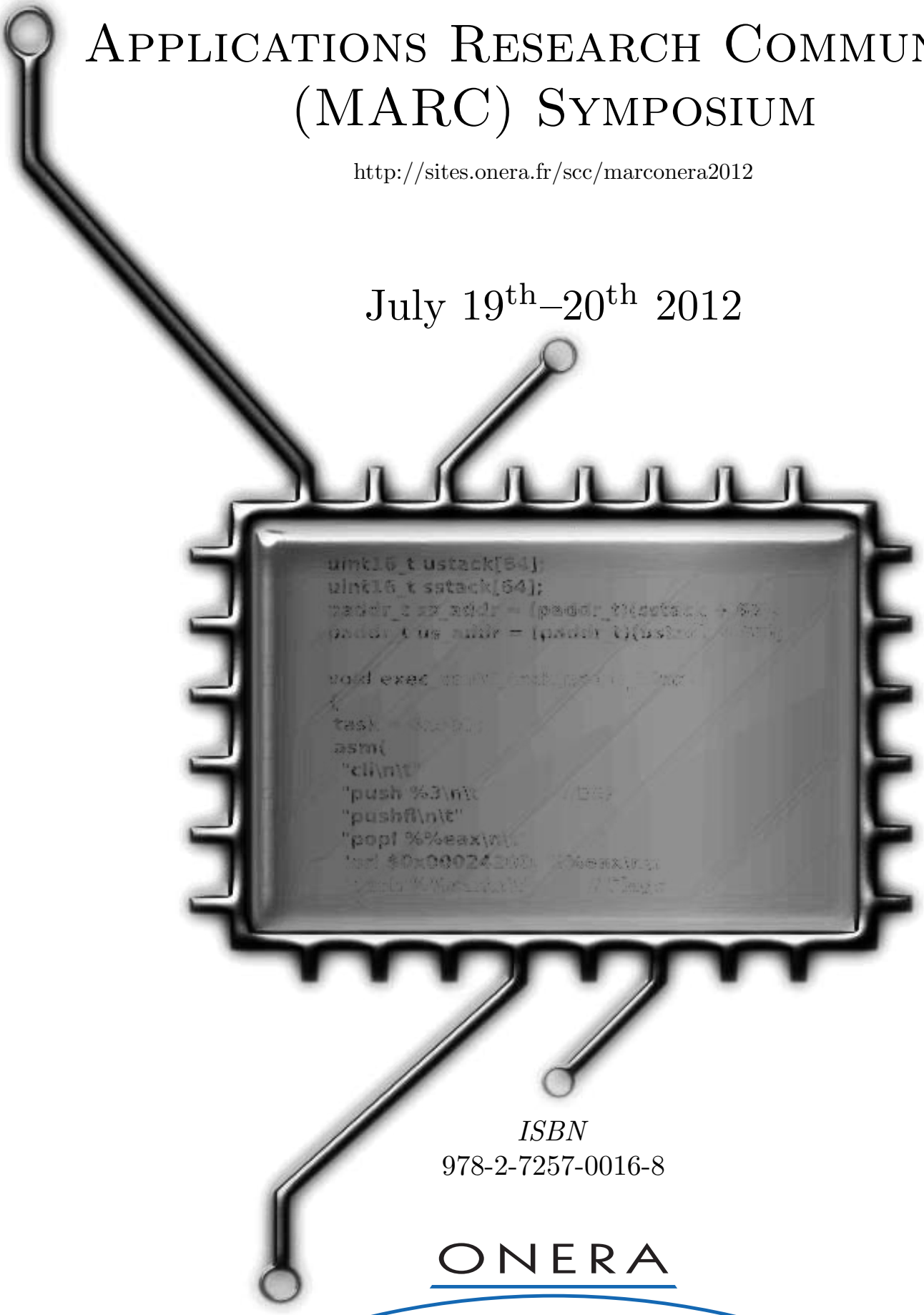
Efficient Implementation of the bare-metal Hypervisor MetalSVM for the SCC

Pablo Reble, Jacek Galowicz, Stefan Lankes, Thomas Bemmerl

Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

http://sites.onera.fr/scc/marconera2012

July 19th–20th 2012



ISBN
978-2-7257-0016-8

ONERA
THE FRENCH AEROSPACE LAB

 ISBN: 978-2-7257-0016-8

Efficient Implementation of the bare-metal Hypervisor MetalSVM for the SCC

Pablo Reble, Jacek Galowicz, Stefan Lankes, Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
{reble,galowicz,lankes,bemmerl}@lfbs.rwth-aachen.de

