Bare-Metal Execution of Hard Real-Time Tasks Within a General-Purpose Operating System

Georg Wassen[1] and Stefan Lankes[2]

1 **Operating Systems Research Group, Faculty of Electrical Engineering and Information Technology, RWTH Aachen University**
Aachen, Germany
wassen@lfbs.rwth-aachen.de

2 **Institute for Automation of Complex Power Systems, E.ON Energy Research Center, RWTH Aachen University**
Aachen, Germany
slankes@eonerc.rwth-aachen.de

Abstract

Integrating high performance and real-time demands on multi-processor systems is a challenging task. We present our concept of isolating processes from a general-purpose operating system without deeply invading modifications. This allows executing code on dedicated CPUs with minimum latency and jitter like bare-metal on micro-controllers. The unbounded execution of mixed critical processes on the same system induces performance interference in real-time tasks. We present the implementation of isolated partitions on multi-processor x86 systems running Linux and describe challenges restoring operating system stability. This work also presents our experience with Non-Uniform Memory Access architectures that allow to partition the system in a way that the impact to memory and I/O transfers of other partitions is minimized.

1998 ACM Subject Classification D.4.7 Real-time systems and embedded systems

Keywords and phrases hard real-time system, high-performance computing, non-uniform memory access, bare-metal execution

Digital Object Identifier 10.4230/OASIcs.WCET.2015.75

1 Introduction

Today, multi-processor systems are commonplace from High-Performance Computing (HPC) to small embedded systems. The shift from faster CPUs to multi-processors is more radical than any other architectural change of the last decades [22] because it requires the software paradigm to care for concurrency. This problem is split in two challenges: Firstly to synchronize correctly to avoid in order races and to get a correct result in every execution and secondly to avoid pitfalls that slow down the performance. HPC has a long head start in designing and optimizing concurrent software but this is mainly reserved to domain-specific experts [10, 11]. Evidently, multi-processor systems are also utilized for real-time applications and a large amount of research concentrates on improving the predictability of concurrent tasks in such systems. Various approaches exist that cover a range of applications from completely controlled to a mix of real-time and general-purpose loads. New implementation methods and tool kits are searched for to ease the development of efficient and race-free software. This is especially true for real-time systems, where the additional goal of predictability must be met.

Multi-processor systems generally include multiple CPUs that have access to the entire memory via a global address range. Multi-core processors are CPUs integrated in the

15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015).
Editor: Francisco J. Cazorla; pp. 75–84

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

same package, usually sharing the Last-Level Cache (LLC). If all processors have the same distance to the memory, the system is a Uniform Memory Access (UMA) architecture. In contrast, the Non-Uniform Memory Access (NUMA) architecture places multiple memory regions directly at groups of processors. These nodes have a lower latency and a higher throughput to their local memory but still can access the remote memory. This does not require special programming since all memory regions are laid out in a linear address range, but the performance can be greatly increased if the data is partitioned accordingly [5, 14].

Real-time systems generally require not only a correct result, but also have a timing requirement, usually a deadline after which the value of the result either vanishes abruptly or declines. We define *hard real-time* tasks to require absolutely no missed deadlines and *soft real-time* as more tolerant, e. g. to a certain percentage of missed deadlines. The former demand can only be proved by a formal verification while soft real-time tasks can usually be evaluated practically [17]. The design and verification is split into the path analysis of tasks, conditions and loops in a Control Flow Graph (CFG) and the run-time estimation of basic blocks [24]. The former includes the schedulability analysis of multiple tasks and must regard dependencies and communication. On multi-processor systems, the effort increases but solutions do exist. In contrast, the run-time estimation of small, linear parts of code – basic blocks – must regard architectural features that are often hardly documented or complexly interweaved on powerful commodity processors with multiple levels of caches. So far, this results either in the requirement of simplified models or in an overly pessimistic estimation.

1.1 Contributions

We consider homogeneous processors with a partitioning approach. This usually means either executing multiple OS instances in an Asymmetric Multi-Processing (AMP) layout or binding certain tasks to dedicated processors in a Bound Multi-Processing (BMP) concept. We present our implementation to provide hard real-time capabilities on a single instance of the general-purpose operating system Linux by completely isolating single processes on dedicated CPUs while the other processes are executed unaffected on the remaining CPUs. We extend our basic concept [23] and present detailed experience implementing benchmarks and demonstration applications. We share our experience with partitioning on NUMA systems. This architecture is well established in HPC but – in our opinion – under-represented in recent real-time research. The main contribution of this work is a demonstration of a NUMA system that allows to execute hard real-time tasks with very low jitter as well as compute-intensive applications on the same system. The architecture naturally minimizes the performance interference which simplifies the design and verification.

