International Journal of Advances in Computer Science & Its Applications
Volume 6 : Issue 3 [ISSN 2250-3765]
Publication Date : 30 December, 2016

System Virtual Machines in the Context of Reconfigurable Computing

Marcel Eckert, Jan Haase, Dominik Meyer, Bernd Klauer

Abstract—**System virtualization techniques are broadly used to distribute computing power, to boot different operating systems virtually on top of a host OS, to enable security features like encapsulation or to provide redundancy and migration concepts. Virtual machines (VMs) targeting different Instruction Set Architectures (ISA) are usually implement by efficient emulation or binary translation techniques.**

This paper uses reconfigurable hardware to support (guest) operating system virtualization and not virtualization of hw-accelerators inside an operating system. The main idea of this paper is to replace emulation / binary translation by (re)configuration. Instead of emulating a processor-ISA on a CPU with another ISA, the required guest-processor will be configured into an FPGA and booted with a guest operating System (OS) as requested by a VM manager.

This approach allows to overcome limitations of conventional system virtualization approaches, like the Popek and Goldberg theorem and provides the possibility to strengthen system security by means of hardware. The presented main idea is implemented and evaluated with a proof of concept demonstrator.

Keywords—**FPGA, Virtual Machine, Reconfigurable Computing**

I. Introduction

Reconfigurable hardware is a promising approach to turn increasing transistor densities into computing performance. Hence, reconfigurable computing has been an emerging research field for over a decade. The application areas of reconfigurable computing focus on accelerator units for embedded systems and high performance computing. A challenging question in the field of reconfigurable computing is, how to simplify the error prone development process of hardware component, so that it is also possible for a software developer to create applications that can take advantage of reconfigurable hardware in terms of execution speed. One approach to achieve this is to provide High-Level-Synthesis compilers, that allow to directly translate parts of an application written in a high level language like C/C++ or Java into hardware.

Another way is to provide abstractions and standardizations of reconfigurable hardware to a sw-developer, so that he or she can implement their applications according to well known interfaces and techniques but without the need to get too much insights into the underlying hardware interdependencies.

Marcel Eckert, Jan Haase,Dominik Meyer, Bernd Klauer
Helmut-Schmidt-University

Hamburg, Germany

Hence, operating systems have been developed in the context of reconfigurable computing to support this approach. This is the common way to interpret the terms virtualization and reconfigurable computing in combination.

Another still emerging research area, coined with the term ”virtualization” is Operating System Virtualization. Research in the area of operating system virtualization is focusing on the minimization of the computational performance loss due to the applied virtualization techniques. Depending on the capabilities of the host processor(s), one of the techniques discussed in the following could be applied to virtualize a guest operating system.

If the guest operating system is expecting the same ISA, as the host processor provides, things are more easier, because direct native execution of the guest operating system is possible. However, care must be taken about the abilities of the underlying host processor. As a result of the Popek and Goldberg theorem [1], the execution of a privileged instruction (An instruction that is only allowed to be executed, when the processor is in a privileged mode.) has to trap when executed in non-priviledged processor mode. This is required, because the guest operating system is an application running on top of the host operating system and therefore does not run in the intended privilege mode of the processor. The trapping of such a privileged instruction is required, because only then, the host operating system is able to emulate the intended behavior for the guest operating system. Inside the x86-architecture this hw-trap mechanism was introduced not before the mid 2000s. Till then, the workaround was to recompile the guest operating system with a replacement of the non-trapping privileged instructions (see [2] for further details).

If the guest operating system is expecting another ISA, than the host processor provides, it is required to translate the guest ISA-instructions into host ISA-instructions. This techniques is called emulation. It can be enhanced by pre-translating and storing repeatedly used translations, which is called binary translation. However, both approaches result in a computational performance penalty for the guest system caused by the emulation/translation process.

This paper combines both, operating system virtualization and reconfigurable computing. This is achieved by avoiding the execution or even emulation of the guest operating system on the same processor as the host operating system, but providing a dedicated processor (and essential devices like timer and interrupt handler) for the guest operating system by the means of reconfigurable hardware. It is important for the reader to remember the difference: This paper uses reconfigurable hardware to support (guest) operating system virtualization and not virtualization of hw-accelerators inside an operating system. By applying this approach, it is expected, to reduce the computational performance costs, caused by the

SEEK DIGITAL LIBRARY

virtualization; to allow the bypassing of the Popek and Goldberg theorem; to provide more security within a virtualization system because guests and host are no longer executed on the same processor.

The paper focuses the following topics: in Section II related work is discussed, in Section III the architectural requirements to implement a hardware supported virtualization system are given in detail. Section IV gives a summarized presentation of the implemented proof of concept demonstrator. Section V evaluates the proof of concept demonstrator in relation to the expected benefits.

II. Related Work

The idea of this work is to combine virtualization and FPGAs. This idea is not new in general. The novelty of the idea presented here, is to fully support operating system virtualization with FPGAs.

According to [3] virtual machine technology allows single computers in the sense of real machines to host multiple virtual machines with individual operating systems.