***Abstract*—The focus of this paper is the efficient implementation of our compact operating system kernel as a bare-metal hypervisor for the SCC. We describe source, functionality, and the operation of our kernel, as well as the interaction with the already published communication layer. Furthermore we give a detailed insight into the boot procedure of the SCC from reset to the starting point of our light-weight operating system kernel. This procedure is performed by a bare-metal framework, which is part of the MetalSVM project. Programmers can use our framework as a springboard for bare-metal programming on the SCC, which goes along with the first release of MetalSVM. Finally, we evaluate the performance of a paravirtualized Linux guest on the SCC hardware and present results of context switch latencies for Linux and MetalSVM hosts.**

I. INTRODUCTION

The Single-chip Cloud Computer (SCC) experimental processor [1] is a *concept vehicle* created by Intel Labs as a platform for many-core software research, which consists of 48 cores arranged in a 6×4 on-die mesh of tiles with two cores per tile. The intended programming approach for this cluster-on-chip platform is based on message passing [2].

For the parallelization of data-intensive algorithms, especially with irregular access pattern a shared memory programming model like *OpenMP* which is based on memory coherence offers an attractive and efficient alternative. If future many core processor architectures have to waive the memory coherency implementation in hardware, *MetalSVM* can enable shared memory programming on those architectures using virtualization.

One logical, but parallel and cache coherent virtual machine runs on top of a virtualization layer. With a Shared Virtual Memory (SVM) system this implements a classic approach for the realization of memory coherence in software in a bare-metal hypervisor. The virtualized Linux instance, called guest, will have the impression of being executed on a symmetric multiprocessor system. As a result, standard shared memory parallelized applications can run on future many-core platforms. Since the shared memory paradigm shows advantages in many scenarios, we are convinced that it is valuable to transparently provide memory coherence even on an architecture without according hardware support.

This paper is structured as follows: In Section II, we motivate the realization of *MetalSVM*[1] and summarize related work of our project. Afterwards, we present in Section III the structure and implementation details of the first version of *MetalSVM*. We describe the Boot process of the hypervisor kernel on the SCC platform in Section IV. Additionally, we compare context switch overhead and the hypervisor implementation performance between Linux and *MetalSVM* in Section V. In Section VI, we explain the benchmarks used for the evaluation of our kernel and present the respective performance results. The final Section VII summarizes this paper and gives an outlook to our next research goals.

[1] http://www.metalsvm.org

II. MOTIVATION AND RELATED WORK

Initially by forking *eduOS*, we started the further development of *MetalSVM*. *eduOS* is a very minimalistic operating system used for educational purposes at the RWTH Aachen University. It is inspired by *Unix* but does not aim to be fully POSIX compliant as, for instance, the Linux kernel or the MINIX kernel, which are also used for operating system courses and research [3].

In fact, the simplicity of *eduOS* leads to an easy customizability and tasks running in kernel space are executed near bare-metal. As a lightweight and small monolithic kernel, it provides adequate functionality for running user space programs. Figure 1 shows the basic kernel structures of *eduOS*.

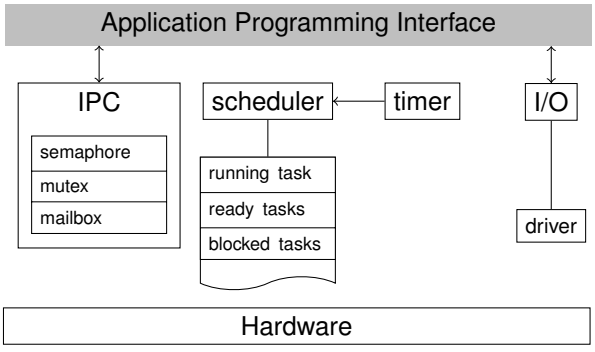


Fig. 1: Kernel structure of *eduOS*

MetalSVM, the further development of *eduOS*, represents a highly optimized codebase for running applications near bare-metal on the Intel SCC. Programmers can use our framework as a springboard for bare-metal programming on the SCC. In [4], we presented a first prototype, and in [5] further improvements of an SVM system, based on our framework. Here,