1.2 Structure

The next Section introduces to our partitioning concept, explains how we implemented fundamental services and presents our demonstration application. Section 3 explains how we use the NUMA architecture and presents the evaluation of memory access patterns. In Sec. 4, we transfer those findings to non-uniform I/O. Section 5 comments on related work and Sec. 6 concludes and provides and outlook to future work.

2 System Architecture

The term *hard real-time* is defined to guarantee a reaction below a defined maximum latency in every possible situation. A program or system can only be proved to comply to this

constraint by a formal analysis of all possible code paths. Besides the code of the real-time tasks, there is other software competing for the processors, foremost the operating system's interrupts and system calls. The analysis of all possible code paths in the OS is beyond practicability for a current general-purpose operating system.

Therefore, we isolate a single hard real-time process on each dedicated CPU. By avoiding system services, we implemented a bare-metal execution that relieves the time-sensitive code from external influences. The usability is restored by providing user-mode communication and device access that allows to interact between isolated tasks and usual processes without introducing new timing dependencies. The remaining influence of hardware effects will be analyzed in the following Sections.

2.1 Application

The isolation concept was implemented by adhering to implementation rules, configuring the base system properly and finally deactivating all interrupts on the isolated CPU. The following system-wide configuration settings are reasonable to support all real-time applications. These settings are available on the x86 architecture, but similar effects are present on other hardware and can usually be addressed by similar means. In our evaluation, some settings can be done in the firmware configuration (BIOS or UEFI setup), others are selected at the Linux boot command line or configured as runtime settings.

Symmetric Multi-Threading (Intel HyperThreading) is deactivated because the logical processors share some execution resources and the local cache in an unpredictable way [6]. The System Management Mode is a special feature of x86 processors to enable the firmware to execute routines for power management, security, etc. It executes at the highest privilege level and can not be influenced or deactivated by the operating system [9]. Experiments on various systems have revealed a duration of System Management Interrupts between 5 μs and 250 ms on all CPUs concurrently. Their use depends on the firmware that may use them frequently. Some can be configured, e. g. by deactivating the legacy keyboard support but every system must be assessed for its acceptability in this regard. The CPU frequency can be adapted by the operating system or the processor itself. These features must be configured to remain at a constant rate. Linux provides an NMI Watchdog that sends non-maskable interrupts to non-responsive CPUs. This feature must be deactivated. Like every real-time and embedded system, all services of the operating system should be evaluated and those not required should be deactivated.

To isolate hard real-time tasks on their dedicated CPUs and guarantee the servicing of all remaining processes, a partitioning concept based on CPU Affinity or the CPU-Set feature was used. CPU-Sets are a convenient mechanism to create groups of CPUs and assign processes to them. That way, isolated partitions can be created for each task. An application partition allows a BMP layout for soft real-time execution of other, less time-critical or best-effort tasks and a system partition gathers all remaining processes and services. To ensure the execution of interrupts, they are assigned either to the CPU of the system partition or to several dedicated CPUs by the means of IRQ Affinity.

The real-time process must be implemented according to the usual recommendations using preallocated buffers and memory locking to avoid page faults and the proper distribution of jobs and the implementation of algorithms suited for real-time. Of course, the application must be correct and error free. The isolated real-time tasks finally clear the interrupt flag to avoid all interruptions. Further, they must avoid all system calls during the real-time operation because those are of unpredictable timing in a General-Purpose Operating System (GPOS).

2.2 Services

For the described approach of isolated tasks on bare-metal processors, the applicability is restored by providing user-mode communication and device access. To interact with the physical world, many real-time applications require reading sensor data and controlling actors. These can be directly connected to I/O adapters that convert analog voltage to digital values and vice-versa. Operating system drivers are not allowed to be used by isolated tasks because their latency would be too large and too unpredictable. But as adapters are controlled either by I/O instructions or by memory-mapping their configuration registers, these devices can also be operated directly by user-mode processes. Devices for real-time applications usually provide libraries for low latency user-mode access.