The term ”virtualization” is used in the context of FPGAs since the mid 1990s. However, it is used to emphasize the abstraction of reconfigurable and exchangeable hardware by an operating systems. BORPH [4] allows to execute an application (process) on it's dedicated hardware and provides mechanisms for inter-process communication between sw- and hw-processes. Other systems like ReconOS [5], CapOS [6] or RTSM [7] enable to support threads with reconfigurable and exchangeable hardware and also provide communication mechanisms for the sw- and hw-threads.

However, these works focus on process virtual machines, whereas this paper focuses on system virtual machines. According to Smith and Nair [2] a process virtual machine allows to execute applications virtualized, whereas a system virtual machine allow to virtualize an entire operating system.

Xia et al. [8] extended the idea of the above mentioned operating systems for reconfigurable computing to allow a paravirtualized system virtualization in their Mini-Nova system. However, the paravirtualized operating systems are still executed on the static processors. In Mini-Nova, only reconfigurable accelerator modules can be provided to the guest operating systems, as opposed to the idea presented within this paper, where an entire machine is provided for each guest operating system.

The work presented within this paper does not focus on paravirtualization but hypervisors. Furthermore, an entire hardware machine is provided to the guest, not only some specialized and exchangeable accelerators.

III. ARCHITECTURAL CHALLENGES FOR FPGA BASED SYSTEM VMs

The overall architecure of a hardware supported virtualization system is shown in Figure 1 and extends a conventional computer architecture by a virtualization facility.

Hardware supported virtual machines are constructed by instantiating a required number of computers inside the virtualization facility. On these additional computers a guest operating system (or also a natively executed application) are executable. However, this paper focuses on the possibility to run a guest operating system on these additional computers.

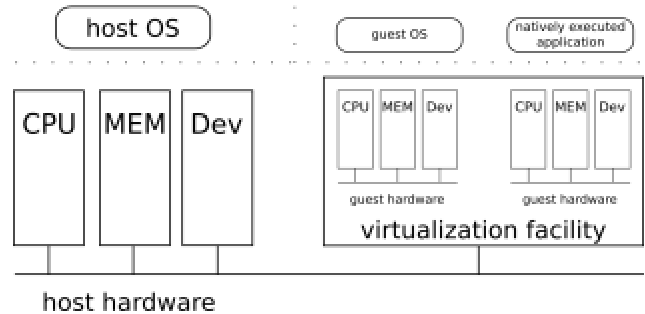

Figure 1. Overall architecure of a hw-supported virtualization system.

How has such a virtualization facility to be constructed? It is possible to build a fixed virtualization facility in silicon, but changing demands regarding the number of needed machines and how they are constituted (such as the number of processor cores, the amount of memory, the number and type of devices) are expected. This strongly implies to take advantage of reconfigurable logic (in form of FPGAs) to implement such a highly adaptable virtualization facility. Hence, the following two paradigms are defined:

- Reconfigurable logic is used on purpose and by principle to instantiate entire computers. Those instantiated systems can be seen as a guest system in terms of system virtual machine concepts. (instantiation paradigm)
- All dedicated resources of the overall system are managed by the host operating system (virtual machine manager), as done by conventional virtual machine managers. This is an essential requirement to see the overall system as virtualization system. (virtualization paradigm)

If the first paradigm is not fulfilled, a conventional virtual machine is implemented. If the second paradigm is not fulfilled, separate and independent systems are used; the only benefit would be the instantiation of guest systems by the host, without further control or supervision mechanisms.

The arising questions, problems and the corresponding solutions focusing on the overall architectural implications are discussed in the following sections. The outline of this discussion is based on the basic parts of a computer architecture: the processor(s), memory, and devices.

A. *Number of Guest Systems*

The number of guest systems is generally not limited. Limitations arise from the size of the available reconfigurable logic area and the area required for instantiating a guest machine.

In the following Sections, problems and ideas are discussed for a single guest environment. However, the

SEEK DIGITAL LIBRARY

proposed solutions would also apply to multi guest environments.

B. Processor Issues

Instantiating the guest processor in reconfigurable logic allows to have different central processing elements for host and guest operating systems and applications (summarized as *software*). This not only avoids conflicts between guest and host software but additionally introduces a physical barrier by design between them. On a conventional virtual machine system this barrier can only be achieved logically[2].

C. Memory Issues

Concerning the memory demands of the guest machines, some questions/problems arise:

- How much memory is required by the guest system? Can this amount of memory be provided by the reconfigurable area?
- How is the virtualization paradigm of the main idea (all resources are managed by the host operating system) enforced for the guest's memory?

1) *Providing enough Physical Memory for the Guest System:* It is not reasonable to use reconfigurable logic to instantiate large amounts (several hundred MByte or more) of memory, rather to use BlockRAM, also provided by today's FPGAs. Therefore, it is necessary to provide the possibility of accessing the overall system's main memory by the guest system. From a traditional computer architectures perspective, this can be seen as dedicated DMA channels for the guest systems.

