Programmable Temporal Isolation in Real-Time and Embedded Execution Environments*

Silviu S. Craciunas Christoph M. Kirsch Hannes Payer Harald Röck Ana Sokolova

firstname.lastname@cs.uni-salzburg.at
Department of Computer Sciences
University of Salzburg, Austria

ABSTRACT

We begin this paper with a wish list of features that we feel a modern real-time and embedded execution environment should offer. We then look at some of the key weaknesses of conventional real-time operating systems (RTOS) and limitations of virtual execution environments (VEE), which typically offer some of the features but not all in one system. We propose to remedy the problem by carefully combining, in a single virtualized execution environment, well-known operating systems and virtualization techniques with an efficient real-time scheduler, which we have recently developed. The scheduler enables temporal isolation of concurrently executing processes and allows to change guaranteed process execution speeds efficiently at any time during execution. We also report on preliminary experiments with a prototypical bare-metal implementation.

1. INTRODUCTION

Firstly, the ideal real-time and embedded execution environment should provide strong temporal and spatial isolation. The temporal behavior of programs running on the system should be determined by application developers and not by the particularities of the underlying hardware platforms or any competing software workloads. In particular, the exact speed at which programs execute should be controlled and adjustable at any time. Programs should be protected from each other in space, when accessing memory and I/O devices, for security and fault isolation. Ideally, programs should also be offered strict bandwidth guarantees (throughput and latency of I/O and IPC) as well as direct and exclusive access to special devices if needed.

*This work is supported by a 2007 IBM Faculty Award, the EU ArtistDesign Network of Excellence on Embedded Systems Design, and the Austrian Science Fund No. P18913-N15.

IIES'09 March 31st, 2009, Nuremberg, Germany.

Secondly, the execution environment should support a variety of program execution models ranging from operating system processes to system virtual machines, which enable system-level virtualization by duplicating real hardware in software, and process virtual machines, which implement abstract machines and memory architectures such as Java virtual machines and their object models. A system VM allows to run legacy software that has originally been developed for the hardware the VM duplicates, including whole operating systems and their device drivers. The execution environment itself may therefore not need to implement drivers for all devices but instead route at least the non-real-time I/O traffic to an instance of a system VM that duplicates the hardware on which the execution environment runs and for which legacy device driver software exists. XEN is a prominent example of a virtual machine monitor or hypervisor, which provides the necessary software infrastructure for running such system VMs [2]. Support of process VMs, however, is equally important. Ideally, the execution environment should support the implementation of process VMs that can directly be targeted by a large range of high-level programming models with and without real-time support.

Thirdly, the implementation of the execution environment should be small, simple, and efficient, and thus amenable to verification as well as exact complexity and execution time analyses. Ideally, the time any invocation of the system may take (system latency) should be bounded by a known constant. The variance in latency (system jitter) should not only be low but also known. The time any preemptive system activity such as I/O transmission and memory allocation and deallocation may take in total (system throughput) should be at most linearly bounded in terms of the involved workload independently from the system state. Again, the variance in throughput should be low and known. Such an implementation may create an execution environment that can offer real performance guarantees on all relevant aspects of the system such as execution speed, memory allocation and deallocation rates, and I/O bandwidth.

Many RTOS (Section 2) as well as many VEE (Section 3) provide some of the features on our wish list but typically not all in one system. There appear to be at least two key

problems with conventional RTOS, related to lack of isolation and complexity of their execution models. Temporal behavior of programs is either underspecified through some non-compositional form of priorities or deadlines, or else overspecified through inflexible, static allocation techniques. RTOS process abstractions are often low-level and non-compositional, and therefore highly machine-dependent and difficult to use, in particular, in complex applications. While most VEEs offer strong spatial isolation and standardized execution models, there is typically no strict control over temporal isolation, or the non-compositional or inflexible techniques from the RTOS world are reused. I/O performance in VEEs is usually also difficult to isolate. System complexity is another issue present in most VEE but also many RTOS implementations, which makes their verification and analysis difficult.

An important step in understanding how to address these issues is to define the notion of temporal isolation. Intuitively, the execution of a piece of sequential program code of a process (called action) is temporally isolated if the response times of the code as well as the variance of the response times (jitter) are solely determined by the code itself and its inputs, independently of any other, concurrently executing actions and the system on which the actions execute. The response time of an action is the duration from the time instant when process execution reaches the beginning of the action (arrival) until the time instant when process execution reaches the beginning of the next action (termination). In this model, process execution corresponds to a possibly infinite sequence of actions. We say that temporal isolation is programmable if response times and jitter can be modified by processes, at least within some platform-dependent range.

