. 5.

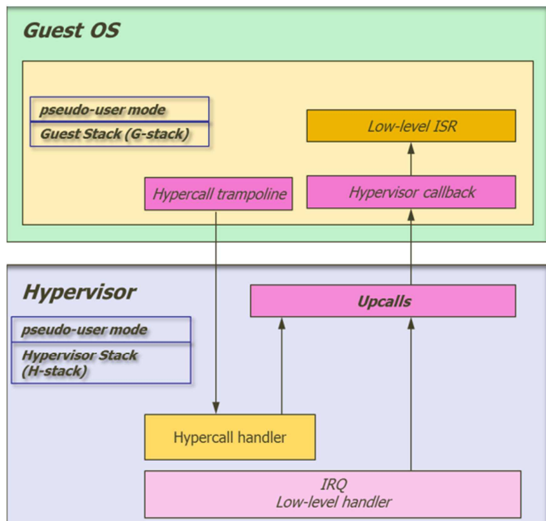

Figure 5. Double stack with *pseudo-user* mode handling

Let us examine the different cases once the guest has been booted up by the hypervisor, and is running. The hypervisor has set a *virtual* execution mode to *pseudo-usr* mode when the guest kernel is started, with its associated *G-stack*. By the way, the two domains have their own stacks, i.e. *G-stack(dom0)* and *G-stack(domU)*. Note that when the guest is running in *user* mode, a dedicated stack is used and no interference may occur with the kernel stack. What is now happening when an interrupt (IRQ) occurs?

We now consider the case where an IRQ happens while *dom0* is running (either in *user* or *SVC* mode): the ARM CPU automatically switches to the *IRQ* mode, which has its own stack, and the low-level IRQ handler is executed within the hypervisor context; as it is the case in *Linux kernel*, this mode will be left as soon as possible to work in the *SVC* mode, which is common to any situation when some kernel code must execute. Therefore, a switch to *SVC* is done after configuring the *SVC* stack correctly[6]. The running stack is still *G-stack(dom0)* when the *SVC* IRQ handler begins its execution. At this point, the hypervisor saves the *G-stack(dom0)* (current address stored in the *stack pointer (sp)*) into its *guest context* descriptor, switches to the *virtual* kernel mode, and resets *sp* to the *H-stack* which has been saved at a fixed memory location right before giving the hand to the guest. The hypervisor can then perform the handler code with the right stack, without any interference with the guest kernel stack (*G-stack*). Once the handler completes, the hypervisor can execute the *upcall* code, which leads to restoring the preempted guest. Of course, one important action of the *upcall* will consist in storing the current *H-stack* in its fixed location, setting *sp* back to *G-stack(dom0)* saved previously in its *guest context* descriptor, and finally resetting the *virtual* execution mode to *pseudo-user* mode. It has to be noted that the guest which is restored on an *upcall* path always restarts its execution in kernel mode, i.e. *pseudo-usr mode*: indeed, either an IRQ must be propagated to the guest, or the guest executes an hypercall at its kernel level, and pursues its execution after the trampoline.

Now, there are two typical scenarios which can take place when the execution is still at the hypervisor level: for example, a domain context switch could occur if the scheduler decides; without entering into details, the scheduler could have been invoked during the *upcall* path. In this case, the restored stack is the corresponding *G-stack* - *G-stack(domU)* in our case - and the stack of *dom0* remains consistent. The other scenario is related to the re-enabling of interrupts along the *upcall* path; interrupts are normally re-enabled to improve system reactivity; another IRQ may occur before the guest has been restored. In this case, the behavior can be differentiated thanks to the use of the *virtual* mode: the hypervisor is indeed running in *kernel* mode, and no stack permutation is needed. Again, the stack remains consistent.

This approach allowed us to deal with a unique (CPU) kernel mode (*SVC)* and to differentiate the execution at the hypervisor and guest level.

[6] Switching to *ABT, UND, FIQ* leads to the same behavior.

B. IRQs and Upcalls

Compared to the original *XEN*, IRQs in *EmbeddedXEN* also rely on the event channel bitmap approach, i.e. an event channel, represented by a single bit, is associated with an IRQ. When an interrupt occurs, the hypervisor performs machine-dependent low-level handling code, depending whether the IRQ is a *physical* IRQ or *virtual* IRQ. The corresponding event channel bit is set, the interrupt line is acknowledged and masked, and the IRQ is propagated to the guest domain on *upcall* path.