2) *Virtualization of the Guests Virtual Memory:* Modern operating systems support virtualization of physical memory for giving an application the perception of having its own memory. This virtual memory concept has to be supported by both - guest and host operating system. The challenge for a virtualization system is that the virtual memory concept of a guest operating system has to be virtualized by the host system. This is one of the main additional problems that must be addressed when executing a guest operating system instead of an application inside the virtualization facility. In essence, the virtual addresses of the guest are translated by the guest's MMU to real addresses, which are subsequently translated to physical addresses by the host's MMU.

On conventional system virtual machines, the virtual-to-real and real-to-physical mapping for the guest VM is enforced by the same Memory Management Unit(s), as host OS and guest OS are executed on the same processor(s). In virtualization systems following the concept of this paper, host OS and guest OS run on different processors (as a consequence of the instantiation paradigm). Hence, other mechanisms to implement real-to-physical memory mapping for guest systems are necessary. To support this mapping, additional hardware, the *Guest Memory Management Unit (GMMU)* is introduced, as shown in Figure 2. Such a GMMU has to be available for each guest machine.

The usage of such a GMMU also provides the possibility to supervise the memory accesses of a guest system and especially allows to enforce the memory limitations of the guest system physically. Conventional virtual machine systems enforce this only logically [2].

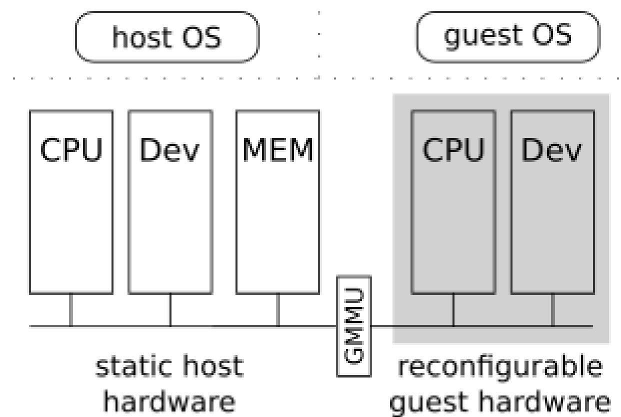

Figure 2. Introducing a Guest Memory Management Unit (GMMU).

In general, there are two possibilities to solve the problem of translating a guest's real addresses to the physically associated ones, as shown in Figure 3. Both solutions are adopted solutions of the real to physical mapping problem for multiprocessor virtualization [2].

a) Contiguous Chunk of Memory: This solution assumes the physical memory associated with the guest system to be contiguous. This assumption allows the GMMU to be implemented easily and straightforward. It just contains two registers, defining the first (Guest Memory Base Register, GMBR) and last address (Guest Memory Limit Register) of the physical memory area to be used by the guest system (see Figure 3)).

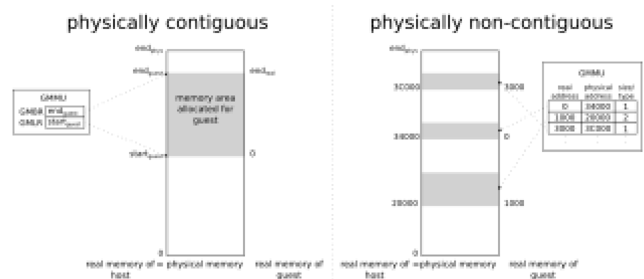

Figure 3. Options to solve the problem of translating a guest's real addresses to the physically associated addreses.

b) Page based Chunks of Memory (Non-Contiguous): The contiguous chunk of memory solution lacks some problems because of its simplicity. There is no dependency regarding the MMU of the host system. Hence, the memory area, associated with the guest system must not be swappable by the host system. Additionally, no differentiation regarding memory protection is possible, for example to mark some regions of the guest's memory as read only for the guest. This would be possible by introducing a GMMU that can be seen as an extension of the host's MMU (part of the host machine's CPU). Therefore, the GMMU has to implement the same memory management mechanisms as the host MMU. A GMMU based on this solution operates similar to an Input/Output Memory Management Unit (IOMMU)[9] used in processors today.

SEEK DIGITAL LIBRARY

D. Device Issues

In this section, device virtualization related questions are discussed. As a starting point, devices are classified into two categories:

- System vital, non sharable devices: These devices are essential for the functionality of the host or a guest system. Hence, sharing of such devices is usually senseless, as for example for an interrupt management controller or a timer. Therefore, they have to be duplicated for each guest system.
- Sharable devices: All other devices. For the main concept of this paper, only shareable devices are of interest concerning the question: How can device sharing between host and guest system be implemented?

1) Physical Interface Sharing

In general, there are three different possibilities to implement the physical sharing of the interface of a device:

- Explicit Hardware Switch: This solution (shown in Figure 4) provides the possibility to physically switch between the host and guest machine. The switching itself has to be controllable by the host operating system to ensure the virtualization paradigm.