We have recently designed, implemented, and analyzed a real-time $O(1)$-scheduler, which uses so-called variable-bandwidth servers (VBS) to isolate process execution temporally on the level of individual process actions [9]. Upon arrival of new processes, the scheduler checks in constant time if there are still sufficient resources left to guarantee temporal isolation. A VBS executes a process for a guaranteed number of time units (called limit) periodically at a guaranteed rate (called period). Limit and period can be changed within a guaranteed range from one action to the next. For example, a latency-oriented piece of code can be executed with a small period (and limit), followed by a throughput-oriented piece that can be executed with a larger period (and limit) and thus fewer interruptions. Being able to switch guaranteed limits and periods efficiently at any time is one of the key features of the scheduler.

The scheduler has become the center piece of our prototypical implementation of a small-footprint, bare-metal VEE called Tiptoe [10]. The prototype has been written from scratch for studying possible ways to realize our wish list of features in a single execution environment. We discuss how VBS may help to extend temporal isolation beyond process execution to other system activities such as memory allocation and deallocation as well as I/O and IPC management (Section 4). The key idea is to focus on subsystems that guarantee at most a linear relationship between the amount of actual work involved in their activities (e.g. object or packet size) and the necessary CPU time to process the work. An integration with VBS would allow us to control temporal isolation of system activities on the level of the actual workloads. We have already designed and implemented an I/O-relaying subsystem [19] and a compacting real-time memory management system [11] with this property but have not yet integrated them into Tiptoe.

Our Tiptoe prototype currently runs on an XScale processor and implements a system VM that virtualizes an Atmega128 processor and interprets arbitrary AVR code. The prototype system schedules multiple VM instances using VBS. We report on preliminary results in a bare-metal experiment showing that Tiptoe can temporally isolate a process executing AVR code of two different actions in the presence of an increasing number of concurrently executing processes up to full CPU utilization (Section 5). We plan to integrate the concepts evaluated in Tiptoe into a microkernel that supports virtualization, e.g. OKL4 [14], but with our VBS scheduler at its core. The goal is to run temporally isolated processes as well as instances of system and process VMs.

2. CONVENTIONAL RTOS

Most RTOS offer some form of process prioritization or static allocation scheme to schedule process execution. Prioritization is typically done either directly through process priorities or indirectly through periodic rates or relative deadlines. The most popular real-time scheduling strategies based on rates and deadlines are rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling, respectively. Static allocation schemes range from low-level, time-slot-based methods such as TTA [16] to higher-level, bandwidth-based approaches such as resource reserves [18] or constant-bandwidth servers [1], which are a less flexible form of VBS restricted to fixed bandwidth configurations.

Many popular RTOSs used in industry, e.g., QNX, VxWorks, and ThreadX come with a priority-based scheduler while more academic RTOS projects, like Hartik [4], Erika [13], and Spring [20] use EDF. This phenomenon may be explained by several misconceptions about the differences between RM and EDF scheduling in practice. Typically, the reasons favoring RM scheduling, given by industry, are that RM scheduling is easier to implement and analyze, is more predictable during transient overload scenarios, and introduces less runtime overhead in comparison to EDF. More recently, however, the differences have been analyzed more carefully and shown either to be false or not to hold in the general case [5].

Nevertheless, we argue that neither process prioritization, including RM and EDF scheduling, nor static alloca-

tion alone can solve the challenges in designing and implementing modern real-time and embedded execution environments. Process prioritization will ultimately fail because realizing compositionality, i.e., programmable temporal isolation, is impractical using priorities, whereas static allocation has potential [15] but is either too inflexible or too inefficient in most of its current implementations.

In addition to the problem of temporal isolation, many conventional RTOS have further shortcomings. Existing implementations tend to be highly specialized systems, which require specialized programmer expertise. RTOS often come with a non-standard API, which makes code development and maintenance expensive. RTOS usually do not attract much attention outside the real-time community and have therefore failed to create a critical mass of developers. Many RTOS are not open-source, which makes it difficult for third parties to modify system components if needed. Although some RTOS support standard APIs, like POSIX, running certain application software and especially device drivers still requires non-trivial porting efforts. Moreover, building heterogeneous applications that involve non-real-time and real-time processes is sometimes difficult. Furthermore, there are RTOS that execute real-time processes in kernel space, e.g., RTLinux [23], and thus have little or no spatial protection against faulty or malicious code.