Physical IRQ (PIRQ) is used when the interrupt must be processed by the guest, and is therefore not virtualized by the hypervisor; PIRQs are particularly helpful since device drivers are normally located in *dom0*[7] and must handle IRQs as normal drivers; low-level handlers (*edge*, *level* or *simple*) are normally applied as they are in normal *Linux*. Virtual IRQ (VIRQ) on the other hand is an IRQ *generated* by the hypervisor (following a real interrupt or not). The VIRQ has its unique identifier and is propagated to the guest as if it was a real interrupt. To do that, the *upcall* consists of jumping to a specific location in the guest by means of a specific callback address which is set by the guest via an *hypercall* during its bootstrap (using a *start_info* structure). As a part of the *xen-guest* code at the guest level, the event channels are processed one after the other, by triggering the corresponding IRQs (the standard *asm_do_IRQ()* function is called). Once all event channels have been processed, *dom0* executes a *hypercall* to ask the hypervisor to interrogate the IRQ controller again to check for potential pending interrupts. This mechanism avoids unnecessary domain switches, improves reactivity, and even reveals necessary on HTC Desire HD, where a lot of IRQs issued from various peripherals are frequently generated, and cannot be delayed too much.

The guest makes no real differentiation between virtual and physical IRQs; the difference mainly concerns the way the hypervisor manages them. The overall interactions between hypervisor and guest are depicted on Fig. 6.

[7] At the moment, only *dom0* can managed PIRQs, but this should not be always true especially when considering resource partitioning between *dom0* and *domU*.

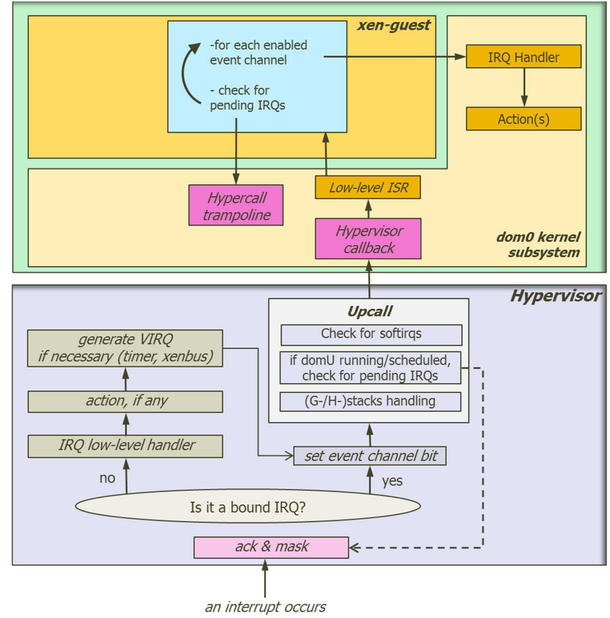

Figure 6. Interactions between the hypervisor and *dom0* when an interrupt occurs

For a better understanding of what happens when an interrupt occurs, let's examine two examples of interrupts: timer interrupt and touchscreen interrupt. Timer interrupts are required by the hypervisor to activate its scheduler periodically, and thus to perform domain switches depending on their time consumption. Timer interrupt therefore cannot be processed directly by *dom0*. But the guest OSes themselves also need such interrupts for their own scheduling purposes; the hypervisor must consequently generate a proper interrupt for each guest - actually a virtual IRQ – by enabling the corresponding VIRQ event channel. Touchscreen interrupts are not of great interest for the hypervisor. Moreover, the *framebuffer* strongly depends on the hardware and needs a particular driver; this driver, as explained in the next section, resides in *dom0*; the interrupt is therefore considered as a physical interrupt bound to *dom0* during its IRQ initialization. Once acknowledged and masked, the hypervisor will directly propagate the interrupt towards the guest via the *upcall* by giving the priority to *dom0* (reschedule of *dom0* is done immediately in this case if *domU* was running before the preemption or newly scheduled).

IV. Drivers, Frontend and Backend

A. Frontend and Backend Drivers

Drivers play a fundamental role in an operating system since they deal with specific hardware and manage all interactions between the hardware and the kernel, and between the kernel and the user space. In a virtualized environment based on *XEN*, the approach with drivers consists in giving the responsibility of dealing with the *real* hardware to *dom0,* which runs in a *semi*-privileged mode, hence with sufficient privileges to perform I/O accesses

properly. In this context, other guest OSes (*domU*) must have a modified version of drivers in order to interact with the driver counterpart in *dom0*. To implement these interactions between drivers from *domU* and *dom0*, drivers are broken down into two components: *frontend* and *backend*. Fig. 7 shows the driver split and the underlying communication mechanism.