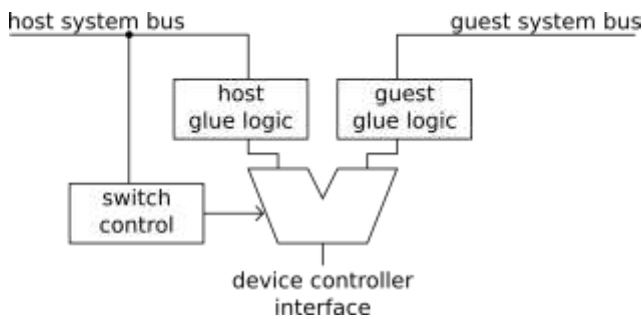

Figure 4. Explicit hardware switch solution.

- Switching by Reconfiguration: Instead of physically switching between guest and host machines, the connection between a guest and a device or the host and a device can by instantiated by means of reconfiguration, as shown in Figure 5. The reconfiguration of the switch is initiated and controlled by the host operating system.

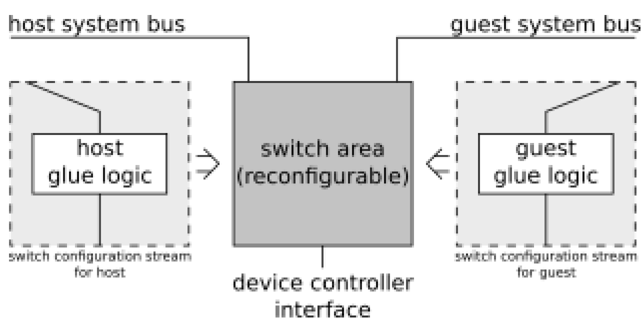

Figure 5. Switching by reconfiguration solution.

- Hardware Supported Mutual Exclusion: The previous solutions only provided mechanisms, where the guest and host machine can access a device exclusively (a device is physically connected to the system bus of exactly one machine at the same time). The hardware supported mutual exclusion solution allows the device to be attached to the guest's and host's system bus at the same time, as shown in Figure 6. Mutual exclusion has to be enforced on a hardware supported base, by implementing some kind of a hardware semaphore inside the switch. The host's supervising policies can be strengthened by enabling the host to suspend the mutex mechanisms.

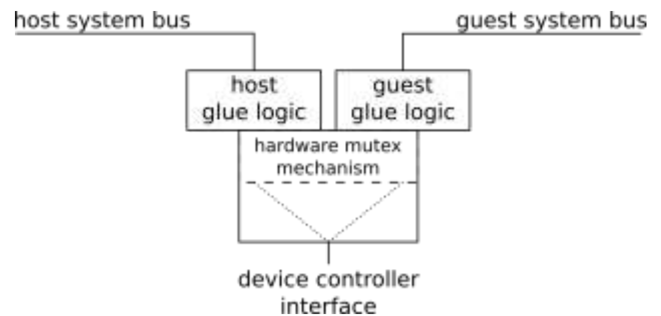

Figure 6. Hardware supported mutual exclusion solution.

2) Physically Supported Emulation

The solutions presented above try to share the physical interface of a device between a host and guests. For conventional system virtual machines, physical sharing is not required, as there is only one physical system bus. On a conventional system virtualization environment, the host operating system provides mechanisms to enable the guest system to use devices: A guest system, trying to access a device, traps to the host system, which itself supervises and handles the device access for the guest.

The mechanisms of virtualizing a physical device or emulating virtual ones, can also be applied to virtualization systems, covered by this paper's main concept, but require adaptations.

A guest operating system cannot cause a trap to be taken on the hosts processor because guest operating system(s) and host operating system are not executed on the same processor. For this reason additional hardware has to be used. This additional hardware provides a physical interface to the guest system, which allows the guest to signal a device request to the host system. This interface also has to be accessible by the host system to emulate the guests requests. It has to provide several addressable device registers and might include interrupt line(s) to avoid busy waiting. Figure 7 gives an example.

- For the guest system, this interface behaves just like a conventional one for the interaction with a device controller. The host system (or more precisely the host operating system) needs a driver for emulating a device controllers behavior for physically non existing devices or translate the guest's request to the physical device it is associated with.

SEEK DIGITAL LIBRARY

International Journal of Advances in Computer Science & Its Applications
Volume 6 : Issue 3 [ISSN 2250-3765]
Publication Date : 30 December, 2016

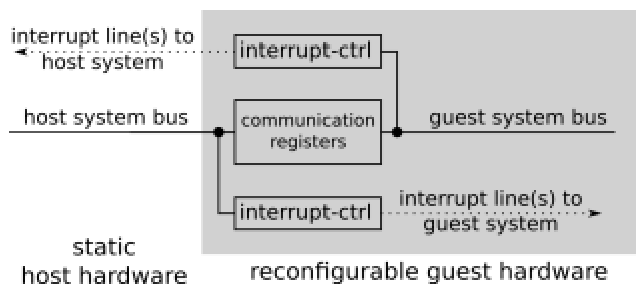

Figure 7. Physically supported device emulation.

Which of the above introduced possibilities to share or provide an interface to a device is the best? There can be no final decision, because this depends on the device itself and especially on the needed time granularity of sharing. Three timing aspects have to be taken into account here: the time needed to perform a reconfiguration; the time to register/unregister a device and load/unload a corresponding device driver and finally the time a device is used, before it is switched to another system.