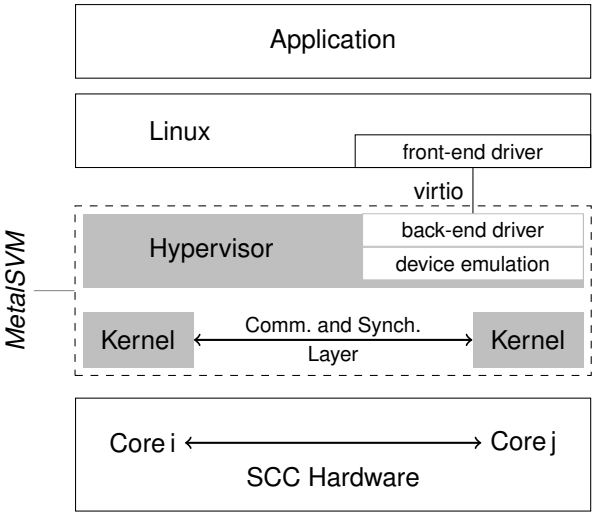


Fig. 2: Basic Concept of MetalSVM [4]

a shared memory application uses special SVM functions explicitly for shared memory allocation. A transparent use of the SVM layer by unchanged software will be enabled by a virtualization layer on top of the functionality of the *MetalSVM* kernel (see Figure 2).

From the application programmer's view, Linux user space applications have limited control over the preemption time, which is affected by context switching and interrupt handling. Consequently, this can be a good reason to run applications bare-metal to avoid this kind of overhead. However, one may be not interested or be able to take care of the rest of the necessary low-level work, which is the common reason for using an operating system. Since *MetalSVM* is configurable, the possibility exists to switch off infrastructure, for instance the hypervisor or the communication layer, which makes our framework comparable to bare-metal frameworks presented at the Intel Communities page [6], [7].

In [8], we evaluated the synchronization and communication hardware support of the SCC for inter kernel usage. For the integration of *iRCCE* into *MetalSVM*, this included an extension in the form of a mailbox system in combination with optimized synchronization support. The result is fast synchronous and asynchronous communication between user and kernel tasks of *MetalSVM* [9].

Besides *MetalSVM*, several projects handle the integration of an SVM system into virtual machines, for an easy application of common operating systems and development environments without changes. An example for such a hypervisor-based SVM system is *vNUMA* [10] that has been implemented on the Intel Itanium processor architecture. In [11] one founder of *vNUMA* proposed to extend this concept for Many-Core Chips. For x86-based compute clusters, the so-called *vSMP* architecture developed by ScaleMP[2] allows for cluster-wide cache-coherent memory sharing. This architecture implements a virtualization layer underneath the OS that handles distributed memory accesses via InfiniBand-based communication. In some respects, these approaches are similar to our hypervisor approach. Both implement the SVM system in an additional virtualization layer between the hardware and the operating system.

The main difference between these approaches is that *vSMP* and *vNUMA* explicitly use message-passing between the cluster nodes to transfer the content of the page frames, whereas our SVM system can cope with direct access to these page frames. In fact, we want to exploit the SVM system with SCC's distinguishing capabilities of transparent read/write access to the global off-die shared memory. This feature will help to overcome a drawback of other hypervisor-based approaches regarding fine granular operations. A recent evaluation [12] of ScaleMP's *vSMP* with synthetic kernel benchmarks as well as with real-world applications has shown that *vSMP* architecture can stand the test if its distinct NUMA characteristic is taken into account. Moreover, this evaluation reveals that fine granular operations such as synchronization are the big drawback of this kind of architectures. Our aim is to avoid this shortcoming by using the distinguished capabilities of transparent remote read/write memory on the SCC.

RockyVisor [13] is the name of another project for the realization of a hypervisor based symmetric multi-processing support for the SCC. In contrast to *MetalSVM*, this project targets the integration of its hypervisor into Linux and not on the base of a minimalistic kernel. Therefore, on the top of all Linux instances runs a virtualized Linux, which assumes that the SCC is an SMP system. From our point of view, such a *Linux on Linux* approach implies unneeded overhead.

III. KERNEL FEATURES

The intended usage for an SVM management system influences the hypervisor kernel. In this section, we detail the implementation of this monolithic kernel including interrupt handling, device drivers, file system, and hypervisor. Additionally, we give reasons for specific design decisions by concrete applications.

The focus in this paper is the kernel implementation for the SCC. However, we compare this implementation to different hardware architectures supported by *MetalSVM*, whose concept is divided in a hardware dependent and independent part.

