OSPERT 2012

Proceedings of the 8th annual workshop on Operating Systems Platforms for Embedded Real-Time applications

Pisa, Italy
July 10, 2012

In conjunction with:
The 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)
July 10-13, 2012

Edited by Gabriel Parmer and Andrea Bastoni

©Copyright 2012 by the authors

Copyright 2012 The George Washington University.
All rights reserved. The copyright of this collection is with The George Washington University. The copyright of the individual articles remains with their authors.

Contents

Message from the Chairs

We aimed to continue the interactive emphasis for this 8th workshop on Operating Systems Platforms for Embedded Real-Time Applications. Toward this, we will have two discussion-based sessions. One is a discussion about the interface of real-time research and smartphones led by a panel of four experts: "Smartphone and Real-Time: Innovation or yet another Embedded Device?". Additionally, the conference will commence with a keynote by Paul E. McKenney who will discuss the challenges of maintaining low response times at scale. OSPERT this year accepted 5 of 7 peer reviewed papers, and we have included an invited paper on a operation system structured for predictable and reliable multi-core execution. We have also included a new category of papers this year based on forward-looking ideas, to focus on innovation in the area of real-time systems. Given the quality and controversial nature of the papers, we expect a lively OSPERT.

We'd like to thank all of the people behind the scenes that were involved in making OSPERT what it is. Gerhard Fohler, and ECRTS chairs who have made this workshop possible, and we appreciate the support and venue for the operating systems side of real-time systems. The program committee has done wonderful work in fastidiously reviewing submissions, and providing useful feedback.

Most of all, this workshop will be a success based on the community of operating systems and real-time researchers that provide the excitement and discussion that defines OSPERT.

The Workshop Chairs,
Andrea Bastoni
Gabriel Parmer

Program Committee

Björn B. Brandenburg, *Max Planck Institute for Software Systems*
Gernot Heiser, *The University of New South Wales*
Shinpei Kato, *Nagoya University*
Jim Anderson, *University of North Carolina at Chapel Hill*
Thomas Gleixner, *Linutronix, Germany*
Steven Rostedt, *Red Hat*
John Regehr, *University of Utah*
Roberto Gioiosa, *Barcelona Supercomputing Center*

Keynote Talk

Real-Time Response on Multicore Systems: It Is Bigger Than You Think

Paul E. McKenney
IBM Distinguished Engineer

Five years ago, I published a Linux Journal article entitled "SMP and Embedded Real Time" (`http://www.linuxjournal.com/article/9361`) stating that SMP systems could in fact support real-time workloads. This article was not well-received in all segments of the real-time community, but there are nevertheless quite a few SMP real-time systems in use today offering scheduling latencies of a few tens of microseconds.

So perhaps you can imagine my surprise when in early 2012 I received a bug report stating that the Linux kernel's RCU implementation was causing 200-microsecond latency spikes. The root cause? I simply hadn't been thinking big enough. This talk tells the story of the ensuing adventure.

Biography:

Paul E. McKenney has been coding for almost four decades, more than half of that on parallel hardware, where his work has earned him a reputation among some as a flaming heretic. Over the past decade, Paul has been an IBM Distinguished Engineer at the IBM Linux Technology Center, where he maintains the RCU implementation within the Linux kernel, dealing with a variety of workloads presenting highly entertaining performance, scalability, real-time response, and energy-efficiency challenges. Prior to that, he worked on the DYNIX/ptx kernel at Sequent, and prior to that on packet-radio and Internet protocols (but long before it was polite to mention Internet at cocktail parties), system administration, business applications, and real-time systems. His hobbies include what passes for running at his age along with the usual house-wife-and-kids habit.

Program

	Tuesday, July 10th 2011
8:30-9:30	Registration
9:30-11:00	Keynote Talk: *Real-Time Response on Multicore Systems: It Is Bigger Than You Think* Paul E. McKenney
11:00-11:30	Coffee Break
11:30-13:00	Session 1: Parallelism in Real-Time Systems Time Management in the Quest-V RTOS *Richard West, Ye Li, and Eric Missimer* Operating Systems for Manycore Processors from the Perspective of Safety-Critical Systems *Florian Kluge, Benot Triquet, Christine Rochange, Theo Ungerer* PRACTISE: a framework for PeRformance Analysis and Testing of real-time multIcore SchEdulers for the Linux kernel *Fabio Falzoi, Juri Lelli, Giuseppe Lipari*
13:30-14:30	Lunch
14:30-16:00	Panel Discussion: *Smartphone and Real-Time: Innovation or yet another Embedded Device?* *Panel members: Wolfgang Mauerer, Claudio Scordino, Heechul Yun, and Paul E. McKenney*
16:00-16:30	Coffee Break
16:30-18:00	Session 2: Real-Time System Potpourri CoS: A New Perspective of Operating Systems Design for the Cyber-Physical World *Vikram Gupta, Eduardo Tovar, Nuno Pereira, Ragunathan (Raj) Rajkumar* Efficient I/O Scheduling with Accurately Estimated Disk Drive Latencies *Vasily Tarasov, Gyumin Sim, Anna Povzner, Erez Zadok* A Dataflow Monitoring Service Based on Runtime Verification for AUTOSAR OS: Implementation and Performances *Sylvain Cotard, S'bastien Faucou, Jean-Luc B' chennec*
18:00-18:30	Discussion and Closing Thoughts

	Wednesday, 11th - Friday, 13th 2011
	ECRTS main proceedings.

Time Management in the Quest-V RTOS *

Richard West, Ye Li, and Eric Missimer

Computer Science Department
Boston University
Boston, MA 02215, USA
{richwest,liye,missimer}@cs.bu.edu

Abstract

Quest-V is a new system currently under development for multicore processors. It comprises a collection of separate kernels operating together as a distributed system on a chip. Each kernel is isolated from others using virtualization techniques, so that faults do not propagate throughout the entire system. This multikernel design supports online fault recovery of compromised or misbehaving services without the need for full system reboots. While the system is designed for high-confidence computing environments that require dependability, Quest-V is also designed to be predictable. It treats time as a first-class resource, requiring that all operations are properly accounted and handled in real-time. This paper focuses on the design aspects of Quest-V that relate to how time is managed. Special attention is given to how Quest-V manages time in four key areas: (1) scheduling and migration of threads and virtual CPUs, (2) I/O management, (3) communication, and (4) fault recovery.

1 Introduction

Multicore processors are becoming increasingly popular, not only in server domains, but also in real-time and embedded systems. Server-class processors such as Intel's Single-chip Cloud Computer (SCC) support 48 cores, and others from companies such as Tilera are now finding their way into real-time environments [18]. In real-time systems, multicore processors offer the opportunity to dedicate time-critical tasks to specific processors, allowing others to be used by best effort services. Alternatively, as in the case of processors such as the ARM Cortex-R7, they provide fault tolerance, ensuring functionality of software in the wake of failures of any one core.

*This work is supported in part by NSF Grant #1117025.

Quest-V is a new operating system we are developing for multicore processors. It is designed to be both dependable and predictable, providing functionality even when services executing on one or more cores become compromised or behave erroneously. Predictability even in the face of software component failures ensures that application timing requirements can be met. Together, Quest-V's dependability and predictability objectives make it suitable for the next generation of safety-critical embedded systems.

Quest-V is a virtualized multikernel, featuring multiple sandbox kernels connected via shared memory communication channels. Virtualization is used to isolate and prevent faults in one sandbox from adversely affecting other sandboxes. The resultant system maintains availability while faulty software components are replaced or re-initialized in the background. Effectively, Quest-V operates as a "distributed system on a chip", with each sandbox responsible for local scheduling and management of its own resources, including processing cores.

In Quest-V, scheduling involves the use of *virtual CPUs* (VCPUs). These differ from VCPUs in conventional hypervisor systems, which provide an abstraction of the underlying physical processors that are shared among separate guest OSes. Here, VCPUs act as resource containers [3] for scheduling and accounting the execution time of threads. VCPUs form the basis for system predictability in Quest-V. Each VCPU is associated with one or more software threads, which can be assigned to specific sandboxes according to factors such as per-core load, interrupt balancing, and processor cache usage, amongst others. In this paper, we show how VCPU scheduling and migration is performed predictably. We also show how time is managed to ensure bounded delays for inter-sandbox communication, software fault recovery and I/O management.

An overview of the Quest-V design is described in the next section. This is followed in Section 3 by a description of how Quest-V guarantees predictability in various subsystems, including VCPU scheduling and migration, I/O

management, communication and fault recovery. Finally, conclusions and future work are described in Section 4.

2 Quest-V Design

Quest-V is targeted at safety-critical applications, primarily in real-time and embedded systems where dependability is important. Target applications include those emerging in health-care, avionics, automotive systems, factory automation, robotics and space exploration. In such cases, the system requires real-time responsiveness to time-critical events, to prevent potential loss of lives or equipment. Similarly, advances in fields such as cyber-physical systems means that more sophisticated OSes beyond those traditionally found in real-time domains are now required.

The emergence of off-the-shelf and low-power processors now supporting multiple cores and hardware virtualization offer new opportunities for real-time and embedded system designers. Virtualization capabilities enable new techniques to be integrated into the design of the OS, so that software components are isolated from potential faults or security violations. Similarly, added cores offer fault tolerance through redundancy, while allowing time-critical tasks to run in parallel when necessary. While the combination of multiple cores and hardware virtualization are features currently found on more advanced desktop and server-class processors, it is to be anticipated that such features will appear on embedded processors in the near future. For example, the ARM Cortex A15 processor is expected to feature virtualization capabilities, offering new possibilities in the design of operating systems.

Quest-V takes the view that virtualization features should be integral to the design of the OS, rather than providing capabilities to design hypervisors for hosting separate unrelated guest OSes. While virtualization provides the basis for safe isolation of system components, proper time management is necessary for real-time guarantees to be met. Multicore processors pose challenges to system predictability, due to the presence of shared on-chip caches, memory bus bandwidth contention, and in some cases non-uniform memory access (NUMA). These micro-architectural factors must be addressed in the design of the system. Fortunately, hardware performance counters are available, to help deduce micro-architectural resource usage. Quest-V features a performance monitoring subsystem to help improve schedulability of threads and VCPUs, reducing worst-case execution times and allowing higher workloads to be admitted into the system.

2.1 System Architecture

Figure 1 shows an overview of the Quest-V architecture. One sandbox is mapped to a separate core of a multicore processor, although in general it is possible to map sandboxes to more than one core [1]. This is similar to how Corey partitions resources amongst applications [7]. In our current approach, we assume each sandbox kernel is associated with one physical core since that simplifies local (sandbox) scheduling and allows for relatively easy enforcement of service guarantees using a variant of rate-monotonic scheduling [12]. Notwithstanding, application threads can be migrated between sandboxes as part of a load balancing strategy, or to allow parallel thread execution.

A single hypervisor is replaced by a separate monitor for each sandbox kernel. This avoids the need to switch page table mappings within a single global monitor when accessing sandbox (guest) kernel addresses. We assume monitors are trusted but failure of one does not necessarily mean the system is compromised since one or more other monitors may remain fully operational. Additionally, the monitors are expected to only be used for exceptional conditions, such as updating shared memory mappings for inter-sandbox communication [11] and initiating fault recovery.

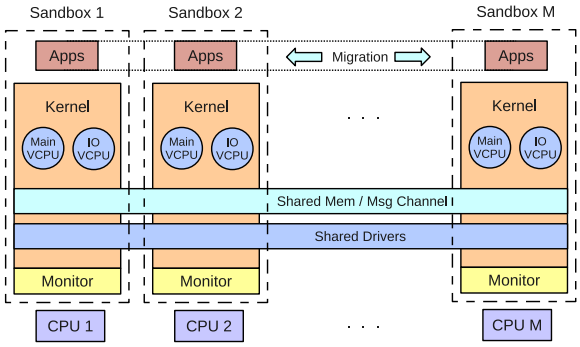


Figure 1. Quest-V Architecture Overview

Quest-V currently runs as a 32-bit system on x86 platforms with hardware virtualization support (e.g., Intel VT-x or AMD-V processors). Memory virtualization is used as an integral design feature, to separate sub-system components into distinct sandboxes. Further details can be found in our complementary paper that focuses more extensively on the performance of the Quest-V design [11]. Figure 2 shows the mapping of sandbox memory spaces to physical memory. Extended page table (EPT [2]) structures combine with conventional page tables to map sandbox (guest) virtual addresses to host physical values. Only monitors can change EPT memory mappings, ensuring software faults or security violations in one sandbox cannot corrupt the memory of another sandbox.

[1] Unless otherwise stated, we make no distinction between a processing core or hardware thread.

[2] Intel processors with VT-x technology support extended page tables, while AMD-V processors have similar support for nested page tables. For consistency we use the term EPT in this paper.

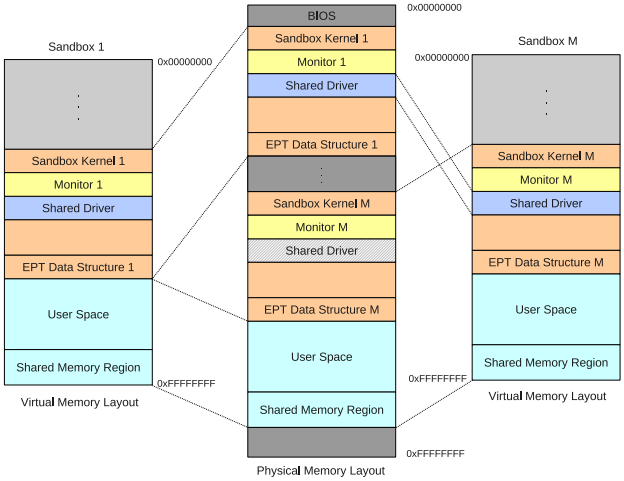


Figure 2. Quest-V Memory Layout

The Quest-V architecture supports sandbox kernels that have both replicated and complementary services. That is, some sandboxes may have identical kernel functionality, while others partition various system components to form an asymmetric configuration. The extent to which functionality is separated across kernels is somewhat configurable in the Quest-V design. In our initial implementation, each sandbox kernel replicates most functionality, offering a private version of the corresponding services to its local application threads. Certain functionality is, however, shared across system components. In particular, we share certain driver data structures across sandboxes [3], to allow I/O requests and responses to be handled locally.

Quest-V allows *any* sandbox to be configured for corresponding device interrupts, rather than have a dedicated sandbox be responsible for all communication with that device. This greatly reduces the communication and control paths necessary for I/O requests from applications in Quest-V. It also differs from the split-driver approach taken by systems such as Xen [4], that require all device interrupts to be channeled through a special driver domain.

Sandboxes that do not require access to shared devices are isolated from unnecessary drivers and associated services. Moreover, a sandbox can be provided with its own private set of devices and drivers, so if a software failure occurs in one driver, it will not necessarily affect all other sandboxes. In fact, if a driver experiences a fault then its effects are limited to the local sandbox and the data structures shared with other sandboxes. Outside these shared data structures, remote sandboxes (including all monitors) are protected by EPTs.

Application and system services in distinct sandbox kernels communicate via shared memory channels. These channels are established by extended page table mappings setup by the corresponding monitors. Messages are passed across these channels similar to the approach in Barrelfish [5].

Main and I/O VCPUs are used for real-time management of CPU cycles, to enforce *temporal isolation*. Application and system threads are bound to VCPUs, which in turn are assigned to underlying physical CPUs. We will elaborate on this aspect of the system in the following section.

2.2 VCPU Management

In Quest-V, *virtual CPUs* (VCPUs) form the fundamental abstraction for scheduling and temporal isolation of the system. Here, temporal isolation means that each VCPU is guaranteed its share of CPU cycles without interference from other VCPUs.

The concept of a VCPU is similar to that in virtual machines [2, 4], where a hypervisor provides the illusion of multiple *physical CPUs* (PCPUs) [4] represented as VCPUs to each of the guest virtual machines. VCPUs exist as kernel abstractions to simplify the management of resource budgets for potentially many software threads. We use a hierarchical approach in which VCPUs are scheduled on PCPUs and threads are scheduled on VCPUs.

A VCPU acts as a resource container [3] for scheduling and accounting decisions on behalf of software threads. It serves no other purpose to virtualize the underlying physical CPUs, since our sandbox kernels and their applications execute directly on the hardware. In particular, a VCPU does not need to act as a container for cached instruction blocks that have been generated to emulate the effects of guest code, as in some trap-and-emulate virtualized systems.

In common with bandwidth preserving servers [1, 9, 14], each VCPU, V, has a maximum compute time budget, C_V, available in a time period, T_V. V is constrained to use no more than the fraction $U_V = \frac{C_V}{T_V}$ of a physical processor (PCPU) in any window of real-time, T_V, while running at its normal (foreground) priority. To avoid situations where PCPUs are idle when there are threads awaiting service, a VCPU that has expired its budget may operate at a lower (background) priority. All background priorities are set below those of foreground priorities to ensure VCPUs with expired budgets do not adversely affect those with available budgets.

Quest-V defines two classes of VCPUs: (1) *Main VCPUs* are used to schedule and track the PCPU usage of conventional software threads, while (2) *I/O VCPUs* are used to account for, and schedule the execution of, interrupt handlers for I/O devices. This distinction allows for interrupts

[3] Only for those drivers that have been mapped as shared between specific sandboxes.

[4] We define a PCPU to be either a conventional CPU, a processing core, or a hardware thread in a simultaneous multi-threaded (SMT) system.

from I/O devices to be scheduled as threads [17], which may be deferred execution when threads associated with higher priority VCPUs having available budgets are runnable. The flexibility of Quest-V allows I/O VCPUs to be specified for certain devices, or for certain tasks that issue I/O requests, thereby allowing interrupts to be handled at different priorities and with different CPU shares than conventional tasks associated with Main VCPUs.

2.2.1 VCPU API

VCPUs form the basis for managing time as a first-class resource: VCPUs are specified time bounds for the execution of corresponding threads. Stated another way, every executable control path in Quest-V is mapped to a VCPU that controls scheduling and time accounting for that path. The basic API for VCPU management is described below. It is assumed this interface is managed only by a user with special privileges.

- *int vcpu_create(struct vcpu_param *param)* – Creates and initializes a new Main or I/O VCPU. The function returns an identifier for later reference to the new VCPU. If the `param` argument is `NULL` the VCPU assumes its default parameters. For now, this is a Main VCPU using a `SCHED_SPORADIC` policy [15, 13]. The `param` argument points to a structure that is initialized with the following fields:

  ```
  struct vcpu_param {
     int vcpuid; // Identifier
     int policy; // SCHED_SPORADIC or SCHED_PIBS
     int mask; // PCPU affinity bit-mask
     int C; // Budget capacity
     int T; // Period
  }
  ```

  The `policy` is `SCHED_SPORADIC` for Main VCPUs and `SCHED_PIBS` for I/O VCPUs. `SCHED_PIBS` is a *priority-inheritance bandwidth-preserving* policy that is described further in Section 3.1. The `mask` is a bit-wise collection of processing cores available to the VCPU. It restricts the cores on which the VCPU can be assigned and to which the VCPU can be later migrated. The remaining VCPU parameters control the budget and period of a sporadic server, or the equivalent bandwidth utilization for a PIBS server. In the latter case, the ratio of C and T is all that matters, not their individual values.

  On success, a `vcpuid` is returned for a new VCPU. An admission controller must check that the addition of the new VCPU meets system schedulability requirements, otherwise the VCPU is not created and an error is returned.
- *int vcpu_destroy (int vcpuid, int force)* – Destroys and cleans up state associated with a VCPU. The count of the number of threads associated with a VCPU must be 0 if the `force` flag is not set. Otherwise, destruction of the VCPU will force all associated threads to be terminated.
- *int vcpu_setparam (struct vcpu_param *param)* – Sets the parameters of the specified VCPU referred to by `param`. This allows an existing VCPU to have new parameters from those when it was first created.
- *int vcpu_getparam (struct vcpu_param *param)* – Gets the VCPU parameters for the next VCPU in a list for the caller's process. That is, each process has associated with it one or more VCPUs, since it also has at least one thread. Initially, this call returns the VCPU parameters at the head of a list of VCPUs for the calling thread's process. A subsequent call returns the parameters for the next VCPU in the list. The current position in this list is maintained on a per-thread basis. Once the list-end is reached, a further call accesses the head of the list once again.
- *int vcpu_bind_task (int vcpuid)* – Binds the calling task, or thread, to a VCPU specified by `vcpuid`.

Functions *vcpu_destroy*, *vcpu_setparam*, *vcpu_getparam* and *vcpu_bind_task* all return 0 on success, or an error value.