IV. PROOF OF CONCEPT

It has already been stated that an extension of a conventional computer architecture by a virtualization facility is necessary to implement a hardware supported virtualization system (see Figure 1). At the beginning of Section III it was also proposed to implement the virtualization facility in an FPGA, resulting in a general architecture as given in Figure 8a).

The FPGA provides the requirements to host one or more hardware based guest systems. However, to investigate systems based on the main concept of this paper, it is necessary to have a testing framework that also offers full flexibility regarding the host systems hardware.

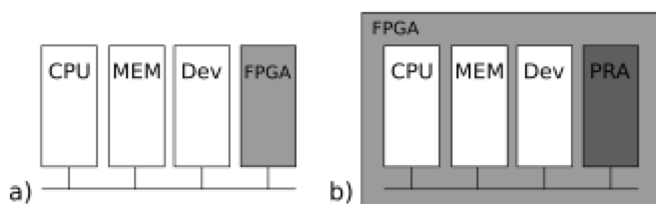

Figure 8. Architectures for implementing the concept. A) FPGA(s) as extension of a conventional system b) FPGA as prototyping environment for the overall virtualization system. (PRA-partial reconfigurable area)

Again, FPGAs are well suited for this purpose. The overall virtualization system hardware is configured onto an FPGA. The reconfigurable area is provided to the static part (host machine) of the overall system by means of a partially and dynamically reconfigurable area, as shown in Figure 8b).

The proof of concept demonstrator is based on the architecture shown in Figure 8b and is based on the *Partial Reconfigurable Heterogeneous System (PRHS) framework* [10]. The PRHS framework is a collection of VHDL modules. It allows to construct entire computer systems based on a self-implemented softcore (ARM-ISA, compatible to the ARM810 and achieves up to 120 MHz and includes optional Caches and Memory Management Units). The framework also includes an adapted Linux kernel (version 4.1, to be used with the self-implemented softcore) and allowing for *busybox* or *Embedian* as Linux distributions.

A. *Demonstrator Architecture*

A simplified overall Architecture for the proof of concept demonstrator is given in Figure 9. Host and guest system are built around the softcore included in the PRHS framework. The host System consists of the processor, several I/O devices, a *reconfiguration interface* (Reconf-IF) and a *memory supervision* element. The latter one is used to implement the *Contiguous Chunk of Memory* solution for memory sharing between the host and the guest system. The partial reconfiguration area, which can hold exactly one guest machine is controllable by the *reconfiguration interface* (Reconf-IF). This includes an *Internal Configuration Access Port (ICAP)* to perform the reconfiguration itself. Furthermore, the Reconf-IF can set the partial reconfiguration area into one of three states:

- *off:* The global reset line of the partial reconfiguration area is asserted and all outgoing lines (from reconfigurable area to host system) are tied to low. The first mechanism allow to reset the guest system, the second one is required when a reconfiguration takes place.
- *on, disabled:* The guest system contains a dedicated *emulation interface* (Emul-IF) and *communication interface* (Comm-IF). The Emul-IF implements the *Physically Supported Emulation* solution for emulating the guests hard disk. The Comm-IF enables to interact with the guest operating systems console from the host side. Figure 9 shows that both devices are connected to the device buses of both, the guest and the static system. However, for proper device driver handling inside the guest system it is necessary to bring up both devices inside the host operating system, before the guest operating system is started. Hence, the *on, disabled* state deasserts the global reset line of the partial reconfiguration area but keeps the clock enable lines of the guest machine disabled and the asserts the clock enable line of the host system attached device side.
- *on, enabled:* Compared to the *on, disabled* state, this state allows the guest machine to execute, eventually.

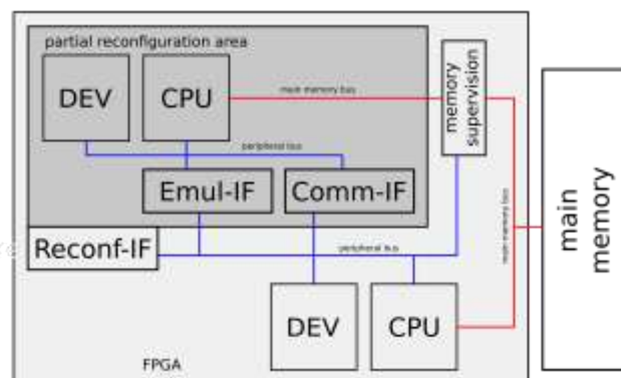

Figure 9. Overall proof of concept demonstrator architecutre (simplified).