A. Hypervisor

The fact that a guest kernel is aware that it runs as a guest and uses hypercalls to do privileged operations is called paravirtualization [14]. Using an existing hypervisor solution from the Linux kernel has been the first choice for the integration into *MetalSVM* [15]. This way we can avoid changes on the Linux kernel code, since interaction between host and guest is based on a de facto standard virtualization interface. *lguest* is an appropriate match in this context, because its about 5000 lines of code keep it quite simple. Despite its small

[2]http://www.scalemp.com

size it provides all required features for the realization of the *MetalSVM* project [16].

For development and testing purposes, we use *QEMU*[3], which is a generic and open source machine emulator and virtualizer. To simplify our tests of standard kernel components, we integrated a driver for the Realtek RTL8139 network chip, which is also supported by *QEMU* as an emulated device.

B. Device Drivers

Communication between the SCC cores running *MetalSVM* is not limited to the iRCCE library and its mailbox extension. With the integration of *lwIP*, a light-weight TCP/IP library, the flexibility is increased [17]. Consequently, BSD sockets are made available to user space applications to establish communication between the SCC cores and the MCPC. In [4], we demonstrated the convincing performance of the resulting network layer.

The network capabilities besides other devices of *MetalSVM* will be forwarded to the guest operating system through the hypervisor via *virtio*. Virtio is Rusty Russell's draft to create an efficient and well-maintained framework for IO-virtualization of virtual devices commonly used by different hypervisors [18]. In our scenario, for instance the network capabilities of *MetalSVM* are used as a backend by just forwarding the requests of the Linux guest operating system to the hypervisor.

C. Interrupt Management

The SCC platform includes 48 P54C cores. As a second generation of *Pentium* cores, the P54C is the first processor which is based on an on-chip local *Advanced Programmable Interrupt Controller* (APIC). This local APIC is used to program the local timer interrupt, which can be used to trigger the scheduler periodically. *MetalSVM* uses a simple priority-based round-robin scheduler, described in detail in Section V.

Beside the timer interrupt, the local APIC possesses two programmable local interrupts (*LINT0* and *LINT1*). Interrupts achieve an important role, because the SCC does not use the traditional way to integrate I/O devices (*IO-APIC*) or to send inter-processor interrupts (*IPIs*). Therefore, a core configuration register exists for each core of the SCC, which is mapped to the address space of all cores. A special bit in these registers triggers a *LINT0* or a *LINT1*. As a result, core x is enabled by the memory-mapped configuration registers to trigger an interrupt on core y. However, with this mechanism the receiving core is now able to determine the origin of the interrupt.

The update of Intel *sccKit* to version 1.4.0 includes a *Global Interrupt Controller* (GIC), which provides a more flexible way to handle interrupts [19]. If an interrupt is triggered by the GIC, the receiver is able to determine the origin of this interrupt. *MetalSVM* uses the GIC especially for inter-core communication via iRCCE or our mailbox system [5]. Here, the information about the origin of an interrupt increases the scalability.

[3]http://www.qemu.org/

D. File system

Since the SCC provides no non-volatile storage, a file system is physically limited in use. Nevertheless, *MetalSVM* has an elementary *inode* file system with an initial population loaded from a ramdisk file. This file system can be manipulated at runtime.

The integration of *newlib*[4], which is a C library intended for use on embedded systems, extends the usage of *MetalSVM*. Regarding the mode to run user-space applications on *MetalSVM* arises the possibility to access custom character devices by the provided `/dev` directory. These can be implemented very comfortably using a well defined interface.

IV. BOOT ON THE SCC

MetalSVM is *Multiboot*[5] compliant. This means that the project framework creates an ELF kernel file and an initial ramdisk image file. A boot loader like GRUB can easily use these files to boot *MetalSVM* on commodity x86 hardware.

Because the available SCC hardware is a research prototype, the booting process differs from commodity hardware. Differences to commodity hardware are the absence of BIOS support and the lack of stand-alone memory initialization of this experimental platform. The only possibility to bootstrap the SCC cores is preloading their memory content into a bootable state. Thus, the general system initialization is realized by a standard PC (MCPC) with direct access to the memory of the SCC and its configuration registers.

In the following, we describe the function of our framework to bring the SCC Platform into a *Multiboot* compliant state. As a result, an entry point for our 32 bit minimalist *Multiboot* kernel is created. Additionally, we describe the interaction with the common *sccKit* tools to boot up the SCC platform with *MetalSVM*.