2.2.2 Parallelism in Quest-V

At system initialization time, Quest-V launches one or more sandbox kernels. Each sandbox is then assigned a partitioning of resources, in terms of host physical memory, available I/O devices, and PCPUs. The default configuration creates one sandbox per PCPU. As stated earlier, this simplifies scheduling decisions within each sandbox. Sandboxing also reduces the extent to which synchronization is needed, as threads in separate sandboxes typically access private resources. For parallelism of multi-threaded applications, a single sandbox must be configured to manage more than one PCPU, or a method is needed to distribute application threads across multiple sandboxes.

Quest-V maintains a `quest_tss` data structure for each software thread. Every address space has at least one `quest_tss` data structure. Managing multiple threads within a sandbox is similar to managing processes in conventional system designs. The only difference is that Quest-V requires every thread to be associated with a VCPU and the corresponding sandbox kernel (without involvement of its monitor) schedules VCPUs on PCPUs.

In some cases it might be necessary to assign threads of a multi-threaded application to separate sandboxes. This could be for fault isolation reasons, or for situations where one sandbox has access to resources, such as devices, not available in other sandboxes. Similarly, threads may need to be redistributed as part of a load balancing strategy.

In Quest-V, threads in different sandboxes are mapped to separate host physical memory ranges, unless they ex-

ist in shared memory regions established between sandboxes. Rather than confining threads of the same application to shared memory regions, Quest-V defaults to using separate process address spaces for threads in different sandboxes. This increases the isolation between application threads in different sandboxes, but requires special communication channels to allow threads to exchange information.

Here, we describe how a multi-threaded application is established across more than one sandbox.

STEP 1: Create a new VCPU in parent process – Quest-V implements process address spaces using `fork/exec/exit` calls, similar to those in conventional UNIX systems. A child process, initially having one thread, inherits a *copy* of the address space and corresponding resources defined in the parent thread's `quest_tss` data structure. Forked threads differ from forked processes in that no new address space copy is made. A parent calling `fork` first establishes a new VCPU for use by the child. In all likelihood the parent will know the child's VCPU parameter requirements, but they can later be changed in the child using `vcpu_setparam`.

If the admission controller allows the new VCPU to be created, it will be established in the local sandbox. If the VCPU cannot be created locally, the PCPU affinity mask can be used to identify a remote sandbox for the VCPU. Remote sandboxes can be contacted via shared memory communication channels, to see which one, if any, is best suited for the VCPU. If shared channels do not exist, monitors can be used to send IPIs to other sandboxes. Remote sandboxes can then respond with *bids* to determine the best target. Alternatively, remote sandboxes can advertise their willingness to accept new loads by posting information relating to their current load in shared memory regions accessible to other sandboxes. This latter strategy is an *offer* to accept remote requests, and is made without waiting for *bid requests* from other sandboxes.

STEP 2: Fork a new thread or process and specify the VCPU – A parent process can now make a special `fork` call, which takes as an argument the `vcpuid` of the VCPU to be used for scheduling and resource accounting. The request can originate from a different sandbox to the one where the VCPU is located, so some means of global resolution of VCPU IDs is needed.

STEP 3: Set VCPU parameters in new thread/process – A thread or process can adjust the parameters of any VCPUs associated with it, using `vcpu_setparam`. This includes updates to its utilization requirements, and also the affinity mask. Changes to the affinity might require the VCPU and its associated process to migrate to a remote sandbox.

The steps described above can be repeated as necessary to create a series of threads, processes and VCPUs within or across multiple sandboxes. As stated in STEP 3, it might be necessary to migrate a VCPU and its associated address space to a remote sandbox. The initial design of Quest-V limits migration of Main VCPUs and associated address spaces. We assume I/O VCPUs are statically mapped to sandboxes responsible for dedicated devices.

The details of how migration is performed are described in Section 3.1. The rationale for only allowing Main VCPUs to migrate is because we can constrain their usage to threads within a single process address space. Specifically, a Main VCPU is associated with one or more threads, but every such thread is within the *same* process address space. However, two separate threads bound to different VCPUs can be part of the same or different address space. This makes VCPU migration simpler since we only have to copy the memory for one address space. It also means that within a process the system maintains a list of VCPUs that can be bound to threads within the corresponding address space. As I/O VCPUs can be associated with multiple different address spaces, their migration would require the migration, and hence copying, of potentially multiple address spaces between sandboxes. For predictability reasons, we can place an upper bound on the time to copy one address space between sandboxes, as opposed to an arbitrary number. Also, migrating I/O VCPUs requires association of devices, and their interrupts, with different sandboxes. This can require intervention of monitors to update I/O APIC interrupt distribution settings.

3 System Predictability

Quest-V uses VCPUs as the basis for time management and predictability of its sub-systems. Here, we describe how time is managed in four key areas of Quest-V: (1) scheduling and migration of threads and virtual CPUs, (2) I/O management, (3) inter-sandbox communication, and (4) fault recovery.

3.1 VCPU Scheduling and Migration

By default, VCPUs act like Sporadic Servers [13]. Sporadic Servers enable a system to be treated as a collection of equivalent periodic tasks scheduled by a rate-monotonic scheduler (RMS) [12]. This is significant, given I/O events can occur at arbitrary (aperiodic) times, potentially triggering the wakeup of blocked tasks (again, at arbitrary times) having higher priority than those currently running. RMS analysis can therefore be applied, to ensure each VCPU is guaranteed its share of CPU time, U_V, in finite windows of real-time.

Scheduling Example. An example schedule is provided in

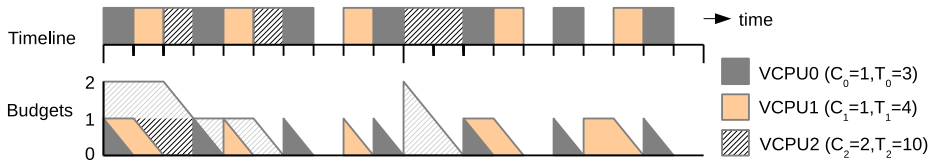


Figure 3. Example VCPU Schedule

Figure 3 for three Main VCPUs, whose budgets are depleted when a corresponding thread is executed. Priorities are inversely proportional to periods. As can be seen, each VCPU is granted its real-time share of the underlying PCPU.

In Quest-V there is no notion of a periodic timer interrupt for updating system clock time. Instead, the system is event driven, using per-processing core local APIC timers to replenish VCPU budgets as they are consumed during thread execution. We use the algorithm proposed by Stanovich et al [15] to correct for early replenishment and budget amplification in the POSIX specification.

Main and I/O VCPU Scheduling. Figure 4 shows an example schedule for two Main VCPUs and one I/O VCPU for a certain device such as a gigabit Ethernet card. In this example, Schedule (A) avoids premature replenishments, while Schedule (B) is implemented according to the POSIX specification. In (B), VCPU1 is scheduled at $t = 0$, only to be preempted by higher priority VCPU0 at $t = 1, 41, 81$, etc. By $t = 28$, VCPU1 has amassed a total of 18 units of execution time and then blocks until $t = 40$. Similarly, VCPU1 blocks in the interval $[t = 68, 80]$. By $t = 68$, Schedule (B) combines the service time chunks for VCPU1 in the intervals $[t = 0, 28]$ and $[t = 40, 68]$ to post future replenishments of 18 units at $t = 50$ and $t = 90$, respectively. This means that over the first 100 time units, VCPU1 actually receives 46 time units, when it should be limited to 40%. Schedule (A) ensures that over the same 100 time units, VCPU1 is limited to the correct amount. The problem is triggered by the blocking delays of VCPU1. Schedule (A) ensures that when a VCPU blocks (e.g., on an I/O operation), on resumption of execution it effectively starts a new replenishment phase. Hence, although VCPU1 actually receives 21 time units in the interval $[t = 50, 100]$ it never exceeds more than its 40% share of CPU time between blocking periods and over the first 100 time units it meets its bandwidth limit.

For completeness, Schedule (A) shows the list of replenishments and how they are updated at specific times, according to scheduling events in Quest-V. The invariant is that the sum of replenishment amounts for all list items must not exceed the budget capacity of the corresponding VCPU (here, 20, for VCPU1). Also, no future replenishment, R, for a VCPU, V, executing from t to $t + R$ can occur before $t + T_V$.

When VCPU1 first blocks at $t = 28$ it still has 2 units of budget remaining, with a further 18 due for replenishment at $t = 50$. At this point, the schedule shows the execution of the I/O VCPU for 2 time units. In Quest-V, threads running on Main VCPUs block (causing the VCPU to block if there are no more runnable threads), while waiting for I/O requests to complete. All I/O operations in response to device interrupts are handled as threads on specific I/O VCPUs. Each I/O VCPU supports threaded interrupt handling at a priority inherited from the Main VCPU associated with the blocked thread. In this example, the I/O VCPU runs at the priority of VCPU1. The I/O VCPU's budget capacity is calculated as the product of it bandwidth specification (here, $U_{IO} = 4\%$) and the period, T_V, of the corresponding Main VCPU for which it is performing service. Hence, the I/O VCPU receives a budget of $U_{IO} \cdot T_V = 2$ time units, and through bandwidth preservation, will be eligible to execute again at $t_e = t + C_{actual}/U_{IO}$, where t is the start time of the I/O VCPU and $C_{actual} \mid 0 \leq C_{actual} \leq U_{IO} \cdot T_V$ is how much of its budget capacity it really used.

In Schedule (A), VCPU1 resumes execution after unblocking at times, $t = 40$ and 80. In the first case, the I/O VCPU has already completed the I/O request for VCPU1 but some other delay, such as accessing a shared resource guarded by a semaphore (not shown) could be the cause of the added delay. Time $t = 78$ marks the next eligible time for the I/O VCPU after it services the blocked VCPU1, which can then immediately resume. Further details about VCPU scheduling in Quest-V can be found in our accompanying paper for Quest [8], a non-virtualized version of the system that does not support sandboxed service isolation.

Since each sandbox kernel in Quest-V supports local scheduling of its allocated resources, there is no notion of a global scheduling queue. Forked threads are by default managed in the local sandbox but can ultimately be migrated to remote sandboxes along with their VCPUs, according to load constraints or affinity settings of the target VCPU. Although each sandbox is isolated in a special guest execution domain controlled by a corresponding monitor, the monitor is not needed for scheduling purposes. This avoids costly virtual machine exits and re-entries (i.e., VM-Exits and VM-resumes) as would occur with hypervisors such as Xen [4] that manage multiple separate guest OSes.

Temporal Isolation. Quest-V provides temporal isolation

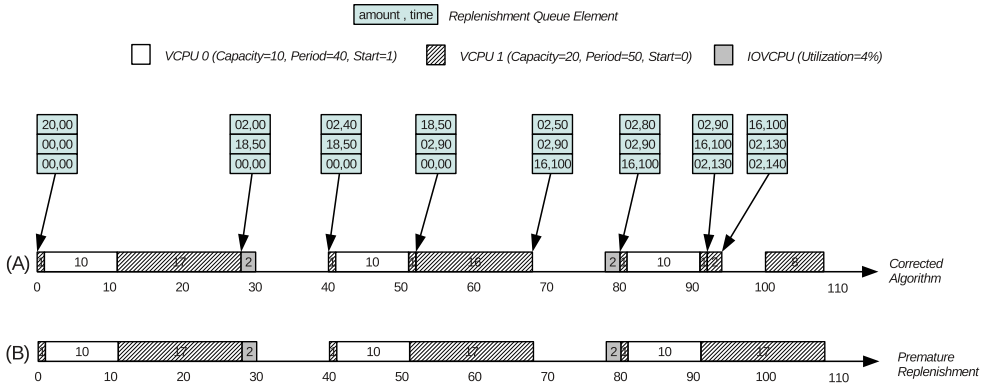


Figure 4. Sporadic Server Replenishment List Management

of VCPUs assuming the total utilization of a set of Main and I/O VCPUs within each sandbox do not exceed specific limits. Each sandbox can determine the schedulability of its local VCPUs independently of all other sandboxes. For cases where a sandbox is associated with one PCPU, n Main VCPUs and m I/O VCPUs we have the following:

$$\sum_{i=0}^{n-1} \frac{C_i}{T_i} + \sum_{j=0}^{m-1} (2 - U_j) \cdot U_j \leq n \left(\sqrt[n]{2} - 1 \right)$$

Here, C_i and T_i are the budget capacity and period of Main VCPU, V_i. U_j is the utilization factor of I/O VCPU, V_j [8].

VCPU and Thread Migration. For multicore processors, the cores share a last-level cache (LLC) whose lines are occupied by software thread state from any of the sandbox kernels. It is therefore possible for one thread to suffer poorer progress than another, due to cache line evictions from conflicts with other threads. Studies have shown memory bus transfers can incur several hundred clock cycles, or more, to fetch lines of data from memory on cache misses [7, 10]. While this overhead is often hidden by prefetch logic, it is not always possible to prevent memory bus stalls due to cache contention from other threads.

To improve the global performance of VCPU scheduling in Quest-V, VCPUs and their associated threads can be migrated between sandbox kernels. This helps prevent coschedules involving multiple concurrent threads with high memory activity (that is, large working sets or frequent accesses to memory). Similarly, a VCPU and its corresponding thread(s) might be better located in another sandbox that is granted direct access to an I/O device, rather than having to make inter-sandbox communication requests for I/O. Finally, on non-uniform memory access (NUMA) architectures, threads and VCPUs should be located close to the memory domains that best serve their needs without having to issue numerous transactions across an interconnect between chip sockets [6].

In a real-time system, migrating threads (and, in our case, VCPUs) between processors at runtime can impact the schedulability of local schedules. Candidate VCPUs for migration are determined by factors such as memory activity of the threads they support. We use hardware performance counters found on modern multicore processors to measure events such as per-core and per-chip package *cache misses*, *cache hits*, *instructions retired* and *elapsed cycles* between scheduling points.

For a single chip package, or socket, we distribute threads and their VCPUs amongst sandboxes to: (a) balance total VCPU load, and (b) balance per-sandbox LLC miss-rates or aggregate cycles-per-instruction (CPI) for all corresponding threads. For NUMA platforms, we are considering cache occupancy prediction techniques [16] to estimate the costs of migrating thread working sets between sandboxes on separate sockets.

Predictable Migration Strategy. We are considering two approaches to migration. In both cases, we assume that a VCPU and its threads are associated with *one* address space, otherwise multiple address spaces would have to be moved between sandboxes, which adds significant overhead.

The first migration approach uses shared memory to copy an address space and its associated `quest_tss` data structure(s) from the source sandbox to the destination. This allows sandbox kernels to perform the migration without involvement of monitors, which would require VM-Exit and VM-resume operations. These are potentially costly operations, of several hundred clock cycles [11]. This approach only requires monitors to establish shared memory mappings between a pair of sandboxes, by updating extended page tables as necessary. However, for address spaces that are larger than the shared memory channel we effectively have to perform a UNIX *pipe*-style exchange of information between sandboxes. This leads to a synchronous exchange, with the source sandbox blocking when the shared channel is full, and the destination blocking when awaiting

more information in the channel.

In the second migration approach, we can eliminate the need to copy address spaces both *into* and *out of* shared memory. Instead, the destination sandbox is asked to move the migrating address space directly from the source sandbox, thereby requiring only one copy. However, the migrating address space and its `quest_tss` data structure(s) are initially located in the source sandbox's private memory. Hence, a VM-Exit into the source monitor is needed, to send an inter-processor interrupt (IPI) to the destination sandbox. This event is received by a remote migration thread that traps into its monitor, which can then access the source sandbox's private memory.

The IPI handler causes the destination monitor to temporarily map the migrating address space into the target sandbox. Then, the migrating address space can be copied to private memory in the destination. Once this is complete, the destination monitor can unmap the pages of the migrating address space, thereby ensuring sandbox memory isolation except where shared memory channels should exist. At this point, all locally scheduled threads can resume as normal. Figure 5 shows the general migration strategy. Note that for address spaces with multiple threads we still have to migrate multiple `quest_tss` structures, but a bound on per-process threads can be enforced.

Migration Threads. We are considering both migration strategies, using special *migration threads* to move address spaces and their VCPUs in bounded time. A migration thread in the destination sandbox has a Main VCPU with parameters C_m and T_m. The migrating address space associated with a VCPU, V_{src}, having parameters C_{src} and T_{src} should ideally be moved without affecting its PCPU share. To ensure this is true, we require the migration cost, $\Delta_{m,src}$, of copying an address space and its `quest_tss` data structure(s) to be less than or equal to C_m. T_m should ideally be set to guarantee the migration thread runs at highest priority in the destination. To ease migration analysis, it is preferable to move VCPUs with full budgets. For any VCPU with maximum tolerable delay, $T_{src} - C_{src}$, before it needs to be executed again, we require preemptions by higher priority VCPUs in the destination sandbox to be less than this value. In practice, V_{src} might have a tolerable delay lower than $T_{src} - C_{src}$. This restricts the choice of migratable VCPUs and address spaces, as well as the destination sandboxes able to accept them. Further investigation is needed to determine the schedulability of migrating VCPUs and their address spaces.

3.2 Predictable I/O Management

As shown in Section 3.1, Quest-V assigns I/O VCPUs to interrupt handling threads. Only a minimal "top half" [17]

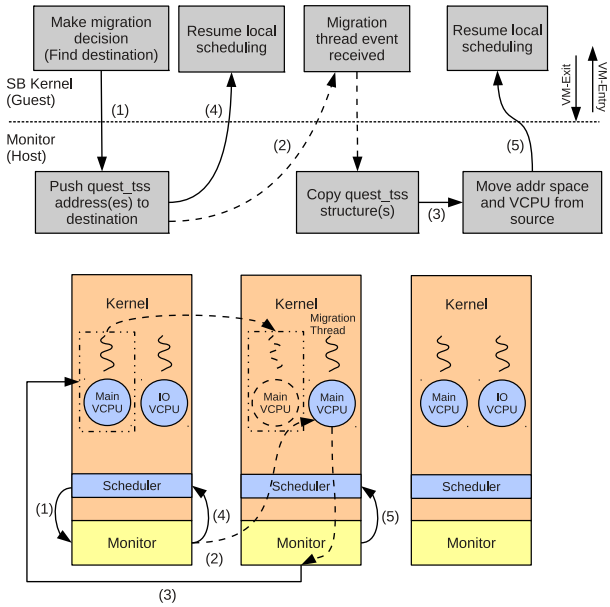


Figure 5. Time-Bounded Migration Strategy

part of interrupt processing is needed to acknowledge the interrupt and post an event to handle the subsequent "bottom half" in a thread bound to an I/O VCPU. A worst-case bound can be placed on top half processing, which is charged to the current VCPU as system overhead.

Interrupt processing as part of device I/O requires proper prioritization. In Quest-V, this is addressed by assigning an I/O VCPU the priority of the Main VCPU on behalf of which interrupt processing is being performed. Since all VCPUs are bandwidth preserving, we set the priority of an I/O VCPU to be inversely proportional to the period of its corresponding Main CPU. This is the essence of priority-inheritance bandwidth preservation scheduling (PIBS). Quest-V ensures that the priorities of all I/O operations are correctly matched with threads running on Main VCPUs, although such threads may block on their Main VCPUs while interrupt processing occurs. To ensure I/O processing is bandwidth-limited, each I/O VCPU is assigned a specific percentage of PCPU time. Essentially, a PIBS-based I/O VCPU operates like a Sporadic Server with *one* dynamically-calculated replenishment.

This approach to I/O management prevents live-lock and priority inversion, while integrating the management of interrupts with conventional thread execution. It does, however, require correctly matching interrupts with Main VCPU threads. To do this, Quest-V's drivers support *early demultiplexing* to identify the thread for which the interrupt has occurred. This overhead is also part of the top half cost described above.

Finally, Quest-V programs I/O APICs to multicast de-

vice interrupts to the cores of sandboxes with access to those devices. In this way, interrupts are not always directed to one core which becomes an I/O server for all others. Multicast interrupts are filtered as necessary, as part of early demultiplexing, to decide whether or not subsequent I/O processing should continue in the target sandbox.

3.3 Inter-Sandbox Communication

Inter-sandbox communication in Quest-V relies on message passing primitives built on shared memory, and asynchronous event notification mechanisms using Interprocessor Interrupts (IPIs). IPIs are currently used to communicate with remote sandboxes to assist in fault recovery, and can also be used to notify the arrival of messages exchanged via shared memory channels. Monitors update shadow page table mappings as necessary to establish message passing channels between specific sandboxes. Only those sandboxes with mapped shared pages are able to communicate with one another. All other sandboxes are isolated from these memory regions.

A *mailbox* data structure is set up within shared memory by each end of a communication channel. By default, Quest-V currently supports asynchronous communication by polling a status bit in each relevant mailbox to determine message arrival. Message passing threads are bound to VCPUs with specific parameters to control the rate of exchange of information. Likewise, sending and receiving threads are assigned to higher priority VCPUs to reduce the latency of transfer of information across a communication channel. This way, shared memory channels can be prioritized and granted higher or lower throughput as needed, while ensuring information is communicated in a predictable manner. Thus, Quest-V supports real-time communication between sandboxes without compromising the CPU shares allocated to non-communicating tasks.