The initialization and allocation of resources can remain under the control of the operating system and should be done before the time sensitive phase of the program begins. During run-time, the reading and writing of values must then be done directly. Given that interrupts are deactivated, the programming can not follow the event-based paradigm but must use polling. Real-time capable libraries provided by hardware vendors must be assessed for not using system calls during run-time. Network drivers could also be realized in user-space. Shared buffers must be mapped at initialization time and the event-based processing of incoming data must be converted to a polling realization. Further, the network stack must be made real-time capable and running in user-space.

We used shared memory for lightweight communication between threads and processes. During the initialization phase, page-table entries directing to the same physical page frames are created. This hardware based method does not require any operating system support after set-up. The caches are coherent by hardware design. As in all concurrent applications, shared resources must be protected from simultaneous access to avoid using invalid data. If a shared variable (flag) is written always by the same task, no synchronization is required. For complex data structures and message queues, the synchronization could be realized with atomic operations. But hard real-time tasks could be blocked by mutexes held in lower prioritized tasks (priority inversion). This can be solved by applying wait-free algorithms [12].

To simplify application development, we implemented a communication and synchronization library based on shared memory. During the start of a process, the shared memory segment is set up using System V shared memory. This is part of the UNIX specification and available on POSIX compliant operating systems. Within that shared segment, the needed synchronization primitives such as flags and message queues are allocated. The shared resources are protected by atomic operations for mutual exclusion and wait-free algorithms. The design of wait-free message queues is possible [19] and provides a Worst-Case Execution Time (WCET) that is limited independently from other users of the object. This restores the communication capabilities of the isolated process under hard real-time terms.

2.3 Operating System Modification

The deactivation of all interrupts on some CPUs leads to the instability of the underlying Linux system. Among the observed problems are other CPUs locking up due to synchronous inter-processor callbacks, unreliable system wall clock time, and increasing kernel memory consumption. We modified the Linux kernel 3.9 imitating the CPU hotplugging to deactivate all subsystems on a CPU to isolate a task. With this modification, the task does not need to clear the interrupt flag to ensure uninterrupted operation. On x86 systems, clearing the interrupt flag can only be done by the code running on the CPU itself, while our mechanism can be triggered externally. This allows to reactivate a non-responsive isolated CPU.

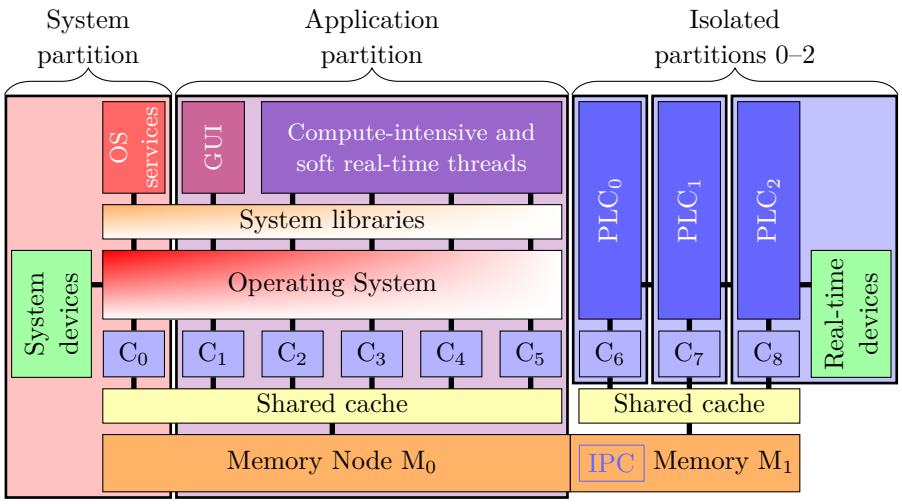

Figure 1 Example application architecture with bare-metal tasks.

2.4 Example

An example application partitioning is shown in Fig. 1. The GPOS with its system services is executed in a dedicated partition. The application partition is used for the main part of the application to execute Graphical User Interface (GUI), compute-intensive tasks and data management. These two partitions can also be combined or split into multiple dedicated partitions depending on the requirements. It is important to create a single hard real-time partition or even dedicated isolated partitions for each CPU where hard real-time tasks are isolated from the compute-intensive loads.