3. VIRTUALIZATION

The key benefits of system-level virtualization, not only for embedded systems but in general, are legacy software support, device driver support, and spatial isolation of the system VMs. Legacy software support is provided by system VMs that virtualize the hardware for which the legacy software has originally been developed. Device drivers of an existing operating system can be reused by relaying device access to the system VM on which the operating system with the driver is running. For instance, running Linux on an embedded system VM would grant access to a large set of applications and devices for which drivers are already available in the Linux kernel. Isolation of system VMs guarantees that faulty or malicious code running in one VM cannot propagate to other VMs.

In recent years, system-level virtualization has been a popular research topic in academia and industry. The main research focus is in the field of server and personal computing where products like XEN [2], VMware [21], and VirtualBox [22] have already been applied successfully.

System-level virtualization is also gaining popularity in embedded systems circles since modern embedded hardware is getting ever more powerful allowing to run more complex embedded applications. There is also a growing demand for running application software which has originally been designed for general purpose operating systems, on embedded hardware. Spatial isolation provided by system-level virtualization even allows to separate critical systems code, e.g., the communication stack on cell phones, from other non-critical application code such as web browsers and media players.

However, Heiser [14] argues that system-level virtualization alone is not sufficient for embedded systems. He proposes instead a hybrid approach based on the microkernel system OKL4, which allows to perform system-level virtualization and to run other non-virtualized elements at the same time. For example, OKL4 can manage standard operating system processes as well as multiple system VMs running standard operating systems such as Linux. A microkernel like OKL4 also supports light-weight but strong encapsulation of interacting components from which new real-time applications could be build.

We agree with Heiser on the issue of providing a variety of execution models ranging from operating system processes to system- and process-level virtualization, but we also emphasize the importance of temporal isolation. OKL4 only provides a simple priority scheduler and no further support for temporally isolating memory allocation and deallocation as well as I/O and IPC management. We argue that temporal isolation throughout the system is not only important for safety- and mission-critical applications but increasingly also for convenience applications as software complexity tends to increase with hardware performance and therefore may become difficult to manage technically and economically without a compositional and flexible timing model. Support of temporal isolation, however, may involve modifications of even the most fundamental parts of the system.

4. SYSTEM INTEGRATION

We begin by describing the current state of our system design and prototype implementation, followed by a short discussion of future plans. We then provide more details on particular system components such as the scheduler, the memory management including our explicit real-time memory management system called Compact-fit (CF) [11], and the channel subsystem, which handles I/O and IPC. The scheduler has already been integrated into the system whereas our memory management and the channel subsystem are still experimental stand-alone components.

We have developed a small-footprint, bare-metal, real-time VEE called Tiptoe [10], which currently runs on an XScale 400MHz-processor with 64MB RAM. Tiptoe comes with its own C library, device drivers for setting I/O pins, an Ethernet driver, a serial driver, a microsecond timer framework, and a 1KHz timer interrupt to keep the system synchronized with real time. Furthermore, we have implemented a VM that virtualizes an Atmega128 processor with 4KB RAM and 128KB Flash storage, and interprets arbitrary AVR assembler code. Tiptoe can run basic real-time processes and multiple, virtual Atmega128 instances.

Tiptoe is meant to be a microkernel-based system that supports executing temporally isolated operating system processes as well as system and process VMs. We are currently exploring different approaches to accomplish this like implementing a system from scratch or reusing an existing

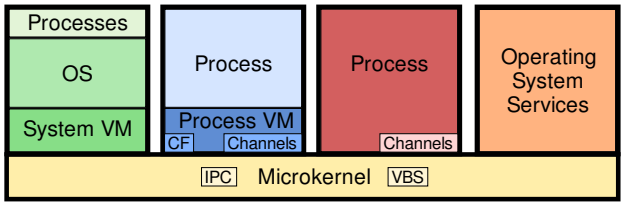

Figure 1: Tiptoe system design

microkernel such as OKL4. The key problem is to design all system components such that there always exists at most a linear relationship between the amount of CPU time required by each component to process a workload and the actual amount of the workload. In this case, the usage of system components would be temporally isolated as well.

Figure 1 shows the Tiptoe system design. On the lowest level, there is a microkernel, which contains the VBS scheduler and an IPC mechanism. On top of the microkernel, processes using the channel subsystem, and operating system services, e.g., device drivers, may run along with operating system instances encapsulated in system VMs, and process VMs, which may take advantage of CF and the channel subsystem. Scheduling parameters for the VBS scheduler are set via system calls.