SEEK DIGITAL LIBRARY

B. *Host OS Extension*

The virtual machine manager is implemented as an application of the host operating system. Therefore, device drivers for handling the Reconf-IF, Emul-IF, Comm-IF and memory supervision devices have been implemented. These device drivers are usable as loadable kernel modules. The virtual machine manger application has to perform the following tasks, to start a guest machine:

- Set the Reconf-IF into *off* state.
- Load the partial bitstream into the partial reconfiguration area.
- Set the Reconf-IF into *on, disabled* state.
- Register Comm-IF device. Load the corresponding device driver if necessary.
- Register Emul-IF device. Load the corresponding device driver if necessary. Afterwards, assign the hard disk file associated with the guest system. This hard disk file also contains the guest systems stage-two bootloader and the intended OS kernel.
- Set up the memory supervision device to provide the associated amount of memory. The entire associated memory of the guest machine is zeroed.
- Finally, set Reconf-IF into *on, enabled* state to allow the guest machine to execute.

The Reconf-IF states *on, disabled* and *on, enabled* can be used to pause and resume the guest machine. This provides the possibility to the host system to inspect the guests memory contents. Care must be taken inside the *memory supervision* device due to the possibility of pending memory request, when pausing the guest system.

Swapping between several guest machines on one partial reconfiguration area is not implemented. This would require a stop, save, restore and resume mechanism. The save and restore steps would require the extraction/setting of the entire hardware state (contents of flip-flops) inside the reconfigurable area. This feature is currently not implemented.

C. *Guest OS Adaptation*

One of the advantages of true system virtualization is the possibility to execute the guest operating system without any modifications. This also holds for the proof of concept demonstrator. Of course, the *emulation interface* (Emul-IF) needs an appropriate driver to use the intended functionality. However, from the guest operating system perspective, this is simply another block-device driver for interacting with a hard disk. The driver handling of the *communication interface* (Comm-IF) is also just another TTY-driver.

V. EVALUATION

After having presented the proof of concept demonstrator in detail, information on reconfigurable resource usage for the demonstrator is given in Table I. The resource footprint is given for the evaluation boards, the proof of concept demonstrator is available for.

In the following, some benchmarks are presented which have been taken for the proof of concept demonstrator on a VC707 evaluation board.

TABLE I. RECONFIGURABLE RESOURCE FOOTPRINT FOR PROOF OF CONCEPT DEMONSTRATOR

board name	XUPv5	ML605	VC707	Nexys4DDR
FPGA	xc5vlx110t	xc6vlx240t	xc7vx485T	xc7a100t
slices host system	6.352 (37%)	9.429 (25%)	10.078 (13%)	5.132 (32%)
slices guest system	3.054 (17%)	3.799 (10%)	3.522 (5%)	2843 (18%)
slices host + guest	9.406 (54%)	13.228 (35%)	13.600 (18%)	7.975 (50%)
f_{CLK} (host + guest)	80MHz	50MHz	80MHz	50MHz
Hard disk type	CF-Card	CF-Card	SD-Card	SD-Card
overall memory	256MB	512MB	1GB	128MB
guest memory	32MB	32MB	32MB	32MB

Values in brackets give the slice count in relation to the overall slice count of the FPGA

A. *Processor Performance*

Performance investigations regarding the processor have been done using the *Dhrystone* benchmark[11]. (Dhrystone measurement is done in Dhrystone million instructions per second (DMIPS). Version 2.1 of the benchmark has been used, compiler optimizations were disabled.) Dhrystone is sufficient here, because two systems with nearly identical architectures are compared. Three different values were measured. P_{static} is the result for the host part of the system with reconfigurable area switched *off*. P_{host} is the performance of the host part with reconfigurable area switched *on, enabled* and P_{guest} is the benchmark result taken within the guest system. Relative values are given here, because the discussion has to focus on performance differences between the host and the guest system. The peak performance (100 %) was achieved with the host system running and guest (reconfigurable area) switched *off*. The results are:

P_{static} = 100%

P_{host} = 95% related to P_{static}

P_{guest} = 95% related to P_{static}

Comparing P_{host} with P_{guest} shows, that the guest systems and the host systems computational performances are identical. So there is no performance loss caused by the virtualization mechanism. However, a performance loss compared to P_{static} is measured. The reason for this is the shared memory controller, so when both machines try to access main memory simultaneously, one has to wait. This waiting is implemented fair: when one time the guest waits for the host, the other time it is vice versa.

B. *Memory Access Time*

Memory access time is measured on the basis of the *ramspeed* benchmark [12]. This benchmark measures the read and write cache/memory access performance in MegaBytes per second based on an increasing blocksize for reads and writes. The results are shown in Figure 10. R(x) is the read performance, W(y) is the write performance.