The system and application partitions execute arbitrary services based on standard libraries and use the OS drivers to access devices that have lower timing requirements or that are throughput bound. The isolated partitions use shared memory and wait-free algorithms [12] to communicate with the other partitions and to access their peripheral devices directly using on user-mode drivers or direct I/O. This ensures the lowest latency and the least impact of the GPOS and the remaining application onto the isolated tasks. The isolated tasks can be executed bare-metal or in an embedded Real-Time Operating System (RTOS) implementing user-mode threads.

We implemented this concept on a twelve processor machine with two NUMA nodes. The System and Application partitions are assigned to the processors of the first NUMA node and three isolated partitions are set up on the second node. The remaining CPUs on that node are left idle. This application was developed with an industry partner and is in successfully used in production. The benchmarks described in the following Sections are implemented using the same architecture.

3 Non-Uniform Memory Access

All processors in a Uniform Memory Access (UMA) system share the system bus and the Memory Controller (MC). Cores sharing the inclusive LLC of a multi-core package may even cause the eviction of cache lines from other core's local caches [13]. In contrast, a Non-Uniform Memory Access (NUMA) system has multiple memory partitions attached directly to MCs located in the multi-core packages. These packages are connected by Interconnect

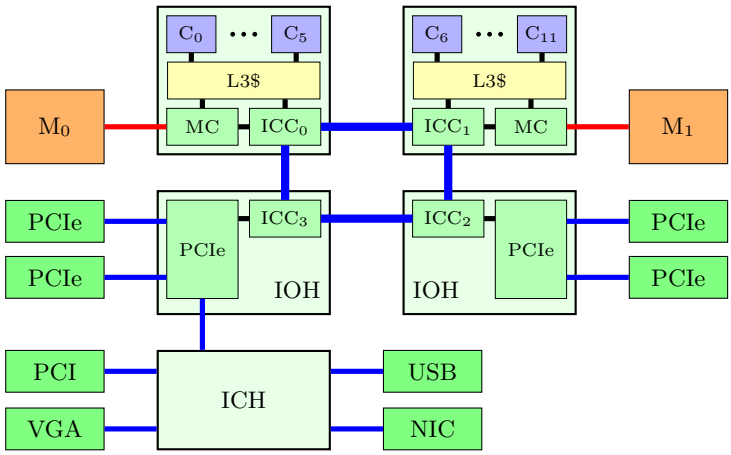

Figure 2 NUMA system architecture of Tyan S7025 main board with two CPU sockets and two I/O bridges.

Controllers (ICCs) that forward memory requests for remote addresses. With the practical knowledge, that shared functional units are potential contention points, we designed the partitioning of the real-time application respecting the NUMA nodes. The system and (soft real-time) applications are executed on different nodes than the isolated tasks.

Our example system (Fig. 2) has two processor sockets each equipped with a six core Intel Xeon E5645 processor. The example application introduced in Fig. 1 maps the system services to core C_0 and the main application to the remaining cores C_1 to C_5 of the first NUMA node. The hard real-time tasks are executed in isolation on dedicated cores of the other NUMA node (C_6 to C_{11}). By default, new memory allocations are placed locally so that the system and the application access mainly the memory node M_0 while the memory regions of the isolated partitions are placed on the other node. Only a small Inter-Process Communication (IPC) buffer is accessed across the interconnect link between ICC_0 and ICC_1.

The benchmark application records the loop execution time for accessing a memory buffer of varying size. At the same time, the main application generates different loads on the other partition. The isolated tasks communicate their state and current timing via wait-free message queues to the main application and watch for a termination signal like in a real application. Figure 3 shows the jitter experienced by the isolated task. The result is independent from the load range as displayed exemplarily for 16 KiB (L1$), 8 MiB (LLC) and 512 MiB as long as the isolated task has a memory usage well below the cache size. Only the memory usage of the isolated task itself increases its own maximum latency if it uses a large part of the LLC. The load does not cause any performance interference on this NUMA system.