3.4 Predictable Fault Recovery

Central to the Quest-V design is fault isolation and recovery. Hardware virtualization is used to isolate sandboxes from one another, with monitors responsible for mapping sandbox virtual address spaces onto (host) physical regions.

Quest-V supports both local and remote fault recovery. Local fault recovery attempts to restore a software component failure without involvement of another sandbox. The local monitor re-initializes the state of one or more compromised components, as necessary. The recovery procedure itself requires some means of fault detection and trap (VM-Exit) to the monitor, which we assume is never compromised. Remote fault recovery makes sense when a replacement software component already exists in another sandbox, and it is possible to use that functionality while the local sandbox is recovered in the background. This strategy avoids the delay of local recovery, allowing service to be continued remotely. We assume in all cases that execution of a faulty software component can correctly resume from a recovered state, which might be a re-initialized state or one restored to a recent checkpoint. For checkpointing, we require monitors to periodically intervene using a preemption timeout mechanism so they can checkpoint the state of sandboxes into private memory.

Here, we are interested in the predictability of fault recovery and assume the procedure for correctly identifying faults, along with the restoration of suitable state already exists. These aspects of fault recovery are, themselves, challenging problems outside the scope of this paper.

In Quest-V, predictable fault recovery requires the use of *recovery threads* bound to Main VCPUs, which limit the time to restore service while avoiding temporal interference with correctly functioning components and their VCPUs. Although recovery threads exists within sandbox kernels the recovery procedure operates at the monitor-level. This ensures fault recovery can be scheduled and managed just like any other thread, while accessing specially trusted monitor code. A recovery thread traps into its local monitor and guarantees that it can be de-scheduled when necessary. This is done by allowing local APIC timer interrupts to be delivered to a monitor handler just as they normally would be delivered to the event scheduler in a sandbox kernel, outside the monitor. Should a VCPU for a recovery thread expire its budget, a timeout event must be triggered to force the monitor to upcall the sandbox scheduler. This procedure requires that wherever recovery takes place, the corresponding sandbox kernel scheduler is not compromised. This is one of the factors that influences the decision to perform local or remote fault recovery.

When a recovery thread traps into its monitor, VM-Exit information is examined to determine the cause of the exit. If the monitor suspects it has been activated by a fault we need to initialize or continue the recovery steps. Because recovery can only take place while the sandbox recovery thread has available VCPU budget, the monitor must be preemptible. However, VM-Exits trap into a specific monitor entry point rather than where a recovery procedure was last executing if it had to be preempted. To resolve this issue, monitor preemptions must checkpoint the execution state so that it can be restored on later resumption of the monitor-level fault recovery procedure. Specifically, the common entry point into a monitor for all VM-Exits first examines the reason for the exit. For a fault recovery, the exit handler will attempt to restore checkpointed state if it exists from a prior preempted fault recovery stage. This is all assuming that recovery cannot be completed within one period (and budget) of the recovery thread's VCPU. Figure 6 shows how the fault recovery steps are managed predictably.

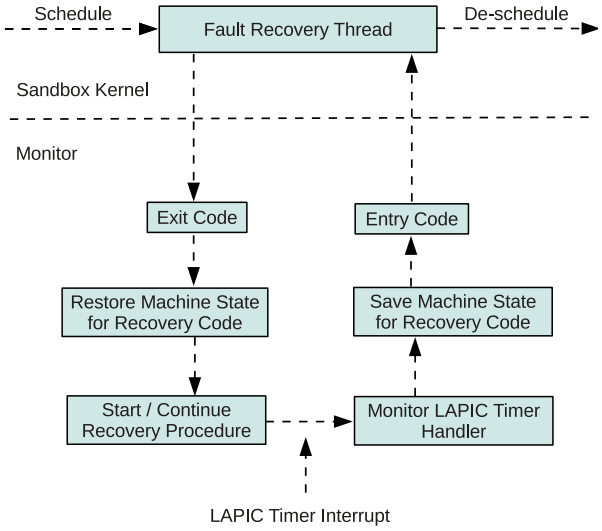


Figure 6. Time-Bounded Fault Recovery

4 Conclusions and Future Work

This paper describes time management in the Quest-V real-time multikernel. We show through the use of virtual CPUs with specific time budgets how several key subsystem components behave predictably. These sub-system components relate to on-line fault recovery, communication, I/O management, scheduling and migration of execution state.

Quest-V is being built from scratch for multicore processors with hardware virtualization capabilities, to isolate sandbox kernels and their application threads. Although Intel VT-x and AMD-V processors are current candidates for Quest-V, we expect the system design to be applicable to future embedded architectures such as the ARM Cortex A15. Future work will investigate fault detection schemes, policies to identify candidate sandboxes for fault recovery, VCPU and thread migration, and also load balancing strategies on NUMA platforms.

References

[1] L. Abeni, G. Buttazzo, S. Superiore, and S. Anna. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 4–13, 1998.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, New York, NY, USA, 2006.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 29–44, 2009.

[6] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore processors. In *USENIX Annual Technical Conference*, 2011.

[7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. hua Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57, 2008.

[8] M. Danish, Y. Li, and R. West. Virtual-CPU Scheduling in the Quest Operating System. In *the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011.

[9] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, 1997.

[10] U. Drepper. *What Every Programmer Should Know About Memory*. Redhat, Inc., November 21 2007.

[11] Y. Li, M. Danish, and R. West. Quest-V: A virtualized multikernel for high-confidence systems. Technical Report 2011-029, Boston University, December 2011.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[13] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, 1989.

[14] M. Spuri, G. Buttazzo, and S. S. S. Anna. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10:179–210, 1996.

[15] M. Stanovich, T. P. Baker, A.-I. Wang, and M. G. Harbour. Defects of the POSIX sporadic server and how to correct them. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[16] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *Operating Systems Review*, 44(4), December 2010. Special VMware Track.

[17] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *the 27th IEEE Real-Time Systems Symposium*, December 2006.

[18] C. Zimmer and F. Mueller. Low contention mapping of real-time tasks onto a TilePro 64 core processor. In *the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2012.

Operating Systems for Manycore Processors from the Perspective of Safety-Critical Systems

Florian Kluge*, Benoît Triquet†, Christine Rochange‡, Theo Ungerer*
* Department of Computer Science, University of Augsburg, Germany
† Airbus Operations S.A.S., Toulouse, France
‡ IRIT, University of Toulouse, France

***Abstract*—Processor technology is advancing from bus-based multicores to network-on-chip-based manycores, posing new challenges for operating system design. While not yet an issue today, in this forward-looking paper we discuss why future safety-critical systems could profit from such new architectures. We show, how today's approaches on manycore operating systems must be extended to fulfill also the requirements of safety-critical systems.**

I. Introduction

Multicore processors have been around for many years. Their use is widely spread in high-performance and desktop computing. In the domain of real-time embedded systems (RTES) the multicore era has just started in the last years. Despite initial reservations against multicore processors for safety-critical systems [1], considerable work to overcome these by appropriate hardware [2]–[5] and software design [6], [7] exists. Today, several multicore- and real-time-capable operating systems are commercially available, e.g. the Sysgo PikeOS [8] or VxWorks [9] from Wind River.

Meanwhile processor technology is progressing, and the number of cores in single chips is increasing. Today several processors are already available that integrate over 32 cores on one die [10], [11]. The single cores in such processors are no longer connected by a shared bus, but by a *Network on Chip (NoC)* for passing messages between the cores of such a *manycore processor*. Recent work shows that even on such platforms a deterministic execution could be made possible [12]. Such manycore architectures pose new challenges not only for application developers, but also for OS design. There is considerable work on operating systems for future manycore processors [13]–[16], but targeting the domains of general-purpose, high performance and cloud computing.

In this paper we present our considerations about a system architecture for future safety-critical embedded systems that will be built from manycore processors. We show the advantages of manycore processors over today's multicore processors concerning application development and certification. We also show what problems will arise by using manycores and introduce ideas how these problems can be overcome.

In section II we present the requirements of avionic computers and show potential impacts by multi- and manycore processor deployment. In section III we discuss existing operating systems for manycore processors. The findings flow into the manycore operating system architecture that is presented in section IV. In section V we show the research challenges we see emerging from the presented manycore operating system. Section VII concludes the paper with a brief summary.

II. Avionic Computers

A. Requirements

Today, avionic computer systems are developed following the *Integrated Modular Avionics (IMA)* architecture. Some IMA software requirements are stated in the ARINC 653 standard. The objective is to run applications of different criticality levels on the same computing module *(mixed criticality)*. This raises the problem of certification. Avionic industry uses the concept of *incremental qualification*, where each component of a system is certified for itself prior to certification of the system as a whole. This can only succeed, if unpredictable interferences between the components can be excluded. ARINC 653 defines the concept of *partitioning* to isolate applications from each other. Each application is assigned its own partition. Communication between partitions is restricted to messages that are sent through interfaces provided by the OS. Thus, *freedom of interference* between applications is guaranteed, i.e. timely interferences and error propagation over partition boundaries are prevented. ARINC 653 requires that an avionic computer can execute up to 32 partitions. Partitions are scheduled in a time-slicing manner. Concerning shared I/O, the underlying OS or hypervisor must help to ensure isolation. Applications consist of multiple processes that can interact through a number of ARINC 653 services operating on objects local to an application. In the current standard, it is unspecified whether processes of one application share a single addressing space, and applications cannot make the assumption that pointers in one thread will be valid from another one. In the upcoming release of the standard, this will be clarified such that processes of one application share a single addressing space so they can also interact through global variables. However, ARINC 653 offers no support for hardware where concurrent access to globals needs to be programmed differently from the sequential programming model, such as hardware requiring explicit memory ordering, non cache-coherent hardware, etc. It is known that some existing ARINC 653 applications do assume that globals can be shared among processes. This is an invalid assumption, although it does work with most implementations.

Figure 1 shows the basic architecture that is coined from the terms defined above. Additionally, it shows the required

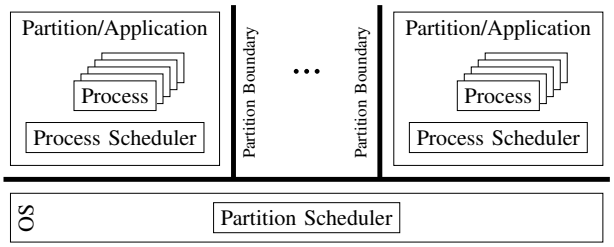


Figure 1. Basic Application Architecture for one computer; Partition boundaries in thick lines

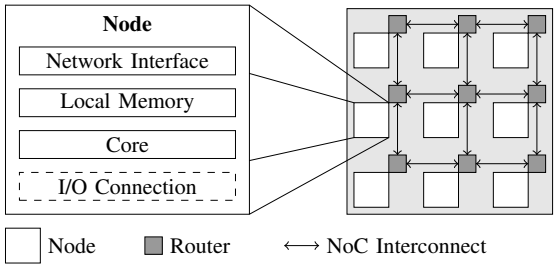


Figure 2. Manycore architecture [18]

partition boundaries. The OS must ensure that there are no interferences across these boundaries. Concerning the development process, this also means that changes within one application must not trigger changes in any other application. All in all, we state the following requirements and properties for safety-critical RTES: (1) The whole system must behave *predictably* and must therefore be *analysable*. This includes a *predictable timing behaviour* to ensure that all deadlines are kept. (2) *Partitioning* in time and space guarantees freedom of interference. Special care must be taken to make accesses to *shared resources* predictable. (3) Fine-grained communication takes only place between processes of the same application. If applications have to exchange data over partition boundaries, special mechanisms are provided by the OS. (4) Furthermore, there is ongoing research on the dynamic reconfiguration of software in safety critical systems [17]. While this is not part of today's standards, we view the capability for *reconfiguration* also as an important requirement for future avionic systems.

B. Multicores in Avionic Systems

Kinnan [1] identified several issues that are preventing a wide use of multicore processors in safety-critical RTES. Most of these issues relate to the certification of shared resources like caches, peripherals or memory controllers in a multicore processor. Wilhelm et al. [2] show how to circumvent the certification issues of shared resources through a diligent hardware design. Later work also shows that even if an architecture is very complex, smart configuration can still allow a feasible timing analysis [3].

Additionally, Kinnan also identified issues that inhere the concept of multicore. The replacement of multiple processors by one multicore processor can introduce the possibility of a single point of failure for the whole system. Separate processors have separate power feeds and clock sources, where the failure of one feed will not impact the other processors.

C. Manycores in Avionic Systems

Problems discussed above stem mostly from a fine-grained sharing of many hardware resources in today's multicore processors. In a NoC-based manycore (see figure 2), the single cores are decoupled more strongly. The only shared resources are the NoC interconnect and off-chip I/O.

We see a great benefit from such a hardware architecture. On the single-core processors used in today's aircrafts, the partitions of one computer share one core. Even with multicore computers, several partitions would have to share one core. With an increasing number of cores, it would be possible to assign each partition its own core or even a set of cores exclusively, resulting in a mapping like depicted in figure 3. The space and time partitioning thus is partly provided by the underlying hardware. We discuss this concept in more depth in later sections.

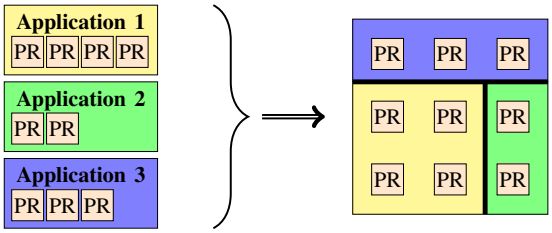


Figure 3. Mapping of processes and partitions to a manycore processor

III. Manycore Operating Systems

In this section we discuss some existing work on manycore operating systems exemplarily. Based on the presented works common principles for a manycore OS are deduced that form the base of our work.

A. Barrelfish

The Barrelfish [13] OS targets multicore processors, including those with heterogeneous core architectures. The design of Barrelfish is guided by the idea that today's computers are already distributed systems. Thus, also the OS should be reconsidered to exploit future hardware. Based on this idea, three design principles were defined for Barrelfish to form a *Multikernel OS*.

In Barrelfish, all communication between cores is *explicit*. The authors argue that this approach is amenable for correctness analysis as a theoretical foundation therefore already exists, e.g. Hoare's communicating processes [19]. The OS structure of Barrelfish is *hardware-neutral*. Hardware-related parts like message transport or CPU and device interfaces are implemented in a small kernel. All other parts of the OS are implemented uniformly on top. The state of OS components in Barrelfish is no longer shared, but *replicated*. OS instances are distributed homogeneously over all cores. If some local instance of the OS has to access data that is possibly shared,

this data is treated as a local replica. Consistency between the instances is ensured through messages.

B. Factored operating system

The *factored operating system (fos)* [14] is based on the idea that scheduling on a manycore processor should be a problem of space partitioning and no longer of time multiplexing. OS and application components are executed on separate cores each, thus forming a rather heterogeneous system. Separate servers provide different OS services. Communication is restricted to passing messages. On each core, fos provides an identical µKernel. Applications and servers run on top of this kernel. Applications can execute on one or more cores. The µKernel converts applications' system calls into messages for the affected server. Furthermore, the µKernel provides a reliable messaging and named mailboxes for clients. Namespaces help to improve protection. fos servers are inspired by internet servers. They work transaction-oriented and implement only stateless protocols. Transactions cannot be interrupted, however long latency operations are handled with the help of so-called *continuation* objects. While the operation is pending, the server thus can perform other work.

C. Tesselation and ROS

Tesselation [15] introduces the concept of *space-time partitioning (STP)* on a manycore processor for single-user client computers. A spatial partition contains a subset of the available resources exclusively and is isolated from other partitions. STP additionally time-multiplexes the partitions on the available hardware. Communication between partitions is restricted to messages sent over channels providing QoS guarantees. The authors discuss how partitioning can help improving performance of parallel application execution and reducing energy consumption in mobile devices. They also argue that STP can provide QoS guarantees and will enhance the security and correctness of a system.

Building on these concepts, Klues et al. [16] introduce the *manycore process (MCP)* as a new process abstraction. Threads within a MCP are scheduled in userspace, thus removing the need for corresponding kernel threads and making the kernel more scalable. Physical resources used by a MCP must be explicitly granted and revoked. They are managed within so-called *resource partitions*. The resources are only provisioned, but not allocated. ROS guarantees to make them available if requested. While they are not used by the partition, ROS can grant them temporarily to other partitions. Like Tesselation, ROS targets general-purpose client computers.

D. Summary

The works presented above base on the fact that a big problem for scalability of operating systems stems from the uncontrolled sharing of resources and especially the use of shared memory. They switch over to message passing and use shared memory only very restrictedly, if at all. Barrelfish and fos also change the view on the whole system. They view a manycore computer no longer as a monolithic block, but as a distributed system. These works have in common that they strongly separate software components by keeping as much data locally as possible. Tesselation continues this approach by introducing *partitioning* for general-purpose computing.

The issue of safety-critical systems has, to the best of our knowledge, not yet been addressed concretely in works concerning future manycore processors. Baumann et al. [13] give some hints about system analysis concerning their Barrelfish approach. The STP concept of Tesselation and its continuation in ROS pursues similar goals as ARINC 653 partitioning, but targets general-purpose computers. The problem of predictability that is central to safety-critical systems, is considered only marginally.

IV. Manycore Operating System for Safety-Critical Applications

The basic idea of our system architecture is to map ARINC 653 applications or even processes to separate cores. Each partition is represented by one core or a cluster of cores (see figure 3). Additionally, we allocate separate cores as servers to perform tasks that need global knowledge or that can only perform on a global level. These include off-chip I/O (for all partitions) and inter-partition communication. There is no need for a global scheduler among applications, regarding user code execution (but there may be a need for I/O or communication schedulers). If multiple processes of one application end up on one core, a local scheduler is required, and it can be somewhat simplified as it does not need to cross addressing space boundaries. If processes of one application end up on more than one core, according to ARINC 653-2 we are only required to implement ARINC 653 buffers, blackboards, events and semaphores such that they work across the cores of one application. The upcoming release of ARINC 653 will require implicit memory migration, which may in turn cause a performance degradation (although it may help that the network traffic remains fairly local).

Figure 4 outlines the overall architecture of the proposed Manycore Operating System for Safety-Critical systems (MOSSCA). The hardware base is formed by *nodes* which are connected by a *real-time interconnect* (cf. figure 2) that provides predictable traversal times for messages. An identical

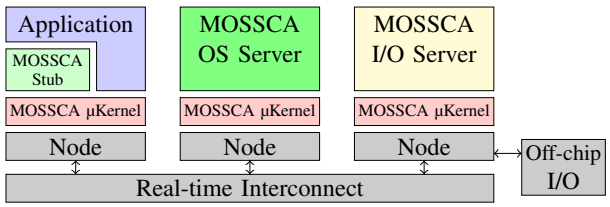


Figure 4. Overall System Architecture

MOSSCA µKernel on each node is responsible for configuration and management of the node's hard- and software. MOSSCA is split into two parts and runs in a distributed manner to achieve high parallelism. Application nodes are equipped with a MOSSCA stub that provides the functional

interface for applications to use the manycore OS. If a MOSSCA service cannot be performed on the calling node, the MOSSCA stub sends a request to a MOSSCA server running on a separate node. Nodes which are connected to external I/O facilities act as *I/O servers*. They are responsible for processing all I/O requests from applications concerning their I/O facility. The general concepts of MOSSCA are based on the works presented in section III. In the following sections, we discuss the additional properties that must be fulfilled to make the OS capable for safety-critical systems.

A. µKernel

The µKernel manages the node-local hardware devices. Concerning e.g. the interface to the NoC interconnect this includes a fine-grained configuration of the send/receive bandwidth within the bounds that are decided during system integration and ensuring that the local application keeps this constraints. The µKernel is also responsible for the software configuration of the node, i.e. loading the code of applications that should run on the node, including MOSSCA and I/O servers. Thus, the µKernel must possess basic mechanisms for coordination with other cores during the boot process, and also provide an interface for the MOSSCA server to allow online reconfiguration of the node. If several processes of an application are mapped to one node, the µKernel must also provide a process scheduler and, if necessary, means for protection between the processes. Code and data of the µKernel must be protected from applications. Finally, all µKernel services must deliver a predictable timing behaviour. This includes the mechanisms for boot coordination and reconfiguration.

B. Servers: General Techniques

Functionalities that must be globally available are implemented as servers. The stateless protocol of fos servers [14] provides a sound base, which we want to extend in MOSSCA. In safety-critical embedded systems, special care must be taken of the predictability of the servers:

1) All transactions that return immediately must be predictable, i.e. they must have at least an upper timing bound, but better perform in constant time. This problem is already solved for today's single processor systems [20].

2) As the server is a shared resource, the worst-case waiting time of a client consists not only of the raw processing time of the request. Instead, the time is increased considerably by requests from other clients that are possibly pending.