SEEK DIGITAL LIBRARY

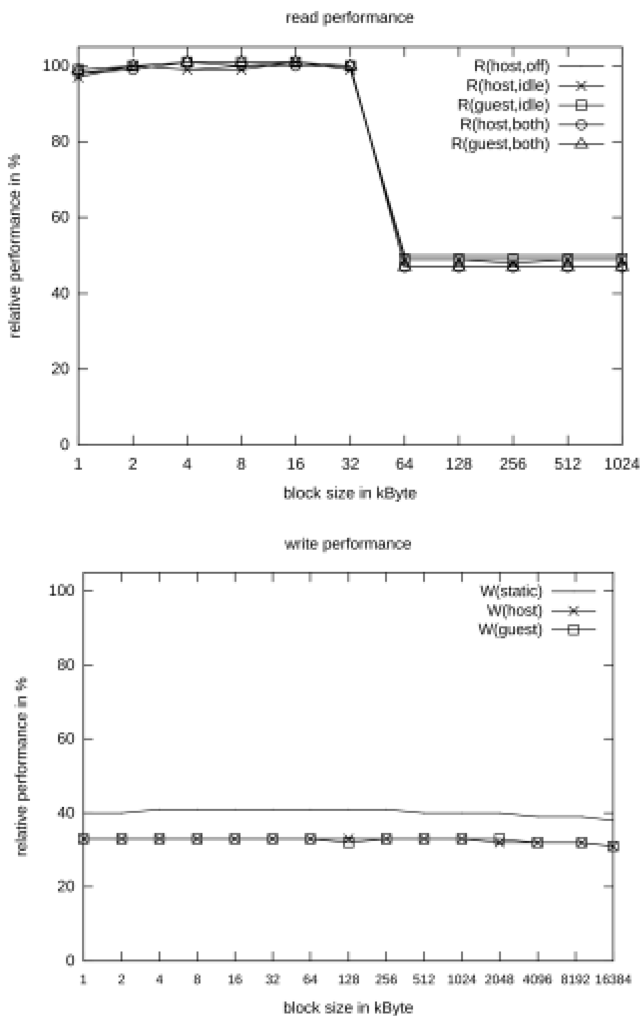

Figure 10. ramspeed benchmark results

Performance is given as a relative value again. The reference value is the peak value of all measured results. This value is used as a reference for read and write performance calculation. For the interpretation of the results, it necessary to mention, that a 16 kB direct mapped cache with write-through strategy is used inside the processors. The cache size can be easily figured out in the read performance chart. The write through strategy is the reason for the constant lines in the write performance chart. The general conclusion for the cache/memory access performance measurement is: Both, the host and the guest system perform identical. Reduced memory access performance for both systems results from the shared memory controller. The amount of performance reduction is the same for both.

C. *Hard Disk Access Times*

The hard disk device of the guest system is implemented with the emulation interface (Emul-IF). Therefore, the hard disk of the guest system is a file on the hard disk of the static systems. The mapping between the Emul-IF and the HD-file of the guest is implemented in the host operating system in a similar way, the *loop*-block device in Linux is implementing this issue. On the guest system side, the Emul-IF is used by a dedicated block device driver. Figure 11 shows benchmark results for hard disk accesses based on the *bonnie++* benchmark [13]. Benchmark results have been measured on the VC707, where the host hard disk is a SD-Card used in SPI-Mode and the guest file systems hard disk image residing on this SD-Card.

Static presents the test case, where the guest system is switched *off* and the host machine is accessing its "real" hard disk (SD-Card). The case *host* is also measured on the host system, but with a running guest system. The *guest* test case measures the emulated hard disk access of the guest system. A hard disk access of the guest system is issued to the host via the Emul-IF. The host itself has to read/write the required data from/to the SD-Card and signal completion to the guest operating system. The guest systems hard disk is therefore simply a file on the SD-Card.

Benchmark results for *static* and *host* are identical (as expected) as both are measured on the host system. The *guest* results are in some scenarios slower (block write), sometimes identical (block rewrite) and sometimes faster (block reads and random seeks). As the guest machines hard disk is emulated by the host system, it was expected to always have a slower benchmark result compared with the host machines accesses. The reason for the unexpected better results of the guest system is the filesystem cache of the host machine. The host is caching the data of the guest systems hard disk file. This caching mechanism of the host system cannot be avoided by the benchmark tool, running on the guest. Therefore, reading from the emulated hard disk inside the guest system (and random seek is another form of reading) seems to be faster than reading on the host system. So, the general summary for hard disk access time is that the emulation of a hard disk slows down the hard disk access time of the guest system, but for filesystem cache reasons, it can be faster in some cases.

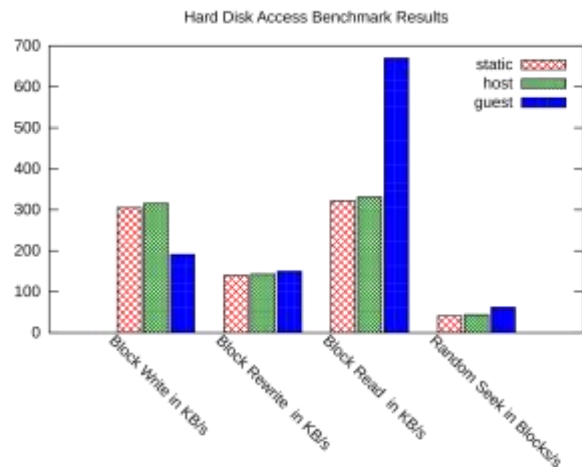

Figure 11. bonnie++ benchmark results

D. *Discussion of Results*

The proof of concept demonstrator instantiated a guest machine, that is identical to the host. This includes the underlying core architecture (processor core) and the guest operating system. The benchmark results show, that there is no performance loss due to the virtualized execution for the

SEEK DIGITAL LIBRARY

computational and memory performance. However, the used processor is a softcore and therefore will always be an order of magnitude slower than a modern hardwired core.