The increasing jitter caused by using large parts of the LLC matches the experience on UMA systems. Since every NUMA node is a multi-core processor, their cores impact each other in the same way as all processors in a UMA system do. Therefore, the isolated tasks should restrict themselves as much as possible to the private Second-Level Cache (L2$) of 256 KiB. In this case their maximum latency remains reliably low. Even occasional cache misses compulsorily caused by shared IPC buffers do not interfere fatally. The worst latency of main memory accesses is in the order of 10^6 CPU cycles (Millisecond range) but occurs very seldom and only if the isolation writes very large memory buffers. Minimum and average

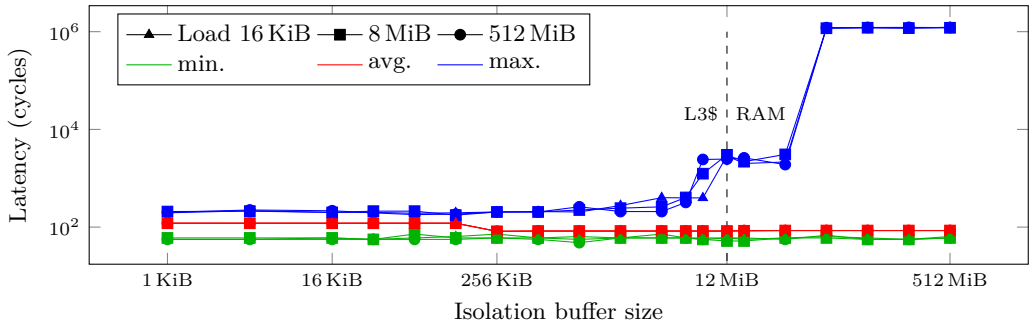

Figure 3 Jitter caused by load on other NUMA node.

latency are not impacted by those events. Our experiments have shown, that the number of events in the range of the maximum latency increases linearly with the isolation buffer size.

4 Non-Uniform I/O

Many hard real-time systems with high frequency and low latency requirements are programmable logic controllers that require direct interaction with physical signals. This can be accomplished on the x86 architecture with data acquisition expansion cards. Their access time in the order of several thousand CPU cycles (Microsecond range) is low compared to what the processor could calculate in this period. But similarly to memory access, the jitter caused by device access is even worse on multi-processor systems because of system bus contention [21]. To solve the problem of unpredictable jitter, co-scheduling of tasks [16] and hardware extensions [1] have been proposed. Since we aim for unrestricted concurrent execution on available systems, these approaches are inapplicable.

Analogous to the memory transfers, in UMA systems all I/O traffic uses the same system bus. In the same way that NUMA architectures are capable of separating contending memory transfers of system applications and time-critical tasks, they also offer a solution for predictable I/O transfers. Additionally, PCI-Express (PCIe) offers prioritized data streams that can be transferred concurrently over switched point-to-point connections without interfering [2]. Like the NUMA architecture's natural partitioning capabilities improving the timing predictability of memory accesses, we expect that the PCIe connections in NUMA architectures are better suited for hard real-time systems than PCI and UMA.

The main board architecture shown in Fig. 2 has two I/O bridges (IOH) that are connected to the CPU sockets in a QuickPath interconnect forming a ring. We therefore connect the peripheral devices used by the real-time tasks to the PCIe slots connected to the right-side IOH. The system partition controls the standard devices (graphics adapter, USB, Ethernet) that are provided by the ICH which is connected to the left IOH. The only data transfers crossing the partition border are still the memory transfers to the IPC buffers.

The test series uses memory-mapped I/O accesses to different devices that are connected to the built-in PCI slot managed by the ICH and to a PCIe slot of the real-time partition. We also measured a PCIe to PCI bridge installed in the real-time partition and one of the PCIe slots assigned to the system partition but those gave similar results as the real-time partition PCIe connection. The benchmark was executed in isolated tasks and recorded a statistic of latencies during high memory and I/O loads executed in the system partition.

The results are displayed in Fig. 4. The large I/O overhead of the x86 architecture in the range of 2000 to 5000 CPU cycles (1–2 μs) covers outliers and jitter of I/O transfers.