Initially, the boot procedure starts by pulling the reset pins of the SCC cores. Next, its Lookup Tables (LUT) are initialized and the memory is set into a bootable state for each core. After a reset pin release the instruction pointer of each core holds the hardwired address `0xfffffff0`. As the SCC does not provide any form of boot loader, our framework provides minimal assembler code for this purpose, which needs to be located at this position. Starting the operation in real mode, this code initializes the stack pointer, installs a rudimentary GDT, switches the processor to protected mode and subsequently to 32 bit mode. As a last step, this setup procedure jumps to the alignment value of the *MetalSVM* kernel, address `0x100000`.

The compiler has to support the pentium architecture for the generation of ELF format output of our minimalist kernel for the SCC. ELF, as the standard binary file format for Unix systems, is currently not supported by the *sccKit* tools. Therefore, the utility `objcopy` is used to generate a directly loadable, raw binary kernel file by discarding all symbols and relocation information. The previously described startup

[4]http://sourceware.org/newlib/

[5]http://www.gnu.org/software/grub/manual/multiboot/

routine, from real to protected mode, a data struct, containing information which is generally provided by the bootloader, and the kernel itself are composed to a single image by `sccMerge`.

Next, `sccMerge` creates rules for the configuration of the LUTs and one object file per memory controller of the SCC platform. Subsequently, `sccBoot` loads the generated object files into the off-die memory and finally `sccReset` is used to release the reset pins of the SCC cores.

V. SCHEDULING

Requirements to a scheduler of the presented hypervisor are lower compared to schedulers of popular modern operating systems. Specifically, the intended use for the scheduler is to handle a few tasks, such as the guest kernel, daemon and monitoring tasks. Hence, a simple but fast algorithm is applied to manage tasks. The scheduler keeps an array with as many items as priority steps exist. Per priority there is one linked list of tasks waiting for execution. Between two timeslices the scheduler appends the previous task to the end of its priority list and selects the head of the current processed priority level list for execution.

The small set of implemented priorities in *MetalSVM* provides the possibility to apply optimization. One optimization is already implemented in the networking layer. Network packet traffic is handled in a special kernel task whose priority can be changed. This way it is possible to balance between high network throughput and overall system latency.

Version 0.1 of *MetalSVM* supports 32 different priority levels. This small number allows *MetalSVM* to create a bitmap of used priority queues in one 32 bit integer. Consequently, with one assembler instruction (`msb`) is it possible to determine the highest used priority queue, which promises an extremely low overhead. Before leaving any interrupt handler, the handler checks, if a task with higher priority is able to run and calls the scheduler if required. In our scenario, a reduction of the latency of the network stack can be achieved, by holding the network thread at a higher priority than the computation tasks.

A. Hardware Context Switch

Early versions of *MetalSVM* used the x86 hardware task switching by default for a context switch. Here, a context switch is performed by a `JMP` to a *TSS* descriptor, which has the advantage of a very simple application. Therefore, the *TSS* (Task State Segment), which stores the state of a task, except the FPU state, is restored.

A disadvantage of this method is the lack of selection which registers are saved and restored [20]. Furthermore, the number of *TSS* entries in the segment table is limited to 8192 [21]. Due to portability reasons most modern operating system implementations use software task switching.

However, a basic component of the SCC is the classic P54C core, which could result in an increased scheduling performance of hardware context switching. Besides a benchmark of the hypervisor layer, we evaluate this assumption in the next section.

VI. BENCHMARKS

Benchmark results of different subsystems of *MetalSVM*, excluding the kernel, have been already published. An evaluation of the synchronization support, including different spin lock and barrier implementations is presented in [8]. In [4], the network layer and a first prototype of the SVM layer are evaluated. Further optimizations of the SVM layer and the mailbox extension of iRCCE are presented in [5].

In this section of this paper, we analyze advantages of a bare-metal implementation of *MetalSVM*. We compare the context switch overhead of *sccLinux 2.6.38* to *MetalSVM* 0.1 on the SCC platform. Additionally, we compare the *lguest* implementation of *MetalSVM* to the implementation of the Linux kernel 2.6.32 and 2.6.38.3. For a comparison of the results, the benchmark application is the single process running on *sccLinux*.

For the benchmark in this section, we obtained measurements by running a single instance of the selected host operating system on a single core of the SCC platform[6]. Because, *sccLinux* in a version 2.6.32 is currently not available in a configuration with *lguest* support, we used an Intel Celeron 550 test system with a frequency of 2 GHz, to benchmark the context switch latencies.