3) Special care must be given to long-latency operations that are handled through continuation objects. When the operation is finished, the server processes the continuation before other pending requests. Thus, these requests are delayed further. The constraints defined for server access must consider also such long-latency operations.

While these requirements are obvious, we see some challenges especially concerning requirement 2). Today, when dealing with shared resources, usually time multiplexing or prioritisation are used. Both of these approaches can lead to a high pessimism in worst-case execution time (WCET) analysis if the number of clients increases. Replicating the servers, if possible at all, can only be part of the solution. However, with concrete knowledge of the application, it might be possible to derive access constraints that allow a less pessimistic WCET estimation, when combined with time multiplexing or prioritisation techniques.

MOSSCA has to provide proper and possibly various means for the implementation of access constraints. Their concrete definition needs knowledge of the application and can only be performed during development of the system.

C. MOSSCA Stub and OS Server

MOSSCA is responsible for the management of all on-chip resources. With help of the µKernel, it manages the NoC interconnect and sets up send/receive policies for all nodes. The first MOSSCA server instance also coordinates the boot process of the chip and configures all other cores by deciding which code to execute on which core. If the need for online-reconfiguration of the system arises, this task is also managed by MOSSCA. The MOSSCA stubs on application cores provide the functional interface of e.g. ARINC 653 and additional, manycore-specific functionalities. The MOSSCA stubs perform as much work as possible locally. Only work that needs interaction with other nodes or a global knowledge is marshaled into messages that are sent to the MOSSCA server, where they are processed. Inter-partition communication is controlled by the MOSSCA. It provides message queues to store messages temporarily until the target partition is ready to receive them.

D. I/O Server

Through the integration of many cores on one chip, only dedicated cores will be able to access I/O hardware directly. These cores act as I/O server for the connected device. They handle all I/O requests from the other cores for this device. The I/O server takes care of time multiplexing and shaping of the applications' output towards its device. It provides means for bandwidth and latency management which allow to give certain guarantees to applications. The concrete implementation of the management will depend on the specific device. MOSSCA has to provide general helpers to alleviate this implementation. The I/O Server is also the primary interrupt handler for its device. If necessary, it may forward interrupt requests to specialised nodes. Then however, one must ensure to not degrade the timing predictability, e.g. when periodic messages arrive with jitter.

V. Research Challenges

Based on the system architecture proposed in section IV, we see the following research challenges on manycore operating systems for safety-critical applications:

1) Management of NoC: The OS must provide means to convey application-level bandwidth/latency requirements to the control mechanisms provided by the available NoC. Such approaches exist for today's manycore processors [21]. However, it is also necessary to develop fault tolerance mechanisms to e.g. detect and exclude nodes with faulty software from the NoC while sustaining the proper operation of all other nodes.

2) Weighting between distribution and centralisation of essential services: MOSSCA services should be executed on the same cores as the calling application as far as possible to improve the worst-case timing behaviour. This however means also to replicate the relevant code. The size of memory locally available on a core will be limited, thus posing a natural bound for the replication.

3) Predictable Servers: As discussed in section IV-B, a diligent design of the servers is essential. The derivation of application-specific access constraints will help tightening WCET estimations. ROS [16] presents another promising approach to this problem, whose applicability in safety-critical systems should be explored.

4) Control of I/O: Similar to NoC management, also for off-chip I/O operations the OS must give guarantees in terms of bandwidth and latency. Usually, I/O operations are afflicted with jitter. As I/O requests have to traverse the NoC, they can incur further jitter. The implementation of an I/O server must not add further jitter to the I/O operations or at least ensure a proper shaping of the jitter as not to degrade the responsiveness of the system.

5) Coordination of boot process: In an IMA system, even the startup time of a computer is critical. For a manycore computer this requires loading all code and data from ROM to the appropriate cores. This can be a very time-consuming process. Therefore, we imagine a staggered boot process, where the applications that are most critical are started foremost, while less critical applications are deferred.

6) Reconfiguration of software at runtime: This is an active research issue already today. On a manycore it must take the special properties and constraints of the processor into account, e.g. sharing of NoC bandwidth.

VI. State of Work

We are exploring the presented MOSSCA concepts based on a manycore simulator developed in our group [18]. However, our approach is not limited to purely NoC-based processors. We are also working on transferring the concepts to a clustered manycore which is developed in the parMERASA project[1].

VII. Summary

Fine-grained sharing of hardware resources is impacting the usability of today's multicore processors in safety-critical real-time systems. We have discussed, how these problems might be alleviated in future manycore processors, where resource sharing will happen on a more coarse level. A manycore OS must provide a sound base for applications on such future processors. Borrowing ideas from existing manycore operating systems, we devised a concept for a Manycore Operating System for Safety-Critical applications (MOSSCA). In our proposal, MOSSCA is distributed as far as possible, but some parts may or even must be centralised on dedicated servers. The central issue of a safety-critical system is the predictability of its behaviour, which today is achieved through partitioning the system. We achieve the partitioning partially by mapping different applications to different cores. Remaining sources of possible interferences are shared resources like the NoC and off-chip I/O. In precluding these interferences we see great challenges for future research.

[1] parMERASA project web site: http://www.parmerasa.eu/

Acknowledgements

Part of this research has been supported by the EC FP7 project parMERASA under Grant Agreement No. 287519.

References

[1] L. Kinnan, "Use of multicore processors in avionics systems and its potential impact on implementation and certification," in *28th Digital Avionics Systems Conference (DASC '09)*, Oct. 2009, pp. 1.E.4 1–6.

[2] R. Wilhelm *et al.*, "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, Jul. 2009.

[3] C. Cullmann *et al.*, "Predictability Considerations in the Design of Multi-Core Embedded Systems," in *Proceedings of Embedded Real Time Software and Systems*, May 2010, pp. 36–42.

[4] T. Ungerer *et al.*, "MERASA: Multicore Execution of HRT Applications Supporting Analyzability," *IEEE Micro*, vol. 30, pp. 66–75, 2010.

[5] D. N. Bui *et al.*, "Temporal isolation on multiprocessing architectures," in *Design Automation Conference (DAC)*, June 2011, pp. 274 – 279.

[6] F. Boniol *et al.*, "Deterministic execution model on cots hardware," in *Architecture of Computing Systems ARCS 2012*, ser. LNCS. Springer Berlin / Heidelberg, 2012, vol. 7179, pp. 98–110.

[7] J. Nowotsch and M. Paulitsch, "Leveraging Multi-Core Computing Architectures in Avionics," in *Ninth European Dependable Computing Conference (EDCC 2012)*, Sibiu, Romania, May 2012.

[8] *PikeOS, Product Data Sheet*, SYSGO AG, 2012.

[9] "Wind River VxWorks Web Site," visited 16.04.2012. [Online]. Available: http://www.windriver.com/products/vxworks/

[10] *TILE-Gx8036 Processor Specification Brief*, Tilera Corporation, 2011.

[11] J. Howard *et al.*, "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," in *IEEE International Solid-State Circuits Conference, ISSCC 2010, USA*. IEEE, Feb. 2010, pp. 108–109.

[12] B. D'Ausbourg *et al.*, "Deterministic Execution on Many-Core Platforms: application to the SCC," in *4th symposium of the Many-core Applications Research Community (MARC)*, Dec. 2011.

[13] A. Baumann *et al.*, "The multikernel: a new OS architecture for scalable multicore systems," in *22nd ACM Symposium on Operating Systems Principles (SOSP 2009), Big Sky, USA*. ACM, Oct. 2009, pp. 29–44.

[14] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 76–85, Apr. 2009.

[15] R. Liu *et al.*, "Tessellation: Space-Time Partitioning in a Manycore Client OS," in *Hot-Par '09*, Berkeley, CA, USA, Mar. 2009.

[16] K. Klues *et al.*, "Processes and Resource Management in a Scalable Many-core OS," in *HotPar '10*, Berkeley, CA, USA, Jun. 2010.

[17] C. Pagetti *et al.*, "Reconfigurable IMA platform: from safety assessment to test scenarios on the SCARLETT demonstrator," in *Embedded Real Time Software (ERTS'12)*, 2012.

[18] S. Metzlaff *et al.*, "A Real-Time Capable Many-Core Model," in *Work-in-Progress Session RTSS 2011*, Vienna, Austria, Nov. 2011.

[19] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585

[20] "ARINC 664, Aircraft Data Network, Part 1: Systems Concepts and Overview," 2006.

[21] C. Zimmer and F. Mueller, "Low contention mapping of real-time tasks onto tilepro 64 core processors," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, M. D. Natale, Ed. IEEE, 2012, pp. 131–140.

PRACTISE: a framework for PeRformance Analysis and Testing of real-time multIcore SchEdulers for the Linux kernel *

Fabio Falzoi, Juri Lelli, Giuseppe Lipari
{name.surname}@sssup.it - Scuola Superiore Sant'Anna - ITALY
Technical Paper

Abstract

The implementation of a multi-core real-time scheduler is a difficult task, due to the many problems that the developer encounters in debugging and testing parallel code inside the kernel. Also, it is often very difficult to compare the performance of alternative implementations, due to the many different parameters that influence the behaviour of the code.

In this paper, we present PRACTISE, a tool for developing, debugging, testing and analyse real-time scheduling data structures in user space. Unlike other similar tools, PRACTISE executes code in parallel, allowing to test and analyse the performance of the code in a realistic multiprocessor scenario.

After describing the organisation of the code, we show how easy it is to port the developed code in the Linux kernel. Also, we compare the performance of two different schedulers, SCHED_DEADLINE and SCHED_FIFO in the kernel and in the tool. Although PRACTISE cannot substitute performance analysis in the kernel, we show that, under certain conditions, PRACTISE can also be used as tool for early performance estimation.

1. Introduction

The wide diffusion of multi-core architectures in personal computing, servers and embedded systems, has revived the interest in multiprocessor scheduling, especially in the field of real-time applications. In fact, real-time scheduling on multi-core and multiprocessor systems is still an open research field both from the point of view of the theory and for the technical difficulties in implementing an efficient scheduling algorithm in the kernel.

*The research leading to these results has received funding from the European Community's Seventh Framework Programme n.248465 "S(o)OS – Service-oriented Operating Systems."

Regarding the second problem, let us discuss a (non exhaustive) list of problems that the prospective developer of a new scheduler must be faced with. The task scheduler is a fundamental part of the operating system kernel: a buggy scheduler will soon crash the system, usually at random and unexpected points. The major difficulty in testing and debugging a new scheduling algorithm derives from the fact that, when the system crashes, it is difficult to reconstruct the situation (i.e. the sequence of events and states) that led to the crash. The developer has to carefully analyse system logs and traces (for example using one of the tools described in Section 3), and reconstruct the state to understand what went wrong. More importantly, it is often impossible to impose a precise sequence of events: crashes can rarely be reproduced deterministically. Hence, it is practically impossible to run a sequence of test-cases.

This problem is exacerbated in multi-core architectures where the scheduler service routines run in parallel on the different processors, and make use of shared data structures that are accessed in parallel. In these cases, it is necessary to ensure that the data structures remain consistent under every possible interleaving of the service functions. A simple solution is to protect the shared data structure with locks. However, a single big lock reduces parallelism and performance does not scale; fine-grain locks may cause deadlock situations, without improving scalability; and lock-free algorithms are difficult to implement and prove correct. As a consequence, many important and interesting scheduling algorithms proposed in the research literature fail to be implemented on popular operating systems like Linux due to the difficulty of the task.

One reasonable approach would be to develop, debug, test and analyse the algorithms in user space. Once the main algorithm is sufficiently tested using user-space debugging and testing techniques, the same algorithm can be ported in the kernel. However, if no specific methodology is followed, the code must be written twice, increasing the possibility of introducing bugs in one of the two versions. Also, if one is unsure of

which algorithm, data structure or locking strategy is more appropriate, the number of versions to implement, test, analyse by hand may become very large.

Hence, we decided to tackle the "user-space approach" by proposing a simple framework to facilitate the development, testing and performance evaluation of scheduling algorithms in user space, and minimise the effort of porting the same algorithms in kernel spaces.

1.1. Contributions of this work

In this paper, we propose PRACTISE (PeRformance Analysis and TestIng of real-time multicore SchEdulers) for the Linux kernel: it is a framework for developing, testing and debugging scheduling algorithms in user space before implementing them in the Linux kernel, that alleviates at least part of the problems discussed above. In addition, PRACTISE allows to compare different implementations by providing early estimations of their relative performance. In this way, the most appropriate data structures and scheduler structure can be chosen and evaluated in user-space. Compared to other similar tools, like LinSched, the proposed framework allows true parallelism thus permitting a full test in a realistic scenario (a short comparison between is done in Section 2)

The main features of PRACTISE are:

- Rapid prototyping of scheduling data structures in user space;
- Effective, quick and extensive testing of the data structures though consistency tests;
- Real multi-core parallelism using multi-threading;
- Relative performance estimation between different algorithms and data structures in user space;
- Possibility to specify application load though probabilistic distributions of events, and statistical analysis;
- Ease of porting to the kernel or to other scheduling simulators.

PRACTISE is available as open source software, and a development version is available for download[1].

The rest of the paper is organised as follows. In Section 2 we provide a survey of existing testing and debugging tools for the Linux kernel; in Section 3 we describe the architecture of the tool and the main implementation choices that we followed; in Section 4 we evaluate the tool by reporting the performance as measured by the tools compared to the performance of the same algorithms in the kernel; finally, in Section 5 we discuss conclusion and future work.

2. State of the art

Several tools exist, as open-source software, that are geared towards, or can be used as effective means to implement, debug and analyse real-time scheduling algorithms for multiprocessor systems. Each one tackles the intrinsic toughness of this field from different angles, generally focusing on one single aspect of the problem.

A valuable tool during the development process of a scheduling algorithm would be the one that allows fast prototyping and easy debugging. Originally developed by the Real Time Systems Group at University of North Carolina at Chapel Hill, and currently maintained by P. Turner from Google, **LinSched** [3][2] lets developers modify the behaviour of the Linux scheduler and test these changes in user-space. One of the major strength points of this tool is that it introduces very few modifications in the kernel sources. The developer can thus write kernel code and, once satisfied by tests, it has kernel ready patches at hand. Furthermore, debugging is facilitated by the fact that LinSched runs as a single thread user-space program, that can hence be debugged with common user-space tools like GDB[3]. Even if single-threading is useful for debugging purposes, it can be a notable drawback when focusing on the analysis of behaviour assuming a high degree of concurrency. LinSched can indeed verify locking, but it cannot precisely model multi-core contention.

LITMUS$^{\mathbf{RT}}$[1] has a completely different focus. The *LITMUS^{RT}* patch, developed by the Real Time Systems Group at University of North Carolina at Chapel Hill, is a (soft) real-time extension of the Linux kernel that allows fast prototyping and evaluation of real-time (multiprocessor) scheduling algorithms on real hardware. The *LITMUS^{RT}* testbed provides an experimental platform that real-time system researchers can use to simplify the development process of scheduling and synchronisation algorithms (compared to modifying a stock Linux kernel). Another nice feature of this testbed is an integrated tracing infrastructure (FeatherTrace [2]) with which performance and overhead data can be collected for off-line processing. Being a research tool rather than a production-quality system, *LITMUS^{RT}* does not target Linux mainline inclusion nor POSIX-compliance: in other words code patches

[1] At the time of submission (April 29 2012), the software can be downloaded cloning the repository available at `https://github.com/Pippolo84/PRAcTISE`

[2] v3.3-rc7 release announce: http://bit.ly/IJsyV3.

[3] http://sources.redhat.com/gdb/

created with it cannot be seamless applied to a "vanilla" Linux kernel.

Lots of other tools exist that make kernel developers lives easier during debugging, some of them can also be used to collect performance data or even extract execution traces from a running system. Among others, these are probably part of every kernel developer arsenal:

- **KVM** [4] **+ GDB**: the very first step after having modified the kernel is usually to run it on a virtualized environment. The KVM virtual machine can here be useful as it can be attached, and controlled, by the GNU Project Debugger (GDB). However, this solution can hardly be used in presence of high concurrency; moreover, it can occasionally affect the repeatability of certain bugs.
- **perf**[9]: the performance counter subsystem in Linux can be used to collect scheduling events and performance data from a real execution. It can also be used in conjunction with LinSched, as it can record an application behaviour that can later be played back in the simulator.
- **Ftrace**[11]: a tracing utility built directly into the Linux kernel. Ftrace is a valuable debugging tool as it brings to Linux the ability to see what is happening inside the kernel. With the ability of synchronise a user-space testing application with kernel execution, one can track function calls up to the point where a bug may happen.
- **LTTng**[4, 5]: the Linux Trace Toolkit is an highly efficient tracing tool for Linux that can help tracking down performance issues and debugging problems involving concurrent execution.

PRACTISE adds one more powerful weapon to this arsenal: the possibility to test and analyse parallel code (like lock-free data structures) in user space via multithreading.

3. PRACTISE Architecture

In this section, we describe the basic structure of our tool. PRACTISE emulates the behaviour of the LINUX scheduler subsystem on a multi-core architecture with M parallel cores. The tool can be executed on a machine with N cores, with N that can be less, equal to or greater than M. The tool can be executed in one of the following modes:

- testing;
- performance analysis.

Each processor in the simulated system is modelled by a software thread that performs a cycle in which:

- scheduling events are generated at random;
- the corresponding scheduling functions are invoked;
- statistics are collected.

In *testing mode*, a special "testing" thread is executed periodically that performs consistency checks on the shared data structures. In the *performance analysis mode*, instead, each thread is *pinned* on a processor, and the memory is locked to avoid spurious page faults; for this reason, to obtain realistic performances it is necessary to set $M \leq N$.

3.1. Ready queues

The Linux kernel scheduler uses one separate ready queue per each processor. A ready task is always enqueued in one (and only one) of these queues, even when it is not executing. This organisation is tailored for partitioned schedulers and when the frequency of task migration is very low. For example, in the case of non real-time best effort scheduling, a task usually stays on the same processor, and periodically a load-balancing algorithm is called to distribute the load across all processors.

This organisation may or may not be the best one for global scheduling policies. For example the `SCHED_FIFO` and `SCHED_RR` policies, as dictated by the POSIX standard, requires that the m highest priority tasks are scheduled at every instant. Therefore, a task can migrate several times, even during the same periodic instance.

The current multi-queue structure is certainly not mandatory: a new and different scheduler could use a totally different data structure (for example a single global ready queue); however, the current structure is intertwined with the rest of the kernel and we believe that it would be difficult to change it without requiring major changes in the rest of the scheduler. Therefore, in the current version of PRACTISE we maintained the structure of distributed queues as it is in the kernel. We plan to extend and generalise this structure in future versions of the tool.

Migration between queues is done using two basic functions: *push* and *pull*. The first one tries to migrate a task from the local queue of the processor that calls the function to a remote processor queue. In order to do this, it may use additional global data structures to

[4] Kernel Based Virtual Machine: http://bit.ly/IdlzXi

select the most appropriate queue. For example: the current implementation of the fixed priority scheduler in Linux uses a priority map (implemented in *cpupri.c*) that records for each processor the priority of the highest priority tasks; the SCHED_DEADLINE [7, 8] patch uses a max heap to store the deadlines of the tasks executing on the processors.

The *pull* does the reverse operation: it searches for a task to "pull" from a remote processor queue to the local queue of the processor that calls the function. In the current implementation of SCHED_{FIFO,RR} and SCHED_DEADLINE, no special data structure is used to speed up this operation. We developed and tested in PRACTISE a min-heap for reducing the duration of the *pull* operation, but, driven by not satisfactory performance figures, we didn't port the structure inside the kernel. Instead, we are currently investigating several different data structures with the aim of comparing them and select the most efficient to be implemented in the next release of the SCHED_DEADLINE patch.

Tasks are inserted into (removed from) the ready queues using the `enqueue()` (`dequeue()`) function, respectively. In Linux, the queues are implemented as red-black trees. In PRACTISE, instead, we have implemented them as priority heaps, using the data structure proposed by B. Brandenburg [5]. However, it is possible to implement different algorithms for queue management as part of the framework: as a future work, we plan to implement alternative data structures that use lock-free algorithms.

3.2. Locking and synchronisation

PRACTISE uses a range of locking and synchronisation mechanisms that mimic the corresponding mechanisms in the Linux kernel. An exhaustive list is given in Table 1. These differences are major culprits for the slight changes needed to port code developed on the tool in the kernel 4.1.

It has to be noted that `wmb` and `rmb` kernel memory barriers have no corresponding operations in user-space; therefore we have to issue a full memory barrier (`__sync_synchronize`) for every occurrence of them.