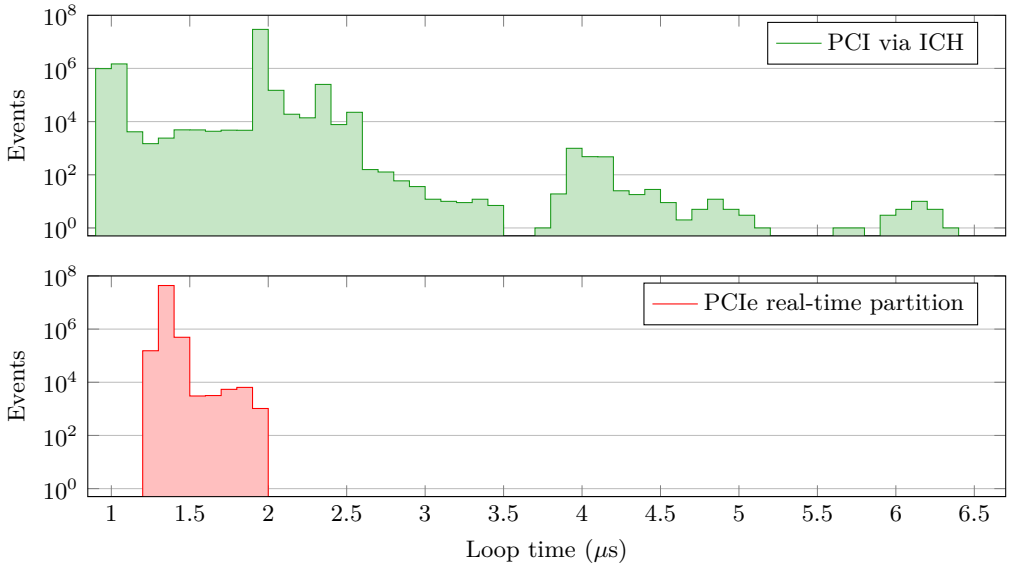

Figure 4 Distribution of access latencies to different I/O locations under system and I/O load.

The PCI device connected to the chipset's ICH has access latencies up to 6 μs under high system load while the access to the PCIe devices is more predictable between 1.2 and 2 μs. The results of the other PCIe partition and a PCIe to PCI bridge in the real-time partition are very close to the lower histogram with a constant offset of approx. 0.2 μs. The PCIe devices are not influenced by the load. The transfers using the PCIe interconnect can be completed in bounded time while the ICH (former south bridge) is easily contended. It remains as future work to verify the interconnect architectures and to evaluate more systems and more recent CPU architectures. However, we have completed extensive test series with uninterrupted execution times up to 72 hours that support these results on our test system.

5 Related Work

Real-time systems with multiple processors have been researched thoroughly [4, 18]. Current approaches [25] to tightly estimate the execution time on multi-processor systems either simplify the architecture [3] or restrict the execution on the entire system by strictly controlling all tasks [15]. In contrast, we aim for allowing arbitrary applications in the system partition and present a method to restrict performance interference. A very deep analysis of the causes of jitter in x86 systems is presented by Dasari et. al. including the memory bus and caches [8] as well as I/O [7]. Some publications mention NUMA systems in their plans for future work [7, Sect. IV.C] or just explain theoretical foundations [20] but they do not analyze the opportunities of such systems. All evaluations of multi-processor I/O so far only regard UMA systems and the older PCI bus. Stohr includes PCIe in his analysis of using the x86 architecture for real-time systems [20] but does not cover NUMA and non-uniform I/O. Despite an extensive literature review, these works cover only UMA systems and even if mentioning NUMA systems, those are only briefly touched and either dismissed or left for future work. To the best of our knowledge, no publications present experience with hard real-time applications on current NUMA systems do exist so far.

6 Conclusion

We demonstrated the value of the NUMA architecture that is underrepresented in hard real-time systems research, so far. With the spacial partitioning of AMP and BMP real-time systems to distinct processors, NUMA systems allow to also partition both memory and I/O transfers. Extensive test series fortified the assumptions derived from HPC experience and architectural analyzes. However, this is only shown for the used CPU architecture (Intel Westmere) and main board (Tyan S7025). Nevertheless, it indicates that a careful design which considers the NUMA architecture is in fact capable to reduce the performance interference and to provide hard real-time guarantees on commodity x86 systems without restricting the load in the system partition.

This concept was implemented in a Hardware-in-the-Loop simulator for industrial integration testing. Compute-intensive high-frequency hard real-time tasks could successfully be ported from special micro-controllers to a standard x86 server system. Other application fields are FPGA codes, that could be integrated easier with existing complex applications to create mixed critical systems.

Acknowledgments. This research was supported by the Chair for Operating Systems at RWTH Aachen University.

References

1 Stanley Bak, Emiliano Betti, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Real-Time Control of I/O COTS Peripherals for Embedded Systems. In *Proc. 30th IEEE Real-Time Systems Symp. (RTSS)*, pages 193–203. IEEE, 2009.