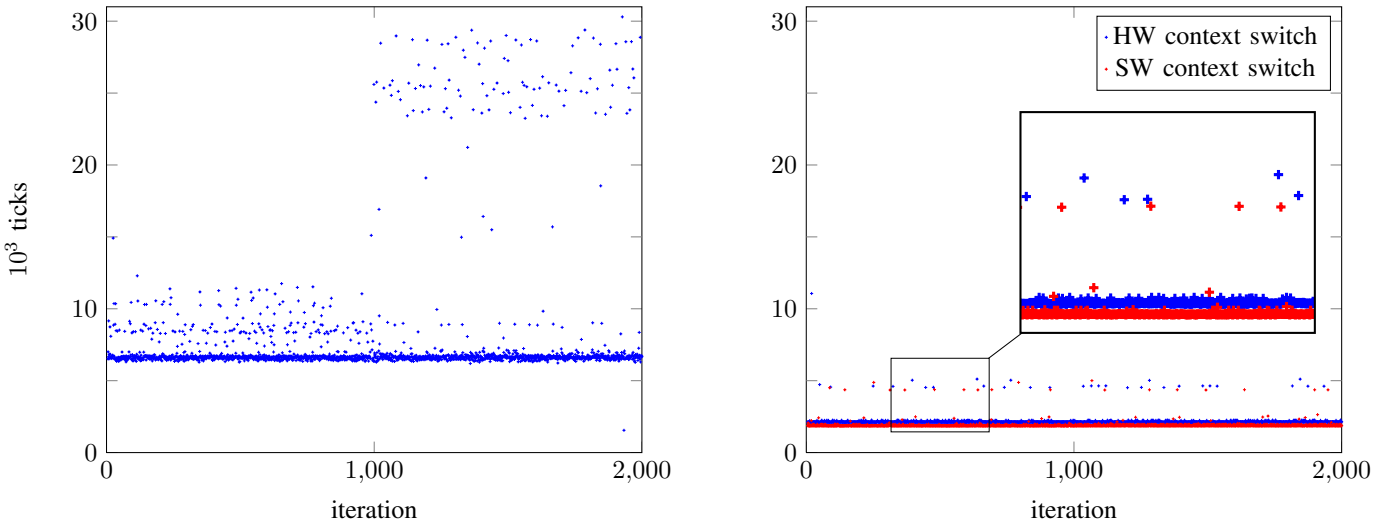
A. Context Switch Latency

For the measurement of context switch latency, two tasks are running on a single core with a high priority. Each task periodically reads the time stamp counter in a loop and stores the result at a shared memory location. Measured gaps, which are shorter than a timeslice and longer than an iteration without interruption, are recorded as an indicator for the latency of a context switch and visualized as a scattered plot in Figure 3. This method is comparable to the hourglass benchmark [22]. But in contrast to our benchmark, the hourglass benchmark tests the general preemption time and gives no information about the context switch latency.

Thus, the benchmark results from Figure 3 can be used for a comparison of context switch latencies between *sccLinux* and *MetalSVM*. As reported by Figure 3a, *sccLinux* has a minimal context switch overhead of about 6400 processor cycles. Figure 3a indicates a certain noise, which has no clear signature and changes from time to time. The picture is different for *MetalSVM*, which generates a minimal context switch overhead of 2100 processor cycles. This is more than 3 times faster. However, Figure 3b shows a second level of about 5000 ticks for context switch latencies. This effect is caused by the process of the lwIP driver, which is running with a high priority.

The scale-up from Figure 3b visualizes the differences between hardware and software context switch for *MetalSVM* on the SCC platform. Here, no significant effect of the context switch method to the context switch latency, except a constant offset, can be identified.

[6] core/mesh/memory frequency: 533 MHz/800 MHz/800 MHz



(a) sccLinux 2.6.38

(b) *MetalSVM*

Fig. 3: Context switch latencies on the SCC (2000 iterations)

B. Hypervisor Performance

The hypervisor plays an important role to establish a transparent shared virtual memory environment. Obviously, its overhead has a significant impact on the performance of its guest machine, for instance concerning memory management, context switches and process handling.

Measurements of three representative latencies identify a reduced virtualization overhead of *lguest* in combination with *MetalSVM*. The *context switch* from guest execution to host execution is performed at each hypercall and at the majority of interrupts. *Page faults* in a guest application can involve up to 3 guest-host roundtrips. Therefore, a fast resolving is aimed for. We measured the duration of *system calls*, exemplary for `getpid`, `fork`, `vfork`, and `pthread_create`. Here, `getpid` indicates the overhead of a system call, since its payload execution time is very low. Due to optimizations in interrupt delivery, `getpid` does not involve a host-guest context switch. The difference of 400 ticks between Linux and MetalSVM as the host operating system can be explained by cache effects. `fork` and `vfork` are used to show the amount of ticks needed for the creation of a task and the copy operation of a whole page directory of the original task. A huge difference between Linux and MetalSVM for the execution time of `pthread_create` is noticeable. This effect can be explained by the coarse granularity of the current timer implementation of *MetalSVM*. Here, the processor frequency has a direct impact.