3.3. Event generation and processing

PRACTISE cannot execute or simulate a real application. Instead, each threads (that emulates a processor) periodically generates random scheduling events according to a certain distribution, and calls the scheduler functions. Our goals are to debug, test, compare and evaluate real-time scheduling algorithms for multi-core processors. Therefore, we identified two main events: task *activation* and *blocking*. When a task is activated, it must be inserted in one of the kernel ready queues; since such an event can cause a preemption, the scheduler is invoked, data structures are updated, etc. Something similar happens when a task self-suspends (for example because it blocks on a semaphore, or it suspends on a timer).

The pseudo-code for the task activation is function `on_activation()` described in Figure 1. The code mimics the sequence of events that are performed in the Linux code:

- First, the task is inserted in the local queue.
- Then, the scheduler performs a *pre-schedule*, corresponding to `pull()`, which looks at the global data structure `pull_struct` to find the task to be pulled; if it finds it, does a sequence of `dequeue()` and `enqueue()`.
- Then, the Linux scheduler performs the real schedule function; this corresponds to setting the `curr` pointer to the executing task. In PRACTISE this step is skipped, as there is no real context switch to be performed.
- Finally, a *post-schedule* is performed, consisting of a `push()` operation, which looks at the global data structure `push_struct` to see if some task need to be migrated, and in case the response is positive, performs a `dequeue()` followed by an `enqueue()`. A similar thing happens when a task blocks (see function `on_block()`).

The pseudo code shown in Figure 1 is an overly simplified, schematic version of the code in the tool; the interested reader can refer to the original source code[6] for additional details.

As anticipated, every processor is simulated by a periodic thread. The thread period can be selected from the command line and represents the average frequency of events arriving at the processor. At every cycle, the thread randomly select one between the following events: `activation`, `early finish` and `idle`. In the first case, a task is generated with a random value of the deadline and function `on_activation()` is called. In the second case, the task currently executing on the processor blocks: therefore function `on_block()` is called. In the last case, nothing happens. Additionally, in all cases, the deadline of the executing task is checked against the current time: if the

[5] Code available here: `http://bit.ly/IozLxM`.

[6] `https://github.com/Pippolo84/PRAcTISE`

Linux	PRAcTISE	Action
raw_spin_lock	pthread_spin_lock	lock a structure
raw_spin_unlock	pthread_spin_unlock	unlock a structure
atomic_inc	__sync_fetch_and_add	add a value in memory atomically
atomic_dec	__sync_fetch_and_sub	subtract a value in memory atomically
atomic_read	simple read	read a value from memory
wmb	__sync_synchronize	issue a memory barrier
rmb	__sync_synchronize	issue a read memory barrier
mb	__sync_synchronize	issue a full memory barrier

Table 1: Locking and synchronisation mechanisms (Linux vs. PRAcTISE).

```
pull() {
  bool found = find(pull_struct, &queue);
  if (found) {
    dequeue(&task, queue);
    enqueue(task, local_queue);
  }
}

push() {
  bool found = find(push_struct, &queue);
  if (found) {
    dequeue(&task, local_queue);
    enqueue(task, queue);
  }
}

on_activation(task) {
  enqueue(task, local_queue);
  pull();      /* pre-schedule  */
  push();      /* post-schedule */
}

on_block(task) {
  dequeue(&task, local_queue);
  pull();      /* pre-schedule  */
  push();      /* post-schedule */
}
```

Figure 1: Main scheduling functions in PRACTISE

deadline has passed, then the current task is blocked, and function `on_block()` is called.

Currently, it is possible to specify the period of the thread cycle; the probability of an activation event; and the probability of an early finish.

3.4. Data structures in PRACTISE

PRACTISE has a modular structure, tailored to provide flexibility in developing new algorithms. The interface exposed to the user consists of hooks to functions that each global structure must provide. The most important hooks:

- `data_init`: initialises the structure, e.g., spinlock init, dynamic memory allocation, etc.
- `data_cleanup`: performs clean up tasks at the end of a simulation.
- `data_preempt`: called each time an `enqueue()` causes a preemption (the arriving tasks has higher priority that the currently executing one); modifies the global structure to reflect the new local queue status.
- `data_finish`: *data_preempt* dual (triggered by a `dequeue()`).
- `data_find`: used by a scheduling policy to find the best CPU to (from) which push (pull) a task.
- `data_check`: implements the *checker* mechanism (described below).

PRACTISE has already been used to slightly modify and validate the global structure we have previously implemented in SCHED_DEADLINE [8] to speed-up `push()` operations (called *cpudl* from here on). We also implemented a corresponding structure for `pull()` operations (and used the tool to gather performance data from both). Furthermore, we back-ported in PRACTISE the mechanism used by SCHED_FIFO to improve `push()` operations performance (called *cpupri* from here on).

We plan to exploit PRACTISE to investigate the use of different data structures to improve the efficiency of the aforementioned operations even further. However, we leave this task as future work, since this paper is focused on describing the tool itself.

One of the major features provided by PRACTISE is the *checking* infrastructure. Since each data structure has to obey different rules to preserve consistency among successive updates, the user has to equip the implemented algorithm with a proper checking function. When the tool is used in testing mode, the `data_check` function is called at regular intervals. Therefore, an on-line validation is performed in presence of real concurrency, thus increasing the probability of discovering bugs at an early stage of the development process. User-space debugging techniques can then be used to fix design or developing flaws.

To give the reader an example, the *checking* function for SCHED_DEADLINE *cpudl* structure ensures the max-heap property: if B is a child node of A, then $deadline(A) \geq deadline(B)$; it also check consistency between the heap and the array used to perform updates on intermediate nodes (see [8] for further details). We also implemented a checking function for *cpupri*: periodically, all ready queues are locked, and the content of the data structure is compared against the corresponding highest priority task in each queue, and the consistency of the flag `overloaded` in the `struct root_domain` is checked. We found that the data structure is always perfectly consistent to an external observer.

3.5. Statistics

To collect the measurements we use the TSC (Time Stamp Counter) of IA-32 and IA-64 Instruction Set Architectures. The TSC is a special 64-bit per-CPU register that is incremented every clock cycle. This register can be read with two different instructions: RDTSC and RDTSCP. The latter reads the TSC and other information about the CPUs that issues the instruction itself. However, there are a number of possible issues that needs to be addressed in order to have a reliable measure:

- *CPU frequency scaling and power management*. Modern CPUs can dynamically vary frequency to reduce energy consumption. Recently, CPUs manufacturer have introduced a special version of TSC inside their CPUs: *constant TSC*. This kind of register is always incremented at CPU maximum frequency, regardless of CPU actual frequency. Every CPU that supports that feature has the flag *constant_tsc* in `/proc/cpuinfo` proc file of Linux. Unfortunately, even if the update rate of TSC is constant in these conditions, the CPU frequency scaling can heavily alter measurements by slowing down the code unpredictably; hence, we have conducted every experiment with all CPUs at fixed maximum frequency and no power-saving features enabled.

- *TSC synchronisation between different cores*. Since every core has its own TSC, it is possible that a misalignment between different TSCs may occur. Even if the kernel runs a synchronisation routine at start up (as we can see in the kernel log message), the synchronisation accuracy is typically in the range of several hundred clock cycles. To avoid this problem, we have set CPU affinity of every thread with a specific CPU index. In other words we have a 1:1 association between threads and CPUs, fixed for the entire simulation time. In this way we also prevent thread migration during an operation, which may introduce unexpected delays.

- *CPU instruction reordering*. To avoid instruction reordering, we use two instructions that guarantees serialisation: RDTSCP and CPUID. The latter guarantees that no instructions can be moved over or beyond it, but has a non-negligible and variable calling overhead. The former, in contrast, only guarantees that no previous instructions will be moved over. In conclusion, as suggested in [10], we used the following sequence to measure a given code snippet:

```
CPUID
RDTSC
code
RDTSCP
CPUID
```

- *Compiler instruction reordering*. Even the compiler can reorder instructions; so we marked the inline asm code that reads and saves the TSC current value with the keyword *volatile*.

- *Page faults*. To avoid page fault time accounting we locked every page of the process in memory with a call to `mlockall`.

PRACTISE collects every measurement sample in a global multidimensional array, where we keep samples coming from different CPUs separated. After all simulation cycles are terminated, we print all of the samples to an output file.

By default, PRACTISE measures the following statistics:

- duration and number of *pull* and *push* operations;

- duration and number of *enqueue* and *dequeue* operations;

- duration and number of `data_preempt`, `data_finish` and `data_find`.

Of course, it is possible to add different measures in the code of a specific algorithm by using PRACTISE's functions. In the next section we report some experiment with the data structures currently implemented in PRACTISE.

4. Evaluation

In this section, we present our experience in implementing new data structures and algorithms for the Linux scheduler using PRACTISE. First, we show how difficult is to port a scheduler developed with the help of PRACTISE into the Linux kernel; then, we report performance analysis figures and discuss the different results obtained in user space with PRACTISE and inside the kernel.

4.1. Porting to Linux

The effort in porting an algorithm developed with PRACTISE in Linux can be estimated by counting the number of different lines of code in the two implementations. We have two global data structures implemented both in PRACTISE and in the Linux kernel: *cpudl* and *cpupri*.

We used the `diff` utility to compare differences between user-space and kernel code of each data structure. Results are summarised in Table 2. Less than 10% of changes were required to port *cpudl* to Linux, these differences mainly due to the framework interface (pointers conversions). Slightly higher changes ratio for *cpupri*, due to the quite heavy use of atomic operations (see Section 3.2). An example of such changes is given in Figure 2 (lines with a − correspond to user-space code, while those with a + to kernel code).

Structure	Modifications	Ratio
cpudl	12+ 14-	8.2%
cpupri	17+ 21-	14%

Table 2: Differences between user-space and kernel code.

The difference on the synchronisation code can be reduced by using appropriate macros. For example, we could introduce a macro that translates to `__sync_fetch_and_add` when compiled inside PRACTISE, and to the corresponding Linux code otherwise. However, we decided for the moment to maintain the different code to highlight the differences between the two frameworks. In fact, debugging, testing and analyse the synchronisation code is the main difficulty, and the main goal of PRACTISE; therefore, we thought that it is worth to show such differences rather than hide them.

```
[...]
-void cpupri_set(void *s, int cpu, int newpri)
+void cpupri_set(struct cpupri *cp, int cpu,
+                int newpri)
 {
-   struct cpupri *cp = (struct cpupri*) s;
    int *currpri = &cp->cpu_to_pri[cpu];
    int oldpri  = *currpri;
    int do_mb = 0;
@@ -63,57 +61,55 @@
    if (newpri == oldpri)
         return;

-   if (newpri != CPUPRI_INVALID) {
+   if (likely(newpri != CPUPRI_INVALID)) {
         struct cpupri_vec *vec =
                       &cp->pri_to_cpu[newpri];

         cpumask_set_cpu(cpu, vec->mask);
-        __sync_fetch_and_add(&vec->count, 1);
+        smp_mb__before_atomic_inc();
+        atomic_inc(&(vec)->count);
         do_mb = 1;
    }
[...]
```

Figure 2: Comparison using `diff`.

However, the amount of work shouldered on the developer to transfer the implemented algorithm to the kernel, after testing, is quite low reducing the probability of introducing bugs during the porting. Moreover, this residual amount of handwork could be eliminated using simple translation scripts (e.g., sed). Additional macros will be introduced in future version of PRACTISE to minimise such effort even further.

4.2. Experimental setup

The aim of the experimental evaluation is to compare performance measures obtained with PRACTISE with what can be extracted from the execution on a real machine.

Of course, we cannot expect the measures obtained with PRACTISE to compare directly with the measure obtained within the kernel; there are too many differences between the two execution environments to make the comparison possible: for example, the completely different synchronisation mechanisms. However, comparing the performance of two alternative algorithms within PRACTISE can give us an idea of their relative

performance within the kernel.

4.3. Results

In Linux, we rerun experiments from our previous work [8] on a Dell PowerEdge R815 server equipped with 64GB of RAM, and 4 AMD^R Opteron^{TM} 6168 12-core processors (running at 1.9 GHz), for a total of 48 cores. This was necessary since the *cpupri* kernel data structure has been modified in the meanwhile [7] and the PRACTISE implementation is aligned with this last *cpupri* version. We generated 20 random task sets (using the `randfixedsum` [6] algorithm) with periods log-uniform distributed in [10ms, 100ms], per CPU utilisation of 0.6, 0.7 and 0.8 and considering 2, 4, 8, 16, 24, 32, 40 and 48 processors. Then, we ran each task set for 10 seconds using a synthetic benchmark [8] that lets each task execute for its WCET every period. We varied the number of active CPUs using the Linux CPU hot plug feature and we collected scheduler statistics through `sched_debug`. The results for the Linux kernel are reported in Figures 3a and 3b, for modifying and querying the data structures, respectively. The figures show the number of cycles (y axis) measured for different number of processors ranging from 2 to 48 (x axis). The measures are shown in boxplot format: a box indicates all data comprised between the 25% and the 75% percentiles, whereas an horizontal lines indicates the median value; also, the vertical lines extend from the minimum to the maximum value.

In PRACTISE we run the same experiments. As depicted in Section 3.3, random scheduling events generation is instead part of PRACTISE . We varied the number of active processors from 2 to 48 as in the former case.

We set the following parameters: 10 milliseconds of thread cycle; 20% probability of new arrival; 10% probability of finish earlier than deadline (cpudl) or runtime (cpupri); 70% probability of doing nothing. These probability values lead to rates of about 20 task activations / (core * s), and about 20 task blocking / (core * s).

The results are shown in Figures 6a and 5a for modifying the *cpupri* and *cpudl* data structures, respectively; and in Figures 6b and 5b for querying the *cpupri* and *cpudl* data structures, respectively.

Insightful observations can be made comparing performance figures for the same operation obtained from the kernel and from simulations. Looking at Figure 3a we see that modifying the *cpupri* data structure is generally faster than modifying *cpudl*: every measure corresponding to the former structure falls below 1000 cycles while the same operation on *cpudl* takes about 2000 cycles. Same trend can be noticed in Figure 6a and 5a. Points dispersion is generally a bit higher than in the previous cases; however median values for *cpupri* are strictly below 2000 cycles while *cpudl* never goes under that threshold. We can see that PRACTISE overestimates this measures: in Figure 6a we see that the estimation for the *find* operation on *cpupri* are about twice the ones measured in the kernel; however, the same happens for *cpudl* (in Figure 5a); therefore, the relative performance of both does not change.

Regarding query operations the ability of PRACTISE to provide an estimation of actual trends is even more evident. Figure 3b shows that a *find* on *cpudl* is generally more efficient than the same operation on *cpupri*; this was expected, because the former simple reads the top element of the heap. Comparing Figure 6b with Figure 5b we can state that latter operations are the most efficient also in the simulated environment.

Moreover, we used PRACTISE to compare the time needed to modify and query the two global data structure for push and pull operations for cpudl. As we can see in Figure 5a and Figure 5b compared against Figure 6a and Figure 6b, the results are the same, as the data structures used are the same. We haven't compared cpudl pull operation against cpupri pull operation since the latter doesn't have a global data structure that hold the status of all run queues where we can issue find and set operations.

5. Conclusions and future work

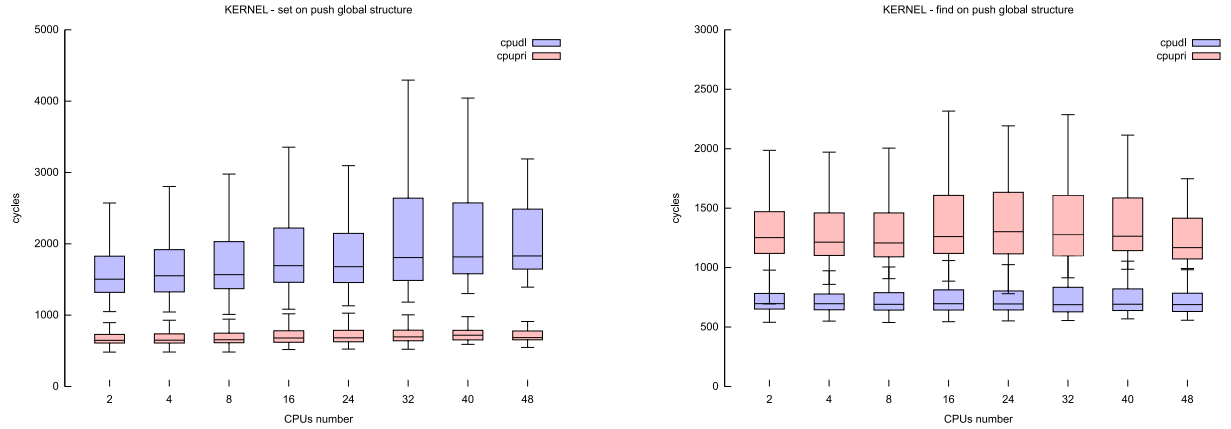
In this paper we introduced PRACTISE, a framework for PeRformance Analysis and TestIng of real-time multicore SchEdulers for the Linux kernel. PRACTISE enables fast prototyping of real-time multicore scheduling mechanisms, allowing easy debugging and testing of such mechanisms in user-space. Furthermore, we performed an experimental evaluation of the simulation environment, and we showed that PRACTISE can also be used to perform early performance estimation.

In future work, we plan to refine the framework adherence to the Linux kernel. In doing so, we have to enhance task affinity management, local run queues capabilities and provide the possibility to generate random scheduling events following probability distributions gathered from real task sets execution traces.

Furthermore, we will exploit PRACTISE to perform a comparative study of different data structures to improve *pull* operation performance. In particular we will try to implement some lock-free data structures and subsequently compare their performances against

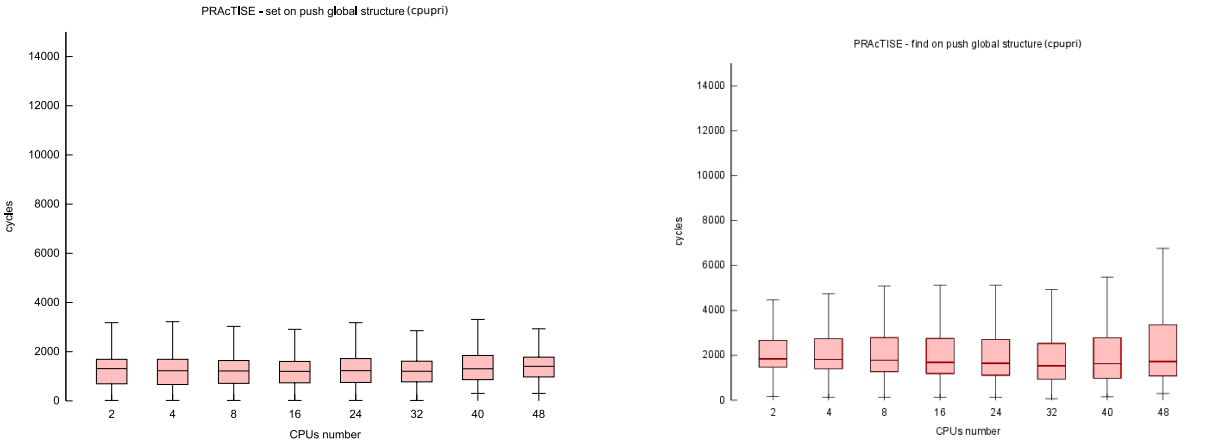
[7] More info here: `http://bit.ly/KjoePl`

[8] rt-app: `https://github.com/gbagnoli/rt-app`.



(a) modify (b) query

Figure 3: Number of cycles (mean) to a) modify and b) query the global data structure (*cpudl* vs. *cpupri*), kernel implementation.



(a) modify (b) query

Figure 4: Number of cycles (mean) to a) modify and b) query the global data structure (*cpupri*), on PRACTISE.

the heap already presented.

As a concluding use-case, it is worth mentioning that PRACTISE has already been used as a testing environment for the last SCHED_DEADLINE release on the LKML [9]. The *cpudl* global data structure underwent major changes that needed to be verified. The tested code has been finally merged within the patch set.

References

[1] Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (*LITMUS^{RT}*). http://www.litmus-rt.org/index.html.

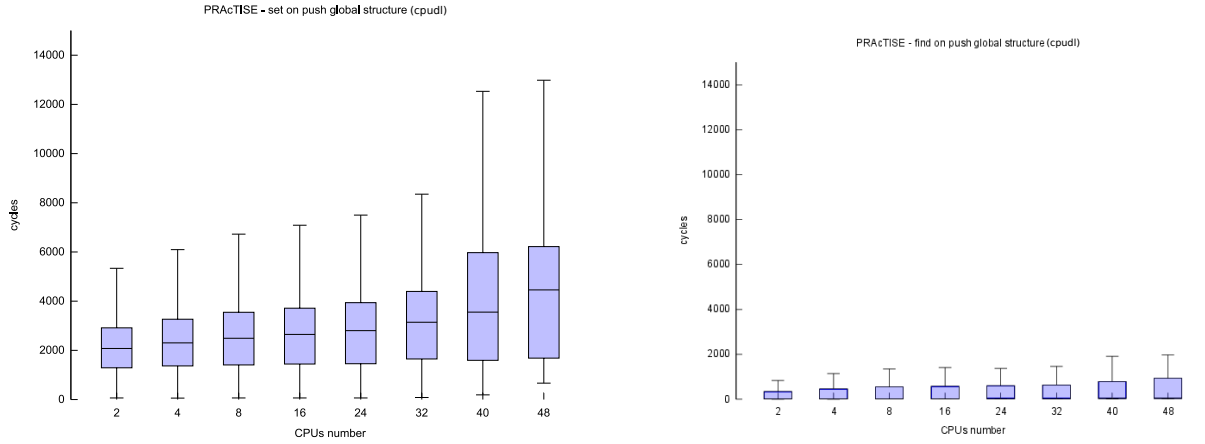
[2] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proc. 3th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2007)*, National ICT Australia, July 2007.