Scheduling

Tiptoe uses a real-time $O(1)$-scheduler for scheduling all system activities. The scheduler [9] is based on the previously mentioned notion of variable-bandwidth servers (VBS). Tiptoe assigns each scheduling task, i.e., process or VM instance, in the system to a unique VBS, which essentially controls the execution speed of the assigned task and may even change the speed at any time upon request.

A VBS is configured by a single number that determines a utilization bound called bandwidth cap in percentage of CPU time. Upon creating a new VBS, the system checks if the sum of the bandwidth caps of the existing VBS and the new VBS is still less than or equal to the system's total capacity. The admission test for running new processes or VM instances is therefore also constant-time.

To configure their actual execution speed, each scheduling task chooses a pair (λ, π) called virtual periodic resource such that λ over π is less than or equal to the bandwidth cap of the VBS to which the task is assigned. The λ and π values correspond to the previously mentioned notions of limit and period, respectively. The VBS will then execute the assigned task for λ units of time every π units of time. The task can switch at any time to a different virtual periodic resource within the range of the bandwidth cap of the VBS, marking the beginning of a new what we previously called action.

The key property of VBS is programmable temporal isolation. If the admission test succeeds for a new VBS, the system guarantees the VBS that the assigned task, upon choosing a virtual periodic resource (λ, π) within the bandwidth cap of the VBS, will be executed for λ units of time every π units of time, with at most one π delay from the point in time when the resource is chosen. Intuitively, what happens is that the VBS must "re-synchronize" with the new virtual periodic resource every time the resource is changed. We tolerate the delay of at most one period because it makes the admission test simple and constant-time. There appears to be a fundamental trade-off between scheduling efficiency and admission complexity. Note that the admission test comes at the expense of precision. Even if the test fails, there may be system configurations in which a VBS could guarantee temporal isolation. However, a more precise test will have to consider the system in more detail and therefore be more expensive.

VBS guarantees that the response times of a process or VM instance is temporally isolated on the level of their individual actions, i.e., portions of code, from any other system activities. Moreover, the response times may only vary within at most one period of the chosen virtual periodic resources. Therefore, the smaller the periods are the smaller the response time jitter will be, however, at the expense of higher administrative overhead through more frequent scheduler invocations. Conversely, the larger the periods are the higher the net CPU throughput will be due to fewer interruptions.

The bound on response time jitter only holds under the assumption that scheduling overhead is zero. In fact, it turns out that in practice the jitter may be more than one period because of the non-zero scheduling overhead, see Section 5. The jitter is nevertheless still bounded but only according to a more complex relationship between scheduling overhead and the periods in the system. We are currently working on a precise formulation.

Legacy code not using VBS and not expecting any guaranteed response times may run outside of the real-time scheduling domain, i.e., during idle time. Multiple non-VBS processes are scheduled (during the idle time of the real-time scheduler) using another scheduling policy such as round-robin.

Memory Management

Similar to other hypervisors and microkernels (e.g. [2, 14]), Tiptoe divides physical memory into pages and provides an interface for processes to request new pages, to return pages not needed anymore, and to update a process' page table entries. A process is responsible to establish the mapping of virtual memory addresses to physical memory addresses using the interface to the microkernel.

Moreover, process VMs that use an object-based memory model may use our Compact-fit explicit memory management system [11] to manage their internal heaps in real time. CF implements real-time versions of malloc and free, and bounds fragmentation through real-time compaction such that the available memory for a given object size can always be determined in constant time. Memory analysis tools like [8] may therefore effectively bound heap sizes required to run programs, if CF is used.

CF memory allocation takes constant time. CF memory deallocation also takes constant time unless compaction has to be performed, which takes linear time in the size of the deallocation request. We plan to integrate CF into Tiptoe such that memory allocation and deallocation rates of process VMs using CF can be effectively controlled through the VBS scheduler. This may be possible since the size of memory objects involved in any allocation and deallocation requests fully determines the maximum CPU time needed to perform the requests, independently of the state of the system.

Channels

Motivated by the VBS concept, we define a channel as a communication link between two processes with a bandwidth cap on the link. For each data-transfer on a channel, a process specifies the total amount of data involved in the transfer (workload), a chunk size (limit), and a rate (period). The chunk size is the amount of data that is guaranteed to be transferred in a single period at the specified rate. The resulting throughput, i.e., the chunk size over the rate, must be less than or equal to the bandwidth cap of the channel. Note that this definition of a channel enables calculation of the response times for a data transfer, depending on the workload, the chunk size, and the rate.