As a *real-life example* we used a floating point operation intensive application in the form of the jacobi solver algorithm. We measured the overall execution efficiency within the virtual guest machine. Additionally, a second setup indicates the overhead of a task plus floating point context switch by running two instances of the solver.

TABLE I: Benchmark results for the Intel Celeron platform (Linux 2.6.32)

Benchmark	Hypervisor		Ratio $\frac{MSVM}{Linux}$
	Linux	*MetalSVM*	
Host-guest context switch	1 406	1 347	96 %
Page fault	40 426	31 978	79 %
`getpid()`	1 039	626	60 %
`fork()`	446 224	301 831	68 %
`vfork()`	163 421	117 536	72 %
`pthread_create()`	3 678 968	40 022 838	1 088 %
Jacobi solver (128x128 Matrix)	$156 \cdot 10^9$	$99 \cdot 10^9$	63 %
Jacobi solver (2 instances)	$317 \cdot 10^9$	$199 \cdot 10^9$	63 %

Values in processor ticks

The 3 tables (I, II, and III) show the tick count of both hypervisor implementations, Linux and *MetalSVM*, for different stages of the development. The light weight *MetalSVM* kernel results in a successful reduction of overhead for our implementation in combination with memory handling code optimizations of the hypervisor (cf. Table I). However, these measurements were taken at an earlier development stage of the hypervisor.

Table II shows benchmark results of *MetalSVM* version 0.1 and a more recent Linux kernel (2.6.38.3), which is available with *sccKit* 1.4.1 for the SCC platform. The Linux kernel has undergone performance improvements from version 2.6.32 to 2.6.38.3, which affects the benchmark results. However, we see a major advantage of a light weight solution, concerning customizability and transparent performance analysis.

TABLE II: Benchmark results for the Intel SCC platform (Linux 2.6.38.3)

Benchmark	Hypervisor		Ratio $\frac{MSVM}{Linux}$
	Linux	*MetalSVM*	
Host-guest context switch	2 042	2 113	103 %
Page fault	918 679	867 676	94 %
`getpid()`	191	191	100 %
`fork()`	3 216 767	3 101 387	96 %
`vfork()`	220 317	236 207	107 %
`pthread_create()`	16 256 988	10 883 839	67 %
Jacobi solver (32x32 Matrix)	$3.74 \cdot 10^9$	$3.74 \cdot 10^9$	98 %
Jacobi solver (2 instances)	$7.51 \cdot 10^9$	$7.48 \cdot 10^9$	98 %

Values in processor ticks

TABLE III: Benchmark results for the Intel Celeron platform (Linux 2.6.38.3)

Benchmark	Hypervisor		Ratio $\frac{MSVM}{Linux}$
	Linux	*MetalSVM*	
Host-guest context switch	3 020	2 590	86 %
Page fault	40 388	43 985	109 %
`getpid()`	607	595	98 %
`fork()`	351 907	371 381	106 %
`vfork()`	132 142	137 366	104 %
`pthread_create()`	1 020 630	40 049 784	3924 %
Jacobi solver (32x32 Matrix)	$2.04 \cdot 109$	$2.03 \cdot 109$	99 %
Jacobi solver (2 instances)	$4.08 \cdot 109$	$4.13 \cdot 109$	101 %

Values in processor ticks

VII. CONCLUSION AND OUTLOOK

In this paper we presented a bare-metal hypervisor, with the roots of a Unix-like monolithic kernel, used for educational purposes. Our framework extends the software package *sccKit* of the many-core platform to run our configurable light-weight bare-metal programming environment. Performance evaluation of the context switch latency proves the assumption that kernel tasks can be executed close to bare-metal. Thus, broad functionality like interrupt handling and inter core communication in a synchronous as well as asynchronous manner is provided.

This meets the requirements for the integration of an SVM system perfectly, which we have already shown in [4] by using an adapted shared memory application. Here, the light-weight kernel benefits from the efficiency of its subsystems.