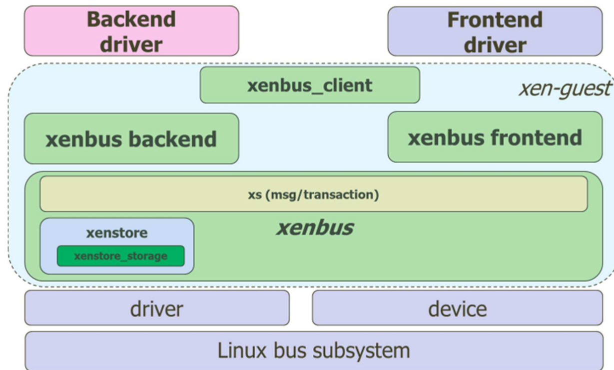

Figure 7. Driver split architecture with frontend and backend drivers

Frontend and *backend* drivers may exchange information asynchronously thanks to a robust communication bus called *xenbus*. The same philosophy is used in *EmbeddedXEN*, however with simplified mechanisms.

All device and driver management uses the standard *Linux* functions, so that user applications can deal with device name exported by the kernel (via */dev* or */sysfs* for example) and access drivers seamlessly. No particular change at this level is required.

B. Xenbus and Inter-domain Communication

The exchange of data between the two domains mainly consists in using a central repository called *xenstore* which is located and managed within *dom0*. *Xenstore* is a kind of in-memory database which stores some *key-value* pairs useful for device and driver management. All available devices are created within *xenstore* at the boot time, and *xenbus* performs calls to probe functions during initialization for each entry found in the database. Any kind of data, such as *framebuffer* properties or block size, can be inserted and queried by the *frontend* and *backend* drivers.

The *xenbus* interactions are managed by means of different threads, namely one thread related to the bus communication system, and another related to *xenstore* in *dom0*. The threads are using a memory page containing two separate ring buffers which store requests/responses issued by *dom0* or *domU*.

Asynchronous requests/answers can be handled with dedicated event channels. When a domain sends a request to the other, *xenbus* invokes an hypercall in order to trigger a virtual IRQ which corresponds to the dedicated event channel. On the other domain, *xenbus* reads the corresponding ring buffer on arrival of the IRQ and performs the transaction. The *xenbus* organization is depicted on Fig. 8.

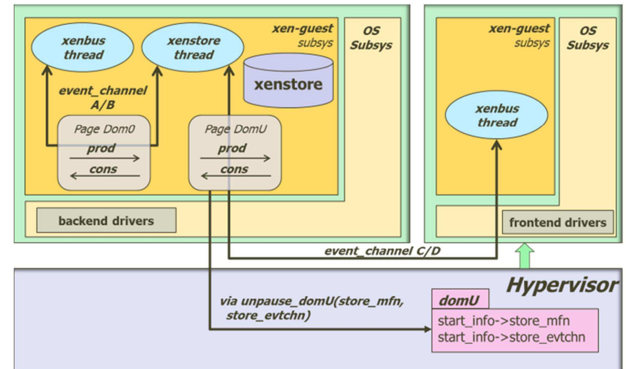

Figure 8. Bus communication with *xenbus* between *dom0* and *domU*

According to this architecture, frontend drivers correspond to the driver part in *domU,* while backend drivers are the part in *dom0*. The split of original drivers is achieved by adapting the low-level driver of each guest OS: on the backend side, the driver must handle two different origins of request to the hardware, as to dispatch the request from the hardware to the right domain (or broadcast it). On the frontend side, the low-level driver intercepts the requests and forwards them to the backend using *xenbus*. On the contrary, when data are received from the hardware and are ready to be transmitted to *domU*, the frontend is informed via the event channel that data are available. However, for performance reasons, all data do not transit via *xenstore* and *xenbus*, but are stored in dedicated pages which are shared or *copied* between domains. This powerful mechanism is based on *grant tables* and explained in the next section.

Since initial entries in *xenstore* are inserted at the bootstrap of the kernel, there is no need to have user space tools to manage *xenstore* as it is the case with the original XEN. So far, no particular need of having dynamic creation/deletion of entries has been identified.

C. Page sharing and grant tables

Sharing pages between *dom0* and *domU* is sometimes necessary to support interactions taking place within drivers. For example, both domains use the same physical *framebuffer*, but with totally different contents. According to XEN architecture (and hence EmbeddedXEN), only *dom0* has access to the real (local) hardware. Memory pages used by the *framebuffer* belong to separate domains, but *dom0* needs access to the *domU* pages. The access to the local storage device (SD card or flash) relies on the same principle: the *dom0* block device driver needs access to physical *domU* pages so that *domU* can mount its root file system.