In Tiptoe, communication of processes with device drivers is also handled by channels since device drivers may either run as special system service processes, just like in other microkernels, or else run in an operating system instance encapsulated in a system VM running on top of the microkernel. As a result, channels may not only allow to control communication between processes or VM instances but may eventually also provide temporal isolation when performing I/O. There is also a stand-alone I/O subsystem that we developed earlier and may use, which relays all I/O traffic to a different device host running Linux [19]. This approach requires multiplexing I/O in real time on to a communication link to the device host but is otherwise similar to relaying I/O traffic to drivers running in an operating system instance encapsulated in a system VM.

For the VBS scheduler, a data-transfer on a channel is just like any other process action, i.e., piece of sequential program code. The action's period π is the application-defined rate for the data-transfer, whereas the channel has to provide a value λ for the action's limit. In the best case, λ is the exact time needed to send a data chunk. Hence the best case is when the channel can map a workload specified in bytes to the exact amount of time needed by the channel to process the workload. However, mapping workload to time is difficult and hardware-dependent. A limit value λ that is larger than the time needed to send the data chunk may result in a violation of the bandwidth cap of the channel but can be dealt with by having the scheduler delay the completion of the data transfer artificially by blocking the invoking process until the "undeserved" amount of time has elapsed [12]. A problem appears when the limit value λ is smaller than the time needed to send the data chunk, since in this case the guaranteed response time of the data transfer action may be exceeded. Determining an adequate value for λ is related to the server design problem, c.f. [17, 3]. Worst-case execution time analysis and performance measurements could help approximate the calculation of the limit value λ.

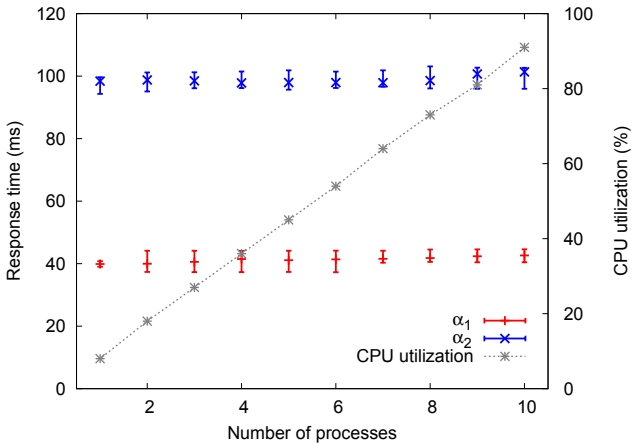

Figure 2: VM experiment

To conclude, channels help fulfill the following Tiptoe goal: both temporal isolation (via the use of VBS) and application-relevant workload-determined response time (via the use of channels) must be guaranteed for each process.

5. EXPERIMENT

Using our current implementation, we show the temporal isolation capabilities of Tiptoe provided by our scheduler as well as its support for adapting the execution speed of portions of code to different latency requirements. Consider a process implementing a simple feedback controller that consists of two actions. The first action (α_1) reads sensor values, computes a new control command, and writes the actuators. In control systems, this portion of a control process requires low input/output latency. The second action (α_2) updates the state of the controller and has less stringent timing requirements. Latency and jitter are critical to control systems, thus splitting a control process in two actions improves controller performance [6, 7]. We specify latency requirements by associating each action with a relevant virtual periodic resource. Action α_1 is associated with the virtual periodic resource $R_1 = (320\mu s, 3550\mu s)$ while action α_2 uses the resource $R_2 = (500\mu s, 5340\mu s)$. The process utilizes the CPU at around 9%. In order to show temporal isolation, we increase system utilization by starting additional processes, each utilizing the CPU at around 10%. We chose the limits and periods of the actions so that system utilization is maximal when running a total of 10 processes. Note that all processes execute actual AVR code but without accessing any real sensors and actuators.

Figure 2 shows the minimum, maximum, and average re-

sponse times of action α_1 and α_2, respectively (left *y*-axis). The response time jitter of each action varies within two periods of the virtual periodic resource used by the respective action independently of the overall system utilization (right *y*-axis). CPU utilization increases from 9% when the measured process is the only process in the system up to 92% when 9 additional processes run concurrently with the measured process (*x*-axis).

The theoretical bound for jitter discussed in Section 4 is one period assuming zero scheduler overhead. In a real system, however, the administrative overhead causes the response time variance of an action to exceed one period. Nevertheless, this variance is still bounded and not influenced by the system utilization. The real bound depends on the scheduler overhead and the periods of the actions scheduled in the system. Giving guaranteed bounds with non-zero scheduler overhead is a topic that we plan to pursue as future work.