[3] John M. Calandrino, Dan P. Baumberger, Tong Li, Jessica C. Young, and Scott Hahn. Linsched: The linux scheduler simulator. In J. Jacob and Dimitrios N. Serpanos, editors, *ISCA PDCCS*, pages 171–176. ISCA, 2008.

[4] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proc. Ottawa Linux Symposium (OLS 2006)*, pages 209–224, July 2006.

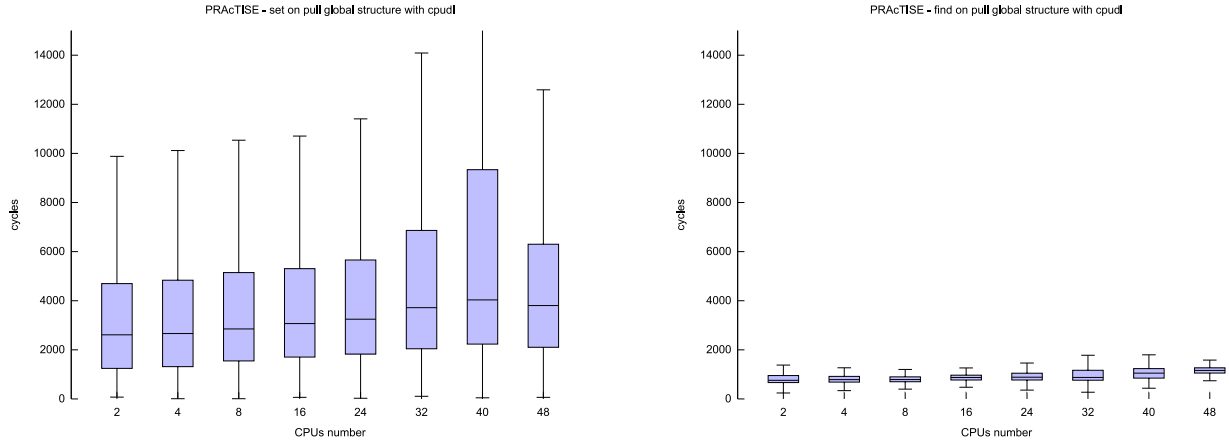
[5] Mathieu Desnoyers. Lttng, filling the gap between kernel instrumentation and a widely usable kernel tracer. Linux Foundation Collaboration Summit, April 2009.

[9] LKML (Linux Kernel Mailing List) thread available at: `https://lkml.org/lkml/2012/4/6/39`



(a) modify (b) query

Figure 5: Number of cycles (mean) to a) modify and b) query the global data structure (*cpudl*), on PRACTISE.



(a) modify (b) query

Figure 6: Number of cycles (mean) to a) modify and b) query the global data structure for speed-up SCHED_DEADLINE pull operations, on PRACTISE.

[6] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, Brussels, Belgium, July 2010.

[7] Dario Faggioli, Michael Trimarchi, Fabio Checconi, Marko Bertogna, and Antonio Mancina. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC)*, Honolulu (USA), March 2009.

[8] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2011)*. July 2011.

[9] Arnaldo Melo. The new linux 'perf' tools. In *17 International Linux System Technology Conference (Linux Kongress)*, Georg Simon Ohm University Nuremberg (Germany), September 21-24 2010.

[10] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 Instruction Set Architectures. Intel White Paper, September 2010.

[11] Steven Rostedt. The world of ftrace. Linux Foundation Collaboration Summit, April 2009.

CoS: A New Perspective of Operating Systems Design for the Cyber-Physical World

[Forward-looking Paper]

Vikram Gupta[†‡], Eduardo Tovar[†], Nuno Pereira[†], Ragunathan (Raj) Rajkumar[‡]
[†]CISTER Research Center, ISEP, Polytechnic Institute of Porto, Portugal
[‡]Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA
vikramg@ece.cmu.edu, {emt, nap}@isep.ipp.pt, raj@ece.cmu.edu

***Abstract*—Our day-to-day life is dependent on several embedded devices, and in the near future, many more objects will have computation and communication capabilities enabling an Internet of Things. Correspondingly, with an increase in the interaction of these devices around us, developing novel applications is set to become challenging with current software infrastructures. In this paper, we argue that a new paradigm for operating systems needs to be conceptualized to provide a conducive base for application development on Cyber-physical systems. We demonstrate its need and importance using a few use-case scenarios and provide the design principles behind, and an architecture of a *co-operating system* or *CoS* that can serve as an example of this new paradigm.**

I. Introduction

The penetration of embedded-systems in our daily lives is increasing at a tremendous rate, and even today, a human being can have tens of devices around him that have computational capabilities. In the future, not only the number of such devices is set to increase further, their capability to communicate among themselves and accomplish complex and distributed logic will become more widespread. In addition to the current *smart* devices such as mobile-phones, music players and tablets, even the *dumb* devices such as lamps, tables and chairs may have computation capabilities and contribute to ambient intelligence. This possible trend has led researchers in academia and industry to foresee an *Internet of Things*, where all (or most) of the objects will be connected to each other and the internet. Such a highly connected world will further enable several applications like home automation, intelligent ambience, green buildings and so on. However, full potential of highly-connected cooperating objects is still difficult to perceive, as there is scope for diverse and revolutionary applications that may not have been conceived yet.

To enable the development of such new applications, new paradigms for embedded-systems software are required. We believe that the currently available operating systems and programming abstractions may not encourage an environment for active application development for future networked embedded systems. In this paper, we argue that the design of the operating systems for networked embedded systems needs to be thought from a different perspective than the one already taken in the popular solutions like TinyOS [1], Contiki [2], Nano-RK [3] etc. Most of the popular research works in the direction of facilitating programming on sensor networks assume that the existing operating systems are the *de-facto* platforms upon which the middleware or the programming abstractions have to be built. This assumption needs to be thought again from a top-down perspective where the new goal is to support dynamic deployment and management for network-level applications.

Existing operating systems were designed to ease the programming of specific hardware that was developed as prototypes for wireless sensor networks. Programming these devices on *bare-metal* is complex and requires high degree of expertise in embedded systems. Platforms like MicaZ and TelosB are resource-constrained yet powerful-enough devices that can easily support a small operating system, custom communication stacks and one or more applications. Operating systems were designed from the perspective of easing the application development process on individual devices because even in their standalone operation they are complex systems with a processor, a radio, several sensors, a programming/communication interface over the USB or the serial port and so on. These hardware and software platforms have contributed a lot towards the development of ground-breaking research and proof-of-concept ideas. Moreover, the research in these areas provided a vision for the future of networked embedded systems. To achieve the goal of ubiquitous connectivity of embedded devices described earlier, there is a need to design (distributed) operating systems from scratch that completely isolate the users from node-level intricacies, and take the application development to a higher level where the whole network ecosystem can be viewed as a single organism. We believe that revamping the way operating systems are designed is a first step towards this goal.

By networked embedded systems we refer to the broader area of Cyber-Physical Systems (CPS) that react to the environment in addition to just sensing the physical quantities as in the case of wireless sensor networks. Timeliness is an important requirement of CPS, because of the tight integration of sensing and actuation. We believe that it is time we move from an operating system to a *co-operating system* or *CoS*, that embodies all fundamental functionalities necessary for encouraging application development for networked embedded systems directly above it. *CoS* is a truly distributed operating system, in the way that it provides a geographically distributed view of the operating system to the user rather than abstracting the network as a single machine. In the rest of this paper, we

describe a few key principles that can motivate the design of such a cooperating-system, and we propose a possible architecture that can satisfy those principles.

II. State of the Art

Many solutions have been designed that aim to provide an environment for convenient application development for networked embedded systems. From the perspective of allowing the development of diverse applications on cyber-physical systems, we classify them into three major classes.

A. Operating Systems

Earlier operating systems and even the more recent ones provide several convenient abstractions for programming the hardware. The popular sensor network operating systems like Contiki, TinyOS, etc., all allow one or more applications to be developed for hardware platforms, and the network-level coordination is the responsibility of the application programmer. These operating systems facilitated and supported computer scientists familiar with programming of general-purpose computers to develop applications for embedded hardware.

Some newer operating systems like LiteOS [4] provides a UNIX-like interface for sensor networks and each device can be accessed or written-to like a file. HomeOS [5], [6] allows connectivity of heterogenous devices such that a typical user can develop complex logic using the appliances in a modern home. HomeOS is a centralized design that connects the deployed devices and provides an interface for configuring the devices according to the needs of the users, based on access privileges and time of the day, for example. HomeOS is an interface above the home automation infrastructure and is closer to being a middleware-layer rather than an OS.

B. Middleware and Abstractions

Facilitating the development and the deployment of applications on heterogenous sensor networks has been a key driver behind the design of several middleware and programming abstractions proposed in the past. Most of the solutions discussed in the recent survey by Mottola and Picco [7] allow programming the network as a whole, while abstracting the user from lower-level complexities. Several different solutions have been proposed that serve varied goals, but anecdotal evidence suggests that almost none of those have been adopted in actual deployments or test-beds other than those for which these systems were built. When a new application is conceived, researchers typically find it more convenient to develop their own middleware/programming framework on top of a popular operating system to deploy and test their application instead of using an existing solution. The reasons behind this relatively less enthusiastic adoption of middleware can be several. Visibility of the solution, maturity of the software, hardware platforms supported and application domains covered, are few such factors that dictate the popularity of a middleware or a programming abstraction.

In addition to those, developers generally are not able to place confidence in third-party middleware for their applications because of the lack of robustness and support. Debugging may eventually require delving into the middleware code and its interactions with the underlying operating system. We aim to design a co-operating system to overcome these limitations, such that all the functionality of the underlying hardware and the possible interactions of devices can be visible to the user-interface, while providing easy development.

C. Standards

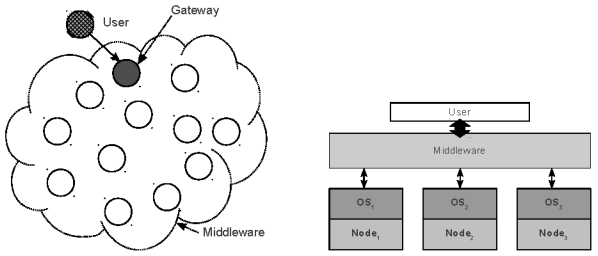
Several standards are being promoted by the industry and academia to foster interoperability between various appliances at home, and hence allow development of applications. Examples of such protocols are DLNA [8], Z-Wave [9] and OSIAN [10]. These standards can be great contributors towards distributed application development for appliances, and any programming system should be able to leverage these standards for communicating with heterogenous devices.

III. Use-case Scenarios

The operating system we envision (propose) should be generic-enough for most of the cyber-physical applications and should allow rapid application development as well. We consider the following two broad application scenarios, on which we can base the design decisions behind a *CoS*.

Intelligent surroundings (Distributed Applications): Cyber-physical systems embody interacting (cooperating) objects, where based on sensed inputs by one or more devices, a distributed action might be taken. We take the example of a conference room in an office building with several chairs, an LCD screen and lighting infrastructure. We also assume that all these devices (the chairs, the lights and the screen) have sensing and actuation capabilities, a computation platform capable of running a *CoS*, and compatible radio-transceivers. Several applications can be conceived on this infrastructure. For example: *i)* adjusting the brightness and color temperature of the LCD screen based on the distance of the chairs from the screen and properties of the ambient light, *ii)* turning the lights on/off based on the occupancy of the room, and *iii)* turning on the heat in the chairs based on the occupant-preferences and so on. Considering that these devices could be manufactured by different vendors, developing distributed applications is challenging. A *CoS* should provide a conducive environment where applications can be deployed across diverse devices with different capabilities. End-to-end device connectivity, application-code dissemination are some of the basic functionalities that should be provided, and higher level device interactions should be specified by the programmer.

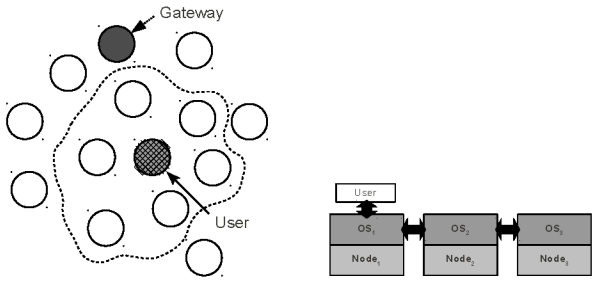
Communicating Vehicles (Dynamic Topologies): A relevant example of connected devices in dynamic topologies is a set of cars at an intersection that are stuck in a traffic jam. In such a scenario, where different vehicles meet at a junction for brief periods, having a centralized middleware or a common abstraction for directing them to carry out a certain goal may or may not be possible. A practical solution can be a modern operating system, that has coordination of dynamic-topologies as a basic functionality, and allows programming this ecosystem in a non-centralized way. A traffic policeman



(a) User interaction through a middleware

(b) Layered diagram showing the user interaction

Fig. 1. Programming with the help of a middleware, to emphasize the centralizing aspect



(a) User interaction through a *CoS*

(b) Layered diagram showing application dissemination using *CoS*

Fig. 2. Programming using a *CoS*

can now simply broadcast a traffic-jam resolving algorithm to all cars for that situation. For example, the policeman can command the cars arriving at the intersection from the north lane to start going into the west lane and so on. The operating system running on the vehicles' will allow the reception of such commands, and take useful action like notifying the driver or driving accordingly if it is an autonomous vehicle.

IV. Design Principles

The motivation behind *CoS* is driven by the observation that the existing operating systems were only designed for facilitating node-level application development, and middleware solutions designed to allow *macro-* or network-level programming are limited in scope and are highly dependent on the underlying operating systems. We discuss some of the principles that stress on the need for a new perspective in the design of operating system for cyber-physical systems.

A. *Programming using* CoS

Traditional network programming approaches typically involve a middleware or a programming abstraction that inevitably tends to centralize the network topology. A user has to interact with a programming layer, that generally resides on a central server or a gateway. As shown in Figure 1, the middleware abstracts the network complexities from a user with the help of a hierarchical setup that can be rigid and highly application specific.

In contrast, *CoS* is designed to facilitate application development in a distributed way for networked objects of the future. The programmer interacts with the operating system directly for creating network-level applications, instead of a middleware or a gateway. *CoS* makes it possible that the user can interact with the system at any logical or topological location in the network, as shown in Figure 2(a). *CoS* manages the communication and dissemination of application program to other nodes, based on the user requirements embedded in the logic of the application. The communication between the nodes can be transparent to the user, and the interaction among them is dictated by the application logic. The user may deploy an application over one or more nodes, each running an instance of *CoS*. Then the application is distributed to other participating nodes by *CoS* as exemplified in Figure 2(b).

B. *Truly Distributed Design*

Traditional distributed operating systems were designed from a perspective of abstracting away the presence of more than one machine from the user. According to Tanenbaum *et al.* in [11]:

> *"As a rule of thumb, if you can tell which computer you are using, you are not using a distributed system. The users of a true distributed system should not know (or care) on which machine (or machines) their programs are running, where their files are stored, and so on."*

This presents a major disconnect from cyber-physical systems, where the devices are not only logically distributed but geographically as well, and more often than not, it is important for the applications to associate the geographical location in their logic. For example, controlling the window blinds based on light level readings in specific rooms *A, B and C*. Most middleware and programming abstractions tend to centralize the network, and cause overheads in maintaining connectivity to all the nodes and may also involve participation of the nodes that may otherwise not be required. A middleware would require a hierarchical architecture to allow deployment of applications. Similar to the example of vehicles at a junction, the nodes may not provide enough support for an active higher layer in dynamic topologies. Hence, the operating system executing on the nodes needs to allow application deployment in a decentralized way, and then execute distributed logic. A distributed operating system for cyber-physical systems (logically and geographically) can decentralize the operation of the network. It should allow a user to deploy applications by connecting to any one or more nodes in the network. *CoS* should obviate the requirement of having a middleware and/or a programming abstraction to program the network.

C. *Other Key Features*

Modular: *CoS* should be modular in design supporting dynamic loading-unloading of modules and application. This can help a programmer create powerful applications with significant ease by making use of existing modules, or creating new ones. Modules can either reside on the system flash memory, or can be delivered over-the-air, if needed.

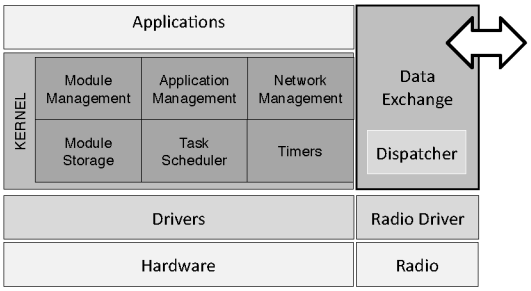


Fig. 3. Typical architecture of a *CoS*

Integrated Network Management: *CoS* should support tightly integrated network management, such that a device is able to discover its neighboring devices and thus allowing updating of routing information. This information should be made available to the programming interface to network-wide application development.

Programming Interface: The traditional way of programming sensor nodes via a direct one-to-one connection to a computer, is designed to facilitate application development while making good use of on-board peripherals. The programming interface of the proposed operating system should be at the network-level by design, rather than being a layer on top of node-level programming abstraction. The programming interface should have information about the network topology and capability of the nodes to provide a global view to the programmer in an intuitive way.

Isolation of Applications: Multiple independent applications on a network of nodes should be supported, while making sure that the data exchange and operation of the applications remains isolated. The data may need to be multiplexed in the network to save energy, and then demultiplexed to deliver to each user.

Support for Heterogenous Platforms: The real-world applications of sensor networks especially in the context of cyber-physical systems may require diverse hardware, including processor, sensors, actuators and communication peripherals. The operating system should be designed such that it supports this varied ecosystem of hardware, and provides suitable programming provisions.

V. *CoS* Architecture

After discussing the design features of *CoS*, we can now describe its architecture. An outline of the architecture showing various components is provided in Figure 3. The following important components constitute *CoS*: Kernel, drivers, exchange plane and the applications on top. We will describe each of those in detail next.

A. Applications

One of the key motivating principles behind the design of *CoS* is easy and convenient deployment of applications directly on top of the *operating system*, rather than having a network-wide middleware or a programming framework. The status-quo in the programming of networked embedded systems involve either copying operating system images with embedded applications onto to the flash, or using a network-level virtual-machine delivery system. Following from the typical trend of writing applications on top of an OS for general-purpose computing systems, *CoS* should allow installing applications at runtime, without the need of a middleware. Given the resource-constrained nature of the embedded systems, a programmer can create and compile applications on a PC, and then deliver the binaries to *CoS*.

As explained earlier, the user may not need to depend on a gateway or a central-server to deploy the application, *CoS* allows any one or more nodes to act as *point of delivery*. The distribution of an application to participating nodes is handled by the data-exchange plane. The kernel provides a *mount-point* to the newly deployed application and adds its information in a local list. The mount-point is a pointer to the memory location where the application resides, so that it can be executed according to the scheduling policy of *CoS*, and the criticality or the priority of the application.

Each application has to specify a scope, both geographic and logical, to help determine the nodes to be associated with that application. In case of more than one application submitted by independent users, *CoS* ensures isolation between the state of the applications both at node- and network-level, thus making sure that the data is delivered to the intended destination in a seamless manner and appropriate action is taken in case of *sense-and-react* applications. This may require conflict resolution or deadlock avoidance support from the kernel. For example, in an intelligent surroundings scenario, if one application requires lights off in the night, and another requires the lights to be turned on if the window shades are down, it may happen that they can be in conflict at some point in time. This conflict has to be resolved among participating applications, and this responsibility lies with the kernel.

B. Kernel

The kernel handles the core functionality, including managing the applications, task-scheduling and timing. Scheduling the applications is one of the most important functions of the kernel. The underlying hardware platform may be significantly resource-constrained that may not support more than a certain number of applications, and the resource usage of applications have to be limited within certain bounds. The kernel ensures that the applications do not misbehave, and their timing requirements are met in the best manner possible. For this purpose, kernel from the NanoRK [3] operating system can be adapted, as it has support for real-time scheduling and task-level resource reservations. In addition to these optimizations, the kernel should be able to resolve conflicts in case of dissimilar requirements of applications. The kernel can make use of priorities, or assign default behaviors to the peripherals. In the example of conflict in the state of lights provided earlier, the kernel may choose to keep the lights off at night, until over-ridden manually.