Nevertheless, the system virtualization approach presented in this paper can be useful, when the instruction set architecture (ISA) of the processor differs between guest and host. For conventional system virtualization mechanisms there is only one solution: emulation or binary translation of the guest ISA with the host ISA resulting in a significant performance loss. The approach within this paper allows to replace emulation by instantiation of a processor with the an appropriate ISA. This has not been shown by the proof of concept demonstrator, but is obvious by design.

Furthermore, a Type-1 Hypervisor is only possible, when the host processor fulfills the Popek and Goldberg theorem (see introduction). With the vitualization approach presented in this paper, the host processor does not need to fulfill this theorem, as the proof of concept demonstrator shows. The arm8-ISA implemented within the used processor does not fulfill the Popek and Goldberg theorem[14].

Conventional virtual machines provide security by separating different guest virtual machines logically by software. Virtualization systems, following the main idea of this paper, enforce this separation even stronger as there is physically separated hardware executing the guest systems software. The introduction of the GMMU (see Section III-C) adds an additional hardware mechanism for limiting a guest's possibility to access the host memory. By using reconfigurable logic to instantiate a guest systems hardware, it is possible to add special devices supervising the guests hardware. It is possible to implement those supervision devices in a way, that they are only manageable by the host system.

VI. CONCLUSION

A new type of operating system virtualization by the usage of configurable logic FPGAs is introduced in this paper. Additionally, a proof of concept demonstrator, based on the PRHS Framework is presented and evaluated in this paper. The advantages of the proposed system virtual machine architecture are:

- The guest system does not suffer from computational performance losses caused by the applied virtualization techniques. However, care must be taken on the performance penalties of device emulation.
- The entire system can be heterogeneous. The guest systems can take profit from different instructions sets without the need of employing strong emulation techniques on the host.
- Type-1 hypervisors are possible on processors, that do not fulfill the Popek and Goldberg theorem.

Further investigations on the following topics are required/ planned:

Resource sharing: The proof of concept demonstrator only implements the *physically supported emulation* mechanism to provide device sharing among host and guest. Further possibilities are discussed in section III-D1 and need to be evaluated.

Suspending mechanisms: The current implementation only allows to start and stop one guest machine. There is neither a possibility to suspend a guest machine nor to resume a previously preempted machine that was brought back into a reconfigurable area. Furthermore an extension of the architecture to host several guests, including processors with different ISAs, side by side is planned for the future.

Security: The proposed architecture for supporting system virtual machines with dedicated hardware allows to strengthen the isolation of guests and host by hardware and not only by software. The benefits and drawbacks in the areas of IT-Security and IT-Forensics should be evaluated in more detail in the future.

References

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," Commun. ACM, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[2] J. Smith and R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design). San Francisco, CA, USA: Morgan Kaufmann Publisher Inc., 2005.

[3] A. S. Tanenbaum and H. Bos, Modern operating systems. Prentice Hall Press, 2014.

[4] H. So and B. University of California, BORPH: An Operating System for FPGA-based Reconfigurable Computers. University of California, Berkeley, 2007.

[5] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "Reconos: An operating system approach for reconfigurable computing," Micro, IEEE, vol. 34, no. 1, pp. 60–71, Jan 2014.

[6] D. Gohringer, M. Hubner, E. Zeutebouo, and J. Becker, "Cap-os: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures," in Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, 2010, pp. 1–8.

[7] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, "Hardware task scheduling for partially reconfigurable fpgas," in Applied Reconfigurable Computing, ser. Lecture Notes in Computer Science, K. Sano, D. Soudris, M. Huebner, and P. C. Diniz, Eds.Springer International Publishing, 2015, vol. 9040, pp. 487–498.

[8] T. Xia, J. C. Prevotet, and F. Nouvel, "Mini-nova: A lightweight armbased virtualization microkernel supporting dynamic partial reconfiguration," in Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International, May 2015, pp. 71–80.

[9] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. V. Doorn, A. Mallick, and E. Wahlig, "Utilizing iommus for virtualization in linux and xen," in In Proceedings of the Linux Symposium, 2006.

[10] M. Eckert, I. Podebrad, and B. Klauer, "Hardware based security enhanced direct memory access," in Communications and Multimedia Security, ser. Lecture Notes in Computer Science, B. De Decker, J. Dittmann, C. Kraetzer, and C. Vielhauer, Eds. Springer Berlin Heidelberg, 2013, vol. 8099, pp. 145–151.

[11] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," Commun. ACM, vol. 27, no. 10, pp. 1013–1030, Oct. 1984

[12] R. M. Hollander and P. V. Bolotoff, "RAMspeed, a cache and memory benchmarking tool," http://alasir.com/software/ramspeed/, 2009.

[13] R. Coker, "Bonnie++," http://http://www.coker.com.au/bonnie++/, 2001.

[14] Advanced RISC Machines Ltd (ARM), ARM810 Data Sheet, 1996.

SEEK DIGITAL LIBRARY