6. CONCLUSIONS

We analyzed current real-time and embedded execution environments and proposed to combine well-known operating systems and virtualization techniques with a real-time scheduler that provides programmable temporal isolation. We presented details and early results of our recently developed scheduler and its use of variable-bandwidth servers (VBS) to achieve temporal isolation across varying system and CPU load. Furthermore, we discussed how the concepts behind VBS may be used to extend temporal isolation to other system activities like memory and I/O management. Finally, we presented a bare-metal experiment that shows the capability of programmable temporal isolation of Tiptoe processes, independent of the CPU utilization.

7. REFERENCES

[1] L. ABENI, AND BUTTAZZO, G. Resource reservation in dynamic real-time systems. *Journal of Real-Time Systems 27*, 2 (2004), 123–167.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proc. SOSP* (2003), ACM, pp. 164–177.

[3] BUTTAZZO, G., AND BINI, E. Optimal dimensioning of a constant bandwidth server. In *Proc. RTSS* (2006), IEEE, pp. 169–177.

[4] BUTTAZZO, G. C. HARTIK: A real-time kernel for robotics applications. In *Proc. RTSS* (1993), IEEE, pp. 201–205.

[5] BUTTAZZO, G. C. Rate monotonic vs. EDF: Judgment day. *Real-Time Systems 29*, 1 (2005), 5–26.

[6] CERVIN, A. Improved scheduling of control tasks. In *Proc. ECRTS* (1999), IEEE, pp. 4–10.

[7] CERVIN, A., AND EKER, J. The Control Server: A computational model for real-time control tasks. In *Proc. ECRTS* (2003), IEEE, pp. 113–120.

[8] CHIN, W.-N., NGUYEN, H. H., POPEEA, C., AND QIN, S. Analysing memory resource bounds for low-level programs. In *Proc. ISMM* (2008), ACM, pp. 151–160.

[9] CRACIUNAS, S., KIRSCH, C., RÖCK, H., AND SOKOLOVA, A. Real-time scheduling for workload-oriented programming. Tech. Rep. 2008-02, University of Salzburg, September 2008.

[10] CRACIUNAS, S. S., KIRSCH, C. M., PAYER, H., RÖCK, H., SOKOLOVA, A., STADLER, H., AND STAUDINGER, R. The Tiptoe system, 2007. `tiptoe.cs.uni-salzburg.at`.

[11] CRACIUNAS, S. S., KIRSCH, C. M., PAYER, H., SOKOLOVA, A., STADLER, H., AND STAUDINGER, R. A compacting real-time memory management system. In *Proc. ATC* (2008), USENIX, pp. 349–362.

[12] CRACIUNAS, S. S., KIRSCH, C. M., AND RÖCK, H. I/O resource management through system call scheduling. *SIGOPS Operating System Review 42*, 5 (2008), 44–54.

[13] GAI, P., LIPARI, G., AND DI NATALE, M. A flexible and configurable real-time kernel for time predictability and minimal RAM requirements. Tech. rep., Scuola Superiore S. Anna, 2001.

[14] HEISER, G. The role of virtualization in embedded systems. In *Proc. IIES* (2008), ACM, pp. 11–16.

[15] JEFFAY, K., AND GODDARD, S. Rate-based resource allocation models for embedded systems. In *Proc. EMSOFT* (2001), Springer, pp. 204–222.

[16] KOPETZ, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[17] LIPARI, G., AND BINI, E. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing 1*, 2 (2005), 257–269.

[18] MERCER, C. W., SAVAGE, S., AND TOKUDA, H. Processor capacity reserves for multimedia operating systems. Tech. rep., Carnegie Mellon University, 1993.

[19] STADLER, H. A virtualized real-time I/O subsystem. Master's thesis, University of Salzburg, Salzburg, Austria, 2008.

[20] STANKOVIC, J. A., AND RAMAMRITHAM, K. The design of the Spring kernel. In *Proc. RTSS* (1987), IEEE, pp. 146–157.

[21] VMWARE, INC. Virtualization overview, 2006. `www.vmware.com/pdf/virtualization.pdf`.

[22] WATSON, J. Virtualbox: bits and bytes masquerading as machines. *Linux Journal 2008*, 166 (2008), 1.

[23] WIND RIVER. RTLinux. `www.windriver.com`.