The benchmark results of selected system calls for a Linux guest system underline the potential of a bare-metal hypervisor implementation. Considered as a whole, it features a convenient development base for research due to its simplicity and limited base of supported hardware architectures.

For transparent execution of shared memory parallelized applications, we plan to boot and connect multiple instances of the presented kernel and run a single paravirtualized Linux instance on top of the hypervisor layer.

ACKNOWLEDGMENT

The research and development was funded by Intel Corporation. The authors would like to thank especially Ulrich Hoffmann, Michael Konow and Michael Riepen of Intel Braunschweig for their help and guidance.

REFERENCES

[1] *SCC External Architecture Specification (EAS)*, Intel Corporation, November 2010, Revision 1.1. [Online]. Available: http://communities.intel.com/docs/DOC-5852

[2] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the intel scc many-core processor," in *Proceedings of the 2011 International Conference on High Performance Computing and Simulation (HPCS 2011)*, Istanbul, Turkey, July 2011, pp. 525 –532. [Online]. Available: http://dx.doi.org/10.1109/HPCSim.2011.5999870

[3] A. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, 3rd ed. Prentice Hall, 1997.

[4] S. Lankes, P. Reble, C. Clauss, and O. Sinnen, "The Path to MetalSVM: Shared Virtual Memory for the SCC," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*, Potsdam, Germany, December 2011. [Online]. Available: http://communities.intel.com/docs/DOC-19214

[5] S. Lankes, P. Reble, C. Clauss, and O. Sinnen, "Revisiting Shared Virtual Memory Systems for Non-Coherent Memory-Coupled Cores," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2012) in conjunction with the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*, New Orleans, LA, USA, February 2012. [Online]. Available: http://doi.acm.org/10.1145/2141702.2141708

[6] ET International, "ETI's SCC Development Framework available," August 2011. [Online]. Available: http://communities.intel.com/thread/17643

[7] M. Ziwisky, "BareMichael baremetal framework," April 2012. [Online]. Available: http://communities.intel.com/thread/28001

[8] P. Reble, S. Lankes, C. Clauss, and T. Bemmerl, "A Fast Inter-Kernel Communication and Synchronization layer for MetalSVM," in *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011. [Online]. Available: http://communities.intel.com/docs/DOC-6871

[9] C. Clauss, S. Lankes, T. Bemmerl, J. Galowicz, and S. Pickartz, *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*, Chair for Operating Systems, RWTH Aachen University, July 2011, Users' Guide and API Manual. [Online]. Available: http://communities.intel.com/docs/DOC-6003

[10] M. Chapman and G. Heiser, "vNUMA: A Virtual Shared-Memory Multiprocessor," in *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, Jun 2009, pp. 349–362.

[11] G. Heiser, "Many-Core Chips — A Case for Virtual Shared Memory," in *Proceedings of the 2nd Workshop on Managed Many-Core Systems (MMCS)*, Washington, DC, USA, March 2009, p. 4 pages.

[12] D. Schmidl, C. Terboven, A. Wolf, D. an Mey, and C. Bischof, "How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture," in *Proceedings of 2010 IEEE International Conference on Cluster Computing*, September 2010, pp. 29 –37.

[13] J.-A. Sobania, P. Tröger, and A. Polze, "Towards Symmetric Multi-Processing Support for Operating Systems on the SCC," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*, Potsdam, Germany, December 2011.

[14] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," in *Proceedings of the USENIX Annual Technical Conference*, 2002.

[15] S. Lankes, "First Experiences with SCC and a Comparison with Established Architectures," Invited talk at the 1st MARC Symposium, Braunschweig, Germany, November 2010. [Online]. Available: http://communities.intel.com/docs/DOC-5848

[16] R. Russell, "lguest: Implementing the little Linux hypervisor," in *Proceedings of the Linux Symposium, Ottawa, Canada*, 2007.

[17] A. Dunkels, *Design and Implementation of the lwIP TCP/IP Stack*, Swedish Institute of Computer Science, 2001.

[18] R. Russell, "virtio: Towards a De-Facto Standard for Virtual I/O Devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400108

[19] *The sccKit 1.4.x User's Guide*, Intel Labs, October 2011. [Online]. Available: http://communities.intel.com/docs/DOC-6241

[20] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed., A. Oram, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[21] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*, Intel Corporation, August 2007.

[22] J. Regehr, "Inferring Scheduling Behavior with Hourglass," in *Proceedings of the USENIX Annual Technical Conference FREENIX Tracki*, Monterey, CA, USA, June 2002, pp. 143–156.