The kernel should be modular in design so that drivers or other modules can be added to enable required functionalities.

Cyber-physical systems can consist of varied sensor and actuator peripherals, and providing out-of-the-box support for such possibly large number of devices may not be practical. Programmers or users should be able to install modules on the nodes covered by their applications. The kernel should allow dynamic loading and unloading of modules in a manner similar to the SOS [12] operating system. The kernel can achieve this with the help of module management and storage components.

As *CoS* may be run on battery-powered devices, minimizing the power consumption is important. A power-management module tries to put the device to sleep for as long as possible. Nodes may operate at very low duty-cycles, hence the power-management module can ensure that different applications execute in way to maximize the sleep interval.

C. Drivers

Hardware support for the peripherals on a node, including the radio, the sensors and the actuators, is provided through drivers. In addition to the default drivers available with *CoS*, drivers can be loaded as modules at the runtime. Such design allows easy integration of heterogenous devices and dynamic behavior in the long-term. The operating system does not need to be *flashed* again if some peripheral devices are added or removed. In addition to the peripherals, drivers can help applications to configure the communication layer as well. Radio configuration, medium-access control and routing can be implemented as modules and changed on-the-fly, if needed.

D. Exchange Plane

One of most important components of the *CoS* architecture is the data-exchange plane. The data-exchange plane handles all the communication to and from the node. Applications created by the user are delivered to the nodes through this plane, and are further relayed to other nodes that participate in the given application. Other responsibilities of the data-exchange plane are ensuring isolation between the applications, delivering data to the nodes involved, and also directing actuation based on the distributed logic of an application.

The data-exchange plane uses information from the network management module in the kernel about the topology and routing information in order to maintain the communication across a multi-hop network. It can use a device-advertisement phase to construct a topology map of the system. The advertisements allow the exchange-plane to maintain information about the capabilities of the neighboring nodes. The radius of the neighborhood may be pre-decided as a design-parameter or specified by the applications. Developing an application may require knowledge about the capabilities of the devices in the network and hence, the advertisements available to the data-exchange plane should be provided to the programmer so that a distributed logic can be implemented, in accordance with the truly distributed design principle explained in Section IV-B.

The flexibility of *CoS* lies mainly in the configurability of the data-exchange plane and how conveniently a programmer can access and adapt this plane in her application. It allows on-demand information gathering about the devices around and topology formation according to the application needs. For more dynamic network topologies, the maintenance of network information and device advertisements can be more frequent if an application requires so. Otherwise, the network may remain relatively dormant if no application-level updates are required.

VI. Conclusions

We proposed a new paradigm in operating system design called *Co-operating System* or *CoS*, that aims to ease the application development for cyber-physical systems. We argued that the current operating systems like TinyOS, Contiki or Nano-RK are designed with a goal to facilitate the programming of individual nodes in a network of embedded devices. Middleware or network programming frameworks are the other end of the spectrum that may reduce the flexibility of applications and jeopardize the reliability and robustness. Perhaps this is the reason that even with the development of several such solutions, not many have been widely adopted, and researchers still depend heavily on developing applications directly on top of an operating system. We provided the design principles behind *CoS* and discussed its architectural aspects that may enable significant changes in the way applications are developed and distributed for networked embedded systems. It can be argued that *CoS* may not be significantly different from a middleware running on top of a traditional OS in terms of the software-architecture, but the fresh perspective of creating network applications directly on *CoS* can provide a conducive setup for rapid and diverse application development for cyber-physical systems.

References

[1] T. T. 2.x Working Group, "Tinyos 2.0," in *the 3rd international conference on Embedded networked sensor systems*, ser. SenSys '05. San Diego, California, USA: ACM, 2005, pp. 320–320.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *the 29th IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[3] A. Eswaran, A. Rowe and R. Rajkumar, "Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks," *IEEE Real-Time Systems Symposium*, 2005.

[4] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The liteos operating system: Towards unix-like abstractions for wireless sensor networks," in *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, april 2008, pp. 233 –244.

[5] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and V. Bahl, "An operating system for the home (to appear)," in *Proceedings of the 9th USENIX conference on Networked systems design and implementation*, ser. NSDI'12, 2012.

[6] ——, "The home needs an operating system (and an app store)," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets. Monterey, USA: ACM, 2010, pp. 18:1–18:6.

[7] L. Mottola and G. Picco, "Programming wireless sensor networks: Fundamental concepts and state-of-the-art," *ACM Computing Surveys*, 2010.

[8] "http://www.dlna.org/."

[9] "http://www.z-wave.com/modules/zwavestart/."

[10] "http://osian.sourceforge.net/."

[11] A. S. Tanenbaum and R. Van Renesse, "Distributed operating systems," *ACM Comput. Surv.*, vol. 17, pp. 419–470, December 1985.

[12] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *the 3rd international conference on Mobile systems, applications, and services*, ser. MobiSys '05. Seattle, USA: ACM, 2005, pp. 163–176.

Efficient I/O Scheduling with Accurately Estimated Disk Drive Latencies

Vasily Tarasov[1], Gyumin Sim[1], Anna Povzner[2], and Erez Zadok[1]
[1]Stony Brook University, [2]IBM Research—Almaden

***Abstract*—**
Modern storage systems need to concurrently support applications with different performance requirements ranging from real-time to best-effort. An important aspect of managing performance in such systems is managing disk I/O with the goals of meeting timeliness guarantees of I/O requests and achieving high overall disk efficiency. However, achieving both of these goals simultaneously is hard for two reasons. First, the need to meet deadlines imposes limits on how much I/O requests can be reordered; more pessimistic I/O latency assumptions limit reordering even further. Predicting I/O latencies is a complex task and real-time schedulers often resort to assuming worst-case latencies or using statistical distributions. Second, it is more efficient to keep large internal disk queues, but hardware queueing is usually disabled or limited in real-time systems to tightly bound the worst-case I/O latencies.

This paper presents a real-time disk I/O scheduler that uses an underlying disk latency map to improve both request reordering for efficiency and I/O latency estimations for deadline scheduling. We show that more accurate estimation of disk I/O latencies allows our scheduler to provide reordering of requests with efficiency better than traditional LBN-based approaches; this eliminates the need of keeping large internal disk queues. We also show that our scheduler can enforce I/O request deadlines while maintaining high disk performance.

I. Introduction

Modern general-purpose computers and large-scale enterprise storage systems need to support a range of applications with different performance and timeliness requirements. For example, audio and video streams in multimedia applications require timely data delivery guarantees, while concurrent interactive applications remain responsive. In large-scale enterprise storage systems, the rise of storage consolidation and virtualization technologies [1], [2] requires the system to support multiple applications and users while meeting their performance constraints. For example, Internet-based services that share a common infrastructure expect I/O performance for each service in accordance with its service level agreement [3].

Managing disk I/O is an essential aspect of managing storage system performance, as disks remain a primary storage component and one of the top latency bottlenecks. A classic way to improve disk performance is to reorder disk I/O requests, because disk performance largely depends on the order of requests sent to the disk device. With an additional requirement of providing timeliness guarantees, the traditional goal of maximizing overall disk efficiency remains an important requirement. As a result, many real-time disk I/O schedulers [4], [5], [6] combine reordering algorithms (such as SCAN) with real-time scheduling (such as EDF) to optimize disk performance while meeting guarantees. Similarly, fair- or proportional-sharing schedulers reorder some requests to improve disk efficiency.

Since operating systems have a limited knowledge of disks, existing disk I/O schedulers perform reordering based on the requests' Logical Block Number (LBN). I/O schedulers assume that the larger the difference between two LBN addresses, the longer it takes to access the second LBN address after accessing the first one. Although this assumption used to be reasonable in the past, we will demonstrate that it no longer holds and is misleading due to complex specifics of the modern disk drive design. The disk drive itself has an internal queue and a built-in scheduler that can exploit the detailed information about the drive's current state and characteristics. Consequently, built-in disk drive schedulers are capable of performing request scheduling with a higher efficiency than LBN-based schedulers can at the OS level.

Best-effort disk I/O schedulers improve their performance and overcome inefficiencies of LBN-based scheduling by keeping as many requests as possible outstanding at the underlying disk device so they are scheduled by the drive's internal scheduler. However, disk I/O schedulers with real-time guarantees cannot take advantage of the drive's internal scheduler, because the I/O scheduler loses control over requests sent to the drive and the drive's internal scheduler is not aware of the host-level request deadlines. Thus, existing real-time schedulers keep inefficient internal disk queues of only one or two requests, or allow more outstanding requests at the disk drive for soft guarantees, but they require frequent draining of internal disk queues in order to meet request deadlines [7].

If OS would have a more accurate source of information about disk drive latencies, it could perform efficient scheduling while meeting request deadlines. But how large can potential benefits of accurate latency estimation be? In this paper, we first propose a novel request reordering algorithm based on maintaining a disk drive latency map within the OS kernel. The map allows us to accurately estimate the *actual* latency between any pair of LBN addresses anywhere on the disk. We designed and implemented a real-time disk I/O scheduler that meets request deadlines as long as the disk can sustain the required throughput. The scheduler learns the disk latencies and adapts to them dynamically; it uses our request reordering algorithm to maintain high efficiency while meeting request deadlines. Real-time schedulers that use a distribution of I/O execution times over all disk-block addresses, tend to overestimate I/O execution times; in contrast, our disk drive

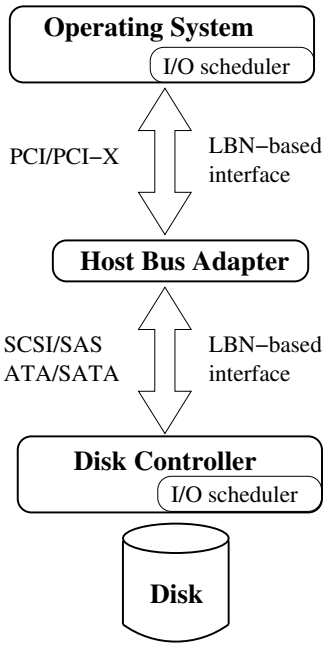


Fig. 1. I/O architecture of commodity servers. The OS accesses the block devices through a standard LBN-based interface that hides the device's physical characteristics.

latency map provides more accurate per-LBN-pair execution time.

We show that while allowing only one request to the underlying disk drive, our reordering algorithm can achieve performance up to 28% better compared to LBN-based schedulers. We also demonstrate that our scheduler enforces request deadlines while providing higher throughput than LBN-based schedulers. We address CPU trade-offs using approximation algorithms and user-defined memory limits. For large datasets our map size can become too large to fit in RAM. Given the benefits of accurate latency prediction as demonstrated in this paper, we expect that various techniques can be used in the future to reduce map sizes and thus provide similar benefits for larger devices.

The rest of the paper is organized as follows. Section II presents experimental results that motivated the creation of our scheduler. In Section III, we describe how latency estimation in our design allows to increase disk throughput for batch applications and enforce deadlines for real-time applications. Section IV details the implementation and Section V evaluates our scheduler against others. In Section VI, we survey related work. We conclude in Section VII and discuss future directions in Section VIII.

II. Background

In this section we describe the overall Input/Output mechanism relevant to the I/O scheduler, deficiencies of this mechanism, and the experiments that led us to create a new scheduler. To the OS I/O scheduler, the disk device appears as a linear array where the Logical Block Number (LBN) is the index

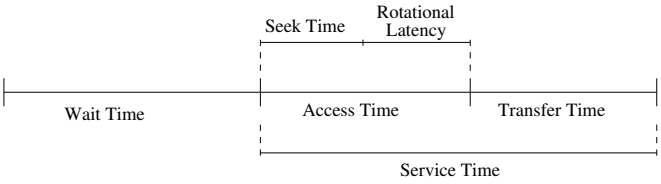


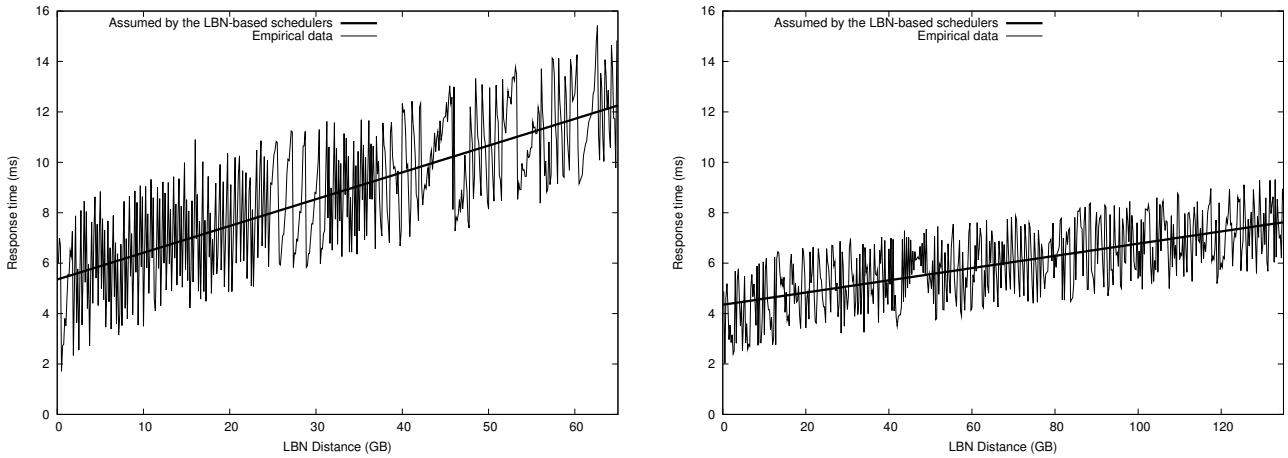
Fig. 2. Decomposition of the request response time.

into this array. Such address representations are used by many I/O protocols (e.g., SATA and SCSI). When the disk scheduler sends a request to the underlying disk device, the Host Bus Adapter (HBA) passes these requests to the disk controller, which in turn maps LBNs to the physical location on the disk. The disk drive has its own internal queue and a scheduler that services requests one by one and then returns completion notifications back to the OS, as seen in Figure 1.

Figure 2 depicts a typical timeline for the request's execution. Response time is the time from the request submission to the I/O scheduler to the request's completion. Response time consists of wait time and service time. Wait time is the time spent in the I/O scheduler's queue. Service time is the time from when the request is picked from the queue for service and until the moment the request completes. Service time consists of access time and transfer time. Access time is required to locate the data; transfer time is the time to transfer the data. For disk drives, the access time consists of the time to position the arm (seek time) and the time until the platter reaches the required position (rotational latency).

Request reordering at the I/O scheduler directly affects access and service times. Indirectly it also affects wait time because shorter service times lead to shorter queues. The OS I/O scheduler knows only about the requests' LBNs and sizes; it is the only criterion OS schedulers can use to perform request scheduling (apart from the knowledge about the request owners, the processes). A common assumption is that the shorter the distance between two LBN addresses is, the smaller is the access time between them. Given this assumption, the scheduler can, for example, sort all requests by their LBNs and submit them in that order (enforcing any required deadlines if necessary).

This assumption used to be true in the early days of disk drives when seek time dominated the rotational latency. Since then, manufacturers significantly improved their disk positioning mechanisms and nowadays rotational latency is of the same magnitude as seek time (e.g., 4msec vs. 2msec for a typical 15K RPM drive). Moreover, the variety of devices and their complexity increased dramatically. ZCAV/ZBR technology, error correction, block remapping, improvements in short seeks, and increased block sizes are just some of many the complex features in modern disk drives. As the number of sophisticated disk technologies grows, the variation among disk models increases [8]. Consequently, one has a large selection of very different devices available on the market. The fact that I/O schedulers still assume a common linear disk drive



(a) 10,000 RPM 3.5" 80GB SCSI Disk

(b) 15,000 RPM 2.5" 146GB SAS Disk

Fig. 3. I/O request response time depends on the LBN distance. The empirical dependency is more complex than the one assumed by common LBN-based schedulers and is unique to specific disk drive models.

model, intuitively, should hurt LBN-based scheduling quality.

We checked how the access time depends on the LBN distance between two data blocks. Figure 3 depicts this dependency for two different devices. The regular I/O scheduler assumes that the access time increases linearly (or at least monotonically) with the LBN distance. However, from the figure we can clearly see that the dependency is not monotonous. The coefficient of linearity is 0.57 and 0.42 for the SCSI and SAS drives, respectively (1.00 corresponds to a perfectly linear dependency). Moreover, the graphs demonstrate that dependencies are different for different models.

An optimal I/O request schedule heavily depends on the specifics of the underlying hardware. Therefore, it seems reasonable to make the storage controller responsible for request scheduling. In fact, both SCSI and SATA standards support command queueing that allow the OS to submit multiple requests to the disk controller, which in turn determines the optimal request sequence [9], [10]. However, there is no way to transfer to the controller the information about desired request deadlines, which are important for real-time applications. According to our measurements, disk controllers can postpone request execution by more than 1.2 seconds if a block address is not on the optimal scheduling path. The situation is worsened by the fact that disk vendors keep their firmwares closed and the user has no control over the scheduling algorithms used within the controllers. Ironically, the only thing that the OS can do to provide more predictable service times is to disable hardware queuing entirely or flush it periodically. But in that case, disk utilization falls dramatically as the OS is unaware of drive's physical characteristics. In this work we propose to augment the OS's I/O scheduler with the knowledge of the drive's physical characteristics. This allows our OS scheduler to enforce deadlines while providing high throughput.

III. Design

Section III-A explains our approach to estimate disk latencies. In Section III-B we explain how our scheduler achieves high throughput. Section III-C describes an additional algorithm that allows the scheduler to enforce deadlines of individual requests.

A. Disk Latency Estimation

A queue of N requests can be ordered in $N!$ different ways. The general task of an I/O scheduler is to pick the order that satisfies two criteria:

1) The order is the fastest when executed by a disk drive; and
2) Individual request response times are within certain limits.

The first criterion provides optimal throughput, and the second one ensures that the deadlines are met. In this paper, we argue that satisfying both criteria is hard and requires an accurate estimation of disk I/O latencies. Assume there is a function $T(o)$ that returns the execution time of some order o of N requests. One can experimentally collect the values of this function and then schedule the requests using it. Here we assume that experimentally collected values are reproducible: i.e., if the same sequence of requests is issued again, its execution time remains the same or sufficiently close. Our measurements indicate that this assumption is true for modern disk drives. For more than a 1,000,000 randomly selected orders, the deviation in execution time was within 1% for 1,000,000 iterations. However, the number of possible orders is so large that it is practically infeasible to collect latencies of all orders.

We can simplify the problem by noticing that $T(o)$ can be calculated as a sum of service times of all requests in the queue:

$$T(o) = \sum_{i=1}^{N} S_i$$

where S_i is the service time of the i-th request. S_i is a function of multiple variables, but for disk drives it depends mostly on two factors: the LBNs of the i-th and the $(i-1)$-th request. This is due to the fact that modern disk drives spend most of their time to locate the data (access time), while transfer time does not contribute much to the service time. Our experiments did not show any difference between read and write service times, but our approach tolerates potential differences [11] by using the worst service time of the two. Large I/O sizes can be addressed by logically dividing a request into smaller sizes.

So, S_i is an approximate function of two variables:

$$S_i \approx Function(LBN_i, LBN_{i-1})$$

At this point it becomes feasible to collect service times for many pairs of requests. This function can be represented as a matrix of $M \times M$ elements, where M is the number of LBN addresses that are covered by the matrix. Ideally, the matrix should cover the disk's sub-region that is accessed more frequently than the rest of the disk. Assuming that the size of on-disk "hot" dataset is 5GB, the matrix granularity is 128KB, and the size of the matrix entry is 4 bits, then the total size of the matrix is around $(5\text{G}/128\text{K})^2 \times 0.5\text{B}/2 = 400\text{MB}$. We divide by two because the matrix is symmetric, so we need to store only half of it. In our experiments, 128KB and 4 bits were enough to demonstrate significant improvements.

The matrix needs to reside in RAM or at least in a fast Flash memory; otherwise, the OS has to perform additional reads from the disk drive just to schedule an I/O request and this would make scheduling completely inefficient. 400MB is a significant amount of memory, but if one spends this 400MB on caching the data, the throughput improvement is only around 8% (400MB/5GB) for a random workload. Our scheduling can improve throughput by up to 28%, as shown in Section V. Therefore, spending expensive memory on just a cache is not justified in this case and it is wiser to use RAM for such a matrix.