In traditional virtualization approaches, these interactions must be highly secured and therefore under the control of the hypervisor in order to guarantee the isolation principle between the domains. The sharing and transfer facilities must rely on dedicated hypercalls. In *EmbeddedXEN* however, the approach is slightly different: all interactions take place via the *xen-guest* APIs and do not need any involvement of the hypervisor. Furthermore, the virtual memory layout remains based on the original Linux layout with *linear* kernel

addresses and virtual addresses. In addition, there is no physical-to-machine address translation table. The mechanisms are much simpler being aware that we do not rely on a strong secured isolation of memory spaces between domains, as it is the case with the original XEN. Fig. 9 shows how a domain can access pages from the other domain. Such pages are referred to as *foreign* pages.

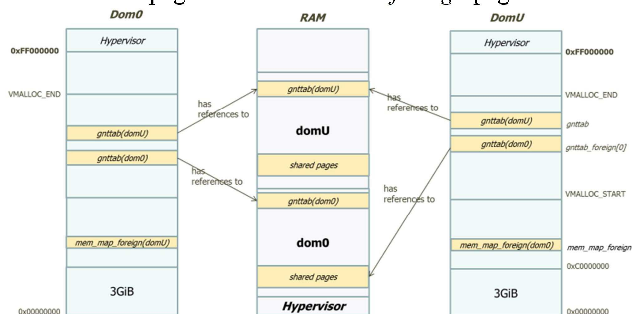

Figure 9. Organization of grant tables between dom0 and domU

At initialization time, each domain receives the page frame number (*pfn*) of the foreign grant table, which is virtually mapped onto its virtual address space. Grant tables are allocated within the *vmalloc* area, therefore "normal" virtual addresses can be used, even if the physical page number is outside the RAM allocated to the domain. This is an important point; such mappings should normally resort to a physical-to-machine conversion table leading to a secondary address translation. It is not the case in our context.

Whenever a page needs to be shared with the other domain, the originating domain requests an association between a grant reference and the corresponding *pfn*. This association is stored within the grant table and the grant reference is passed to the other domain via *xenstore*. The target domain (which needs to access the shared page) must retrieve the *pfn* from the foreign grant table, and map it within its virtual address space. Another particularity has to be highlighted: some macros doing conversions between virtual/physical addresses and their corresponding *struct page*[8] addresses (or vice-versa) are based on offsets from the frame table base; therefore, these macros have been slightly adapted: if the *pfn* is outside the RAM space, the *struct page* is found in a shadow (foreign) frame table, which is allocated at initialization time. This is necessary to maintain coherence when using address conversion functions.

As previously mentioned, only virtual addresses (not linear) can be mapped, since conversion functions between virtual and physical addresses for this kind of address rely on simple addition/subtraction and do not use page tables. Furthermore, ARM cores give the possibility to map large (1 MByte) sections of memory without the use of a second level page table. For this reason, it is not possible to re-map linear virtual addresses to 4 KBytes pages without compromising the whole section[9].

However, Linux drivers such as block device drivers (with the *bio* subsystem) use memory pages which are allocated in the linear address space. Page sharing as implemented in *EmbeddedXEN* therefore becomes hard to use and page transfer mechanisms are preferable. Page transfer allows a guest to access a real copy of page contents. Pages can therefore be allocated using *kmalloc()* within the linear address space, and linear macros can be used. This time, the contents of the foreign page is transferred in a local page and must be re-synchronized with the foreign page when necessary. A dedicated *grant copy* function is available in *xen-guest*.

V. Dealing with Heterogeneity

A. ARM Core Hegerogeneity

EmbeddedXEN is mainly focused on various versions of ARM cores which correspond to different instruction sets (v5, v6, v7). Recent cores add some new features and instructions, but most recent ones remain backward compatible [1]. Therefore, binary code compiled on previous instructions sets should run without any problems. However, some functionalities regarding cache management or MMU may lead to incompatibilities and must be treated with care. For example, running a *domU* guest OS on an ARMv7 (HTC-like) which has been cross-compiled on a ARMv6 CPU may lead to some problems when particular cache-related instructions are executed. This problem has been solved by calling dedicated *hypercalls* in the *domU* kernel in the cache handling functions such as *flush_kern_cache_all()*. The real execution of this function is therefore performed within the hypervisor by using cache handling functions corresponding to the host architecture.