2 Ajay V. Bhatt. Creating a PCI Express Interconnect. White paper, Technology and Research Labs, Intel Corp., Santa Clara, CA, USA, 2002.

3 Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic Execution Model on COTS Hardware. In Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte, editors, *Proc. 25th Int. Conf. Architecture of Computing Systems (ARCS)*, volume 7179 of *LNCS*, pages 98–110. Springer, 2012.

4 Björn B. Brandenburg, John M. Calandrino, and James H. Anderson. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. In *Proc. 29th IEEE Real-Time Systems Symp. (RTSS)*, pages 157–169. IEEE, 2008.

5 Timothy Brecht. On the importance of parallel application placement in NUMA multiprocessors. In *4th Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pages 1–18, 1993.

6 Steve Brosky and Steve Rotolo. Shielded Processors: Guaranteeing Sub-millisecond Response in Standard Linux. In *Proc. 2003 IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*. IEEE, 2003.

7 Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. Identifying the Sources of Unpredictability in COTS-based Multicore Systems. In *Proc. 8th IEEE Int. Symp. on Industrial Embedded Systems (SIES)*, pages 39–48. IEEE, 2013.

8 Dakshina Dasari, Björn Andersson, Vincent Nélis, Stefan M. Petters, et al. Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *IEEE 10th Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1068–1075. IEEE, 2011.

9 Shawn Embleton, Sherri Sparks, and Cliff Zou. SMM Rootkits: A New Breed of OS Independent Malware. In *Proc. of the 4th Int. Conf. on Security and Privacy in Communication Networks (SecureComm)*, pages 11:1–11:12. ACM, 2008.

10 Agner Fog. *Software optimization manuals.* Technical University of Denmark, Copenhagen, Denmark, 2013. `http://www.agner.org/optimize/`.

11 Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, Inc., Boca Raton, FL, USA, 2010.

12 Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

13 Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches. In *Proc. 43nd Annu. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, pages 151–162. IEEE, 2010.

14 Zoltan Majo and Thomas R. Gross. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In *Proc. Int. Symp. on Memory Management (ISMM)*, pages 11–20. ACM, 2011.

15 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, et al. A Predictable Execution Model for COTS-based Embedded Systems. In *Proc. of 17th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 269–279. IEEE, 2011.

16 Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In *Proc. 29th IEEE Real-Time Systems Symp. (RTSS)*, pages 221–231. IEEE, 2008.

17 Peter Puschner and Martin Schoeberl. On Composable System Timing, Task Timing, and WCET Analysis. In Raimund Kirner, editor, *8th Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 8, Dagstuhl, Germany, 2008. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

18 Simon Schliecker and Rolf Ernst. Real-time Performance Analysis of Multiprocessor Systems with Shared Memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, 2011.

19 Philippe. Stellwag, Alexander Ditter, and Wolfgang Schröder-Preikschat. A Wait-Free Queue for Multiple Enqueuers and Multiple Dequeuers Using Local Preferences and Pragmatic Extensions. In *Proc. 4th IEEE Int. Symp. on Industrial Embedded Systems (SIES)*, pages 237–248. IEEE, 2009.

20 Jürgen Stohr. *Auswirkungen der Peripherieanbindung auf das Realzeitverhalten PC-basierter Multiprozessorsysteme.* Doctoral dissertation, Fakultät für Elektrotechnik und Informationstechnik, TU München, Munich, Germany, March 2006.

21 Jürgen Stohr, Alexander von Bulow, and Georg Färber. Bounding Worst-Case Access Times in Modern Multiprocessor Systems. In *Proc. 17th Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 189–198. IEEE, 2005.

22 Herb Sutter. The Free Lunch Is Over. *Dr. Dobb's Journal*, 30(3), March 2005.

23 Georg Wassen, Stefan Lankes, and Thomas Bemmerl. Harte Echtzeit für Anwendungsprozesse in Standard-Betriebssystemen auf Mehrkernprozessoren. In Wolfgang A. Halang, editor, *Herausforderungen durch Echtzeitbetrieb*, Informatik aktuell, pages 39–48, Berlin, Germany, 2012. Springer.

24 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, et al. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

25 Reinhard Wilhelm and Jan Reineke. Embedded Systems: Many Cores – Many Problems. In *Proc. 7th IEEE Int. Symp. on Industrial Embedded Systems (SIES)*, pages 176–180. IEEE, 2012.