Similarly, VFP instructions are slightly different and must be processed carefully. Still, using *hypercalls* to control the execution of VFP remain the best solution.

B. Peripheral Heterogeneity

Of course, dealing with heterogeneity at peripheral level is at the heart of *EmbeddedXEN*. A *domU* guest OS is normally compiled for a foreign device making obvious use of local peripherals, which are not the same as those on the host. We have seen in this paper how device drivers are structured between *dom0* and *domU*; splitting the device driver in a *frontend* part in *domU* and a *backend* part in *dom0* enables the achievement of this heterogeneity as long as the *frontend* is able to provide *virtualized* interfaces to the upper layer and/or user applications. In this context, we used the driver split philosophy proposed in the original *XEN* to cope with various embedded peripherals which can actually provide similar functionalities.

---

[8] *struct page* is the frame entry which is associated to each physical page. All entries are stored contiguously facilitating their access.

[9] In the *xen-arm* project, they configure the entire linear space with a 4 KB mapping granularity which enables mappings at page level. However, it requires a 2nd page table translation which may introduce some latencies in embedded systems.

VI. CONCLUSIONS

We presented the architecture of *EmbeddedXEN,* which is a new virtualization framework tailored to various ARM-based embedded systems such as HTC smartphones or end-user devices such as *Logitech SB Touch.*

EmbeddedXEN differs from the other ARM hypervisors in that the architecture has been greatly simplified and can be used on ARM CPUs based on *v5, v6 or v7* instruction sets. In this sense, the virtualization framework can be deployed on ARM cores which do not support any virtualization mechanisms yet. However, security in terms of supporting various CPU execution modes has been neglected, provided that the Linux kernel is secure enough to avoid undesirable corruptions between host and guest operating system. Memory isolation is guaranteed thanks to adequate handling of virtual address spaces, and appropriate stack handling relies on a pseudo-user mode with distinct (separated) stacks between the guest and the hypervisor.

As of Today, only two OSes are supported: the host OS, which can be *Android*, and a guest third-party OS, which can be taken from another piece of hardware. Heterogeneity of CPUs is based on the fact that recent ARM architectures remain backward-compatible with previous ones. Furthermore, heterogeneity of peripherals between a third-party device and the host device is managed thanks to hardware virtualization of devices. Currently, most of popular peripherals such as block device, serial device, networking, sound, display and touchscreen are fully supported. Virtualization of peripherals is achieved thanks to driver split organization with a frontend driver in *domU* and a backend driver in *dom0*.

EmbeddedXEN is now publicly available on Sourceforge[10] and will be presented during the *XenSummit North America* conference*,* August 2012, in San Diego.

ACKNOWLEDGMENT

I would like to express my profound thanks to the REDS Engineering Team who has been involved in *EmbeddedXEN* for a while, in particular Romain Bornet, Laurent Colloud and Lionel Sambuc, who spent so many hours in solving some complex problems and debugging thousands lines of code in extreme conditions. Thank You!

Finally, I am deeply grateful to the CTI/KTI Swiss Organization who funded a two years research project with an industrial partner which allowed us to pursue our research in the field of embedded virtualization paying a constant attention to the market needs.

VII. BIBLIOGRAPHY

[1] ARM Limited. (2009) Architectures, Processors, and Devices. [Online]. http://infocenter.arm.com

[2] Open Kernel Labs and NICT and University of New South Wales, "The Role of Virtualization in Embedded Systems," in *First Workshop on Isolation and Integration in Embedded Systems (IIES'08)*, Glasgow, UK, 2008.

[3] Fabiano Hessel Alexandra Aguiar, "Embedded Systems' Virtualization: The Next Challenge?," in *IEEE International Symposium on Rapid System Prototyping*, Fairfax, Virginia, USA, 2012.

[4] David Chisnall, *The Definitive Guide to the Xen Hypervisor.*: Prentice Hall, 2008.

[5] Joo-Young Hwang et al., "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones," in *Fifth IEEE Consumer Communications & Networking Conference*, Las Vegas, Nevada, 2008.

[6] Steve Furber, *ARM System-on-chip Architecture*, 2nd ed.

[7] Paul Barham et al., "Xen and the Art of Virtualization," in *19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, New York, USA, October 19-22, 2003.

[8] Thomas R. Gross Mathias Payer, "Fine-Grained User-Space Security Through Virtualization," in *Virtual Execution Environments (VEE'11)*, Newport Beach, California, USA, March 9–11, 2011.

[10] https://sourceforge.net/projects/embeddedxen