A single matrix can be shared across an array of hundreds of identical disk drives (e.g., in a filer), which saves significant amount of RAM. Furthermore, some workloads prohibit caching, e.g., database writes are usually synchronous. In this case, all available RAM can be devoted to the matrix. E.g., a machine with 32GB of RAM and 100 disks can cover a dataset of 4.5TB size ($(4,500\text{G}/100/128\text{K})^2 \times 0.5\text{B}/2 = 32\text{GB}$).

However, this matrix approach scales poorly with the size of the *single* device: if a disk drive contains M blocks then the size of the matrix covering the whole disk is proportional to M^2. Another way to obtain the value of S_i is to model the drive and get S_i as the output of this model. In this case memory consumption can be significantly reduced but several other concerns emerge: how accurate is the model, how universal is it, and how high is the modeling CPU overhead. The accuracy of the model is crucial for optimal scheduling. We used DiskSim [12] to emulate several disk drives and compared the access times it predicted with the values from a pre-collected matrix. We found out that the accuracy was within 5% for 99.9% of all blocks. Our measurements also showed that CPU consumption increases by 2–3% when DiskSim is used, which we believe is a reasonable trade-off for the memory we saved.

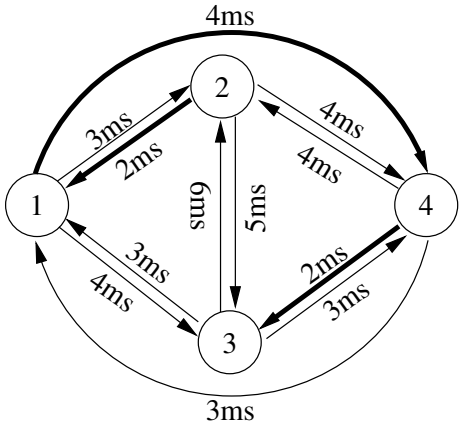


Fig. 4. Request scheduling problem as a TSP problem.

Hardware schedulers achieve high throughput using efficient reordering. Internally they use mathematical models that prove that it is possible to predict latency accurately; our work demonstrates how much benefits one can get provided that there is an accurate source of latency prediction. Our approach works for small devices, but modeling can extend it to larger ones.

B. Achieving High Throughput

When a service time for 2 consequent requests is available, the problem of scheduling to optimize disk efficiency resembles the well-known Traveling Salesman Problem (TSP). For example, assume that there are 4 requests in the queue and they are enumerated in the order they came from the applications; see Figure 4. Each request can be thought of as a vertex in a graph. The edges of the graph represent the possibility of scheduling one request after the other. As any order of requests is possible, all vertices are connected to each other (a fully connected graph). An edge's weight represents the time to service one request after the other. To find the optimal order of requests one needs to find the shortest path that covers all vertices. From Figure 4 we can see that although the requests come in the order 1-2-3-4, it is more optimal to execute them in the order 2-1-4-3, because it constitutes the shortest path.

It is well known that finding an exact solution to a TSP is exponentially hard. We initially implemented a scheduler that solves TSP exactly, but as expected it did not scale well with the queue size. There are a number of approximation algorithms that solve TSP. We picked an algorithm that is quick, easy, and provides reasonable precision, the Nearest Insertion Algorithm [13]. It works as follows:

1) Initialize the Shortest Path Edges set SPE and the Shortest Path Vertices set SPV to the empty sets.
2) For each graph vertex V_i that is *not* in the SPV:
   a) For each edge V_jV_k in the SPE set, calculate the SPE path increase $I_{V_jV_k}$ if the edge V_jV_k is

replaced by the V_jV_i and V_iV_k edges:

$$I_{V_jV_k} = W_{V_jV_i} + W_{V_iV_k} - W_{V_jV_k}$$

where W_{edge} is the weight of the $edge$.

b) For boundary vertices V_{b1} and V_{b2} in the SPV set (i.e., the vertices that have less than two adjacent edges in the SPE), calculate the path increases I_{b1} and I_{b2} if edges V_iV_{b1} and V_jV_{b2} are added to SPE, in order:

$$I_{b1} = W_{V_iV_{b1}},\ I_{b2} = W_{V_jV_{b2}}$$

Only one boundary vertex exists in the very first cycle of this loop.

c) Pick the smallest one among $I_{V_jV_k}$, I_{b1}, I_{b2} and add the corresponding edge (V_jV_i, V_iV_{b1}, or V_jV_{b2}) to the SPE set.

d) Add V_i to the SPV set.

3) When all graph vertices are in SPV, the SPE set contains an approximate solution of the TSP.

The complexity of this algorithm is $O(N^2)$, which is reasonable even for a relatively long queue. Queues that are longer than 256 requests are rare in real servers because they dramatically increase the wait time [14]. The worst case approximation ratio of the described algorithm is 2 (i.e., the resulting path might be twice longer than the optimal one). Although there are algorithms with better approximation ratios, they are much more difficult to implement, their running time is worse for small graphs, and they are often not space-efficient. The Nearest Insertion Algorithm is considered to be the most applicable one in practice [15].

In order for a TSP solution to be optimal for a real disk, an accurate service time matrix is required. Our scheduler does not require a special tool to collect this information. It collects the matrix as it runs, inferring the latency information from the requests submitted by the applications. Initially the map is empty and the scheduler orders the requests so that it can fill empty cells. For example, if there are requests $rq1$, $rq2$, and $rq3$ in the queue and the map contains information about the service time for only the $(rq1, rq2)$ and $(rq2, rq3)$ pairs, but no information about the $(rq1, rq3)$ pair, our scheduler schedules the request $rq3$ after $rq1$. As more and more matrix cells are filled with numbers, and the model becomes more precise, scheduling becomes more efficient. We demonstrate this behavior in Section V.

C. Deadlines Enforcement

More accurate estimation of disk I/O latencies allows our reordering algorithm to provide high performance without the need to keep large internal disk queues. This allows a real-time scheduler using such a reordering scheme to tightly control request service times while maintaining high efficiency. This section describes how we extended our I/O scheduler described earlier to support the notion of deadlines and guarantee request completion times as long as device's throughput allows that.

Let us formulate this problem in terms of the graph theory as we did in the previous section. Again, we have a fully connected graph with N vertices which represent requests in the queue. All edges of the graph are weighted in accordance with the service time, $W_{V_iV_j}$. In addition to that there is a deadline D_{V_i} for each vertex V_i. Deadlines are measured in the same time units as the weights of the edges. A traveling salesman should visit every vertex in the graph and it is important for him to be in certain vertices within the certain deadlines.

Assume that the salesman has picked some path through the vertices. Then it will visit each vertex V_i at specific time C_i (completion time). We call the *overtime* O_{V_i} the time by which the salesman was late at vertex V_i:

$$O_{V_i} = max(0, C_{V_i} - D_{V_i})$$

The problem of TSP with deadlines, or I/O scheduling with guarantees, is to find a sequence of vertices $V_1V_2...V_i...V_N$, such that:

$$\max_i(O_{V_i})) \rightarrow min \quad (1)$$

$$\sum_{i=1}^{i=N-1} W_{V_iV_{i+1}} \rightarrow min \quad (2)$$

Equation (1) expresses the fact that no overtime, or minimal overtime, should be found. Equation (2) states that the path should be minimal. The order of these requirements is important: we first guarantee minimal overtime; then, among the remaining solutions, we pick the one that provides the shortest path. If the system is not overloaded, overtime will be zero for all vertices, so the algorithm enforces hard deadlines. Notice, however, that this is under the assumption that estimated latency matrix is accurate enough. Our scheduler keeps updating the values in the matrix as it works. It stores only *the worst* time it has ever observed for a pair of LBN addresses. This allows to enforce deadlines with a very high probability. According to our experiments, after 100 measurements the probability to observe an even worse service time is less than $10^{-6}\%$. Storing only the worst time also addresses potential problems with the very fast accesses to the disk cache hits. Updating the values in the matrix makes our scheduler adaptive to the changes in the device characteristics, which happens, for example, when bad blocks are remapped.

This problem is proven to be NP-complete [16]. We developed an approximation algorithm to solve it. The classic method to solve deadline scheduling is the Earliest Deadline First (EDF) algorithm. It simply executes requests in the deadline order. EDF provides minimal overtime, but does not take into account service time variation among requests and consequently does not find an optimal throughput (i.e., it does not pick the shortest path in terms of the graph). Somasundara et al. solved a similar problem for mobile-element scheduling in sensor networks [16]. However, there are two important differences between mobile-element scheduling and I/O request scheduling. First, every I/O request should be eventually serviced even if it cannot meet its deadline. Second, once the request is serviced, it is removed from the original set and there is no deadline update. We merged the EDF and

Disk Model	Interf.	Cap. (GB)	RPM	Avg Seek (ms)	HBA
3.5" Maxtor 6Y200P0	PATA	200	7,200	9.3	ServerWorks CSB6
3.5" Seagate ST380013AS	SATA	80	7,200	8.5	Intel 82801FB
3.5" Seagate ST373207LW	SCSI	73	10,000	4.9	Adaptec 29320A
2.5" Seagate ST9146852SS	SAS	146	15,000	2.9	Dell PERC 6/i

TABLE I
DEVICES USED IN THE EVALUATION

the k-*lookahead* algorithm by Somasundara et al. to provide deadline enforcement in our scheduler. Our algorithm operates as follows:

1) Sort vertices by the deadline in the ascending order.
2) Pick the first k vertices from the sorted list L.
3) For all permutations $P_l = (V_1 V_2 ... V_k)$ of k vertices:
   a) Calculate the maximum overtime M_{P_l} for P_l:
   $$M_{P_l} = \underset{1..k}{max}(O_{V_i})$$
   b) Calculate the sum of weights S_{P_l} for P_l:
   $$S_{P_l} = \sum_{i=1}^{i=k-1} W_{V_i V_{i+1}}$$
4) Among all P_l ($l = 1..k!$), pick the permutations that minimize M_{P_l}.
5) If there are multiple permutations with the same minimal M_{P_l}, then pick the case for which S_{P_l} is minimal.
6) Add the first vertex in the selected permutation P_l to the final sequence F and remove this vertex from the sorted list L.
7) Repeat Steps 2–6 if there are still vertices in L.
8) At the end, the final sequence F contains an approximate solution of the problem.

This algorithm looks through permutations of k vertices and picks the next vertex to follow among them. Because k nodes make $k!$ permutations, the overall running time of the algorithm is $O(Nk!)$. The approximation ratio in this case is $O(N/k)$. There is a trade-off in selecting the value of the constant k. If one sets k to a large value, the precision of the algorithm becomes higher (i.e., when $k = N$ the solution is absolutely optimal). However, CPU consumption grows with k. Our experiments showed that the increase of k value beyond 4 does not yield benefits, so our scheduler sets k to 4 by default, but this can be tuned.

IV. IMPLEMENTATION

We implemented our matrix-based schedulers in Linux kernel version 2.6.33. Linux has a convenient plugable interface for I/O schedulers, so our code conveniently resides in a single C file of less than 2,000 LoC. We also wrote several tools for matrix analysis which total to about 1,000 LoC. All sources can be downloaded from the following URL: https://avatar.fsl.cs.sunysb.edu/groups/mapbasedioscheduler/. Our scheduler exports a number of tunable parameters through the sysfs interface:

```
# ls /sys/block/sdb/queue/iosched/
latency_matrix
deadlines
lookahead_k
timesource
```

Servers are occasionally rebooted for maintenance and because of the power outages. We incorporated the ability to save and restore the matrix in our disk scheduler through the `latency_matrix` file. This feature also helps the user to shorten the scheduler's learning phase. If one has a matrix for some disk drive, then it makes sense to reuse it on the other machines with identical drives. If there are differences in the device characteristics, they will be automatically detected by the scheduler and the matrix will be updated accordingly.

Linux has system calls to assign I/O priorities to processes and we used those to propagate deadline information to our scheduler. The mapping between priority levels and deadlines is loaded through the `deadlines` file. The parameter k for the lookahead algorithm is set through the `lookahead_k` file. We used two time sources: a timer interrupt and the RDTSC instruction. One can set up the time source through the `timesource` file. In our experiments the RDTSC time source provided better results due to its higher precision.

We also implemented several other LBN-based scheduling algorithms to evaluate our scheduler against. We provide details in Section V-B. To experiment with hardware scheduling we modified a few HBA drivers so that one can explicitly set the size of the hardware's internal queue.

V. EVALUATION

Section V-A details the hardware we used. In Section V-B we summarize the schedulers we evaluated. In Section V-C we demonstrate that accurately estimated latency map allows to achieve better throughput than LBN-based OS I/O schedulers. Finally, in Section V-D we show that hardware scheduling does not provide adequate deadlines support, while our scheduler does.

A. Devices

We experimented with several disks to evaluate our scheduler. Table I lists the key characteristics of the drives and HBAs we used. We picked devices that differ by interface, form factor, capacity, and performance to ensure that our scheduler is universal. We present the results for the SCSI disk only, but our experiments found approximately the same degree of improvements and the behavioral trends on all of the listed devices.

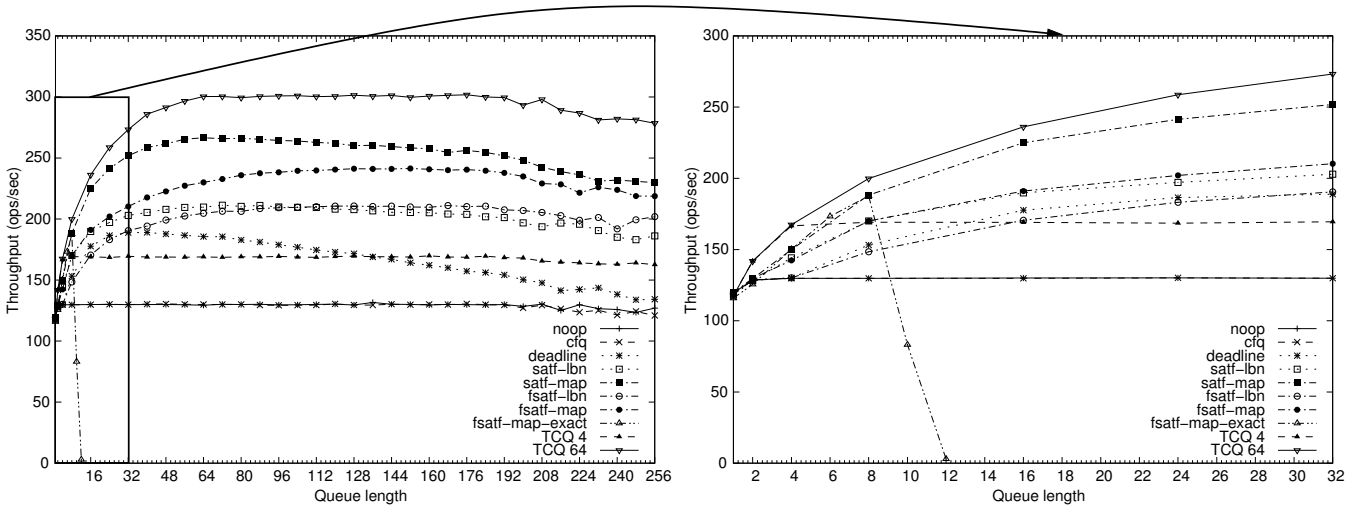


Fig. 5. Throughput depending on the queue length for different schedulers. We show the graph for 1–256 requests on the left. We zoom into the 1–32 range of requests on the right.

B. Schedulers

We picked several schedulers to compare against ours:

1) We implemented a SATF-LBN scheduler that calculates access times purely by the LBN distance and uses SATF (Shortest Access Time First) policy for scheduling. SATF is known to provide the best possible throughput among all scheduling policies [17].
2) The FSATF-LBN scheduler is identical to SATF-LBN but it freezes the queue during each dispatch round, meaning that the new incoming requests are sorted in a separate queue (FSATF). This algorithm is important because unlike SATF, it prevents postponing requests indefinitely.
3) The first variation of our scheduler, SATF-MAP, implements the algorithm described in Section III-B. It finds the shortest path using the latencies stored in the map.
4) The second variation of our scheduler, FSATF-MAP, is identical to SATF-MAP but uses a FSATF freeze policy.
5) The FSATF-MAP-EXACT scheduler solves the corresponding TSP problem exactly, without approximation.
6) NOOP is a slightly modified version of Linux's NOOP scheduler that uses pure FCFS policy (without modifications it performs sorting by LBN addresses). It serves as a baseline for the results of the other schedulers.
7) TCQ4 is hardware scheduling with a queue length of 4. We confirmed that the selection of the OS I/O scheduler does not matter in this case. In fact, the hardware completely ignores the request order enforced by the OS and applies its own ordering rules.
8) TCQ64 is the same as TCQ4, but the queue length is 64.
9) For the sake of completeness we also include the results of CFQ and DEADLINE schedulers. The Linux CFQ scheduler is the default one in most Linux distributions, so its performance results would be interesting for a lot of users. The Linux DEADLINE scheduler is often recommended for database workloads. It uses the SCAN policy but also maintains an expiration time for each request. If some request is not completed within its expiration time, the scheduler submits this request immediately, bypassing the SCAN policy.

C. Throughput

In this section, we evaluate the efficiency of our reordering algorithm. We used Filebench to emulate random-like workloads [18]. Filebench allows us to encode and reproduce a large variety of workloads using its rich model language. In this experiment, N processes shared the disk and each process submitted I/Os synchronously, sending next I/O after the previous one completed. We varied the number of processes from 1 to 256, which changed the scheduler's queue size accordingly (since each process had one outstanding request at a time). Processes performed random reads and writes covering the entire disk surface.

To speed-up the benchmarking process (in terms of matrix collection), we limited the number of I/O positions to 10,000. We picked positions randomly and changed them periodically to ensure fairness. We set the I/O size to 1–4KB, which corresponds to Filebench's OLTP workload. We set the matrix granularity to 128KB and collected performance numbers when the matrix was filled.

Figure 5 depicts how the throughput depends on the queue length for different schedulers. The left figure shows the results for the queue lengths from 1–256 and the right one zooms into the results for 1–32 queue lengths.

The NOOP scheduler's throughput does not depend on the queue length and is equal to the native throughput of the disk: slightly higher than 130 IOPS. This performance level corresponds to the situation when no scheduling is done. CFQ's throughput is identical to NOOP's, because each request is submitted synchronously by a separate process (as it is common in database environments). CFQ iterates over the list of processes in a round-robin fashion, servicing only the

requests corresponding to the currently selected process. If there is only a single request from a currently selected process, CFQ switches to the next process. For synchronous processes this effectively corresponds to NOOP: dispatch requests in the order they are submitted by the applications. The DEADLINE scheduler uses the SCAN policy based on the LBN distance. Consequently, its throughput is up to 50% higher compared to NOOP and CFQ. However, when requests pass a certain expiration time (500ms by default), it starts to dispatch requests in FCFS order. This is seen in the graph: after a certain queue length, the line corresponding to DEADLINE starts to approach NOOP's line.

As expected, the SATF-LBN and FSATF-LBN schedulers exhibit better throughput compared to the previously discussed schedulers. Specifically, SATF-LBN's throughput is the best one that scheduling algorithms can achieve if they use LBN distance solely as the access-time metric. For queues shorter than 100 requests, SATF-LBN outperforms FSATF-LBN, because SATF-LBN inserts requests in the live queue, allowing more space for optimal reordering. However, with longer queues, SATF-LBN needs to perform more sorting than FSATF-LBN, and this causes SATF-LBN to perform worse.

The SATF-MAP scheduler, which implements the same SATF policy as SATF-LBN, but uses the matrix to solve the problem, performs up to 28% better. This is where the value of the latency matrix is seen: the better knowledge of the access times allows us to perform more optimal scheduling.

We implemented a FSATF-MAP-EXACT scheduler that finds the exact optimal solution of the TSP problem. As expected, its performance did not look appealing. When the queue length reaches 8 requests, its throughput drops rapidly because of the exponential complexity of the algorithm. Approximate solutions to the TSP problem performed well. The FSATF-MAP scheduler was up to 17% better than FSATF-LBN, and 13% better on average.

Finally, hardware-implemented algorithms' performance depends on the controller's internal queue length. TCQ4 provides higher throughput compared to not scheduling requests at all (NOOP), but does not outperform other OS-level I/O schedulers. TCQ64's throughput is higher than any other scheduler. Our scheduler cannot reach this level because there is an inherent overhead in submitting one request at a time compared to giving multiple requests to the scheduler at once. We believe this can be addressed by allowing the controller to accept multiple requests at a time, but forcing it to preserve the order. Although it should be possible by setting a *ordered queue tag* on every request in the SCSI queue, it did not work for our controller. Block trace analysis revealed that the controller ignores this flag. Although TCQ64 provides high throughput, it is impossible to guarantee response times. In Section V-D we discuss this problem in more details.

To demonstrate the adaptive nature of our scheduler, we collected the behavior of its throughput over time (Figure 6). It is noticeable that in the beginning of the run the scheduler's performance is relatively low because it picks the request orders that lead to completion of the matrix, not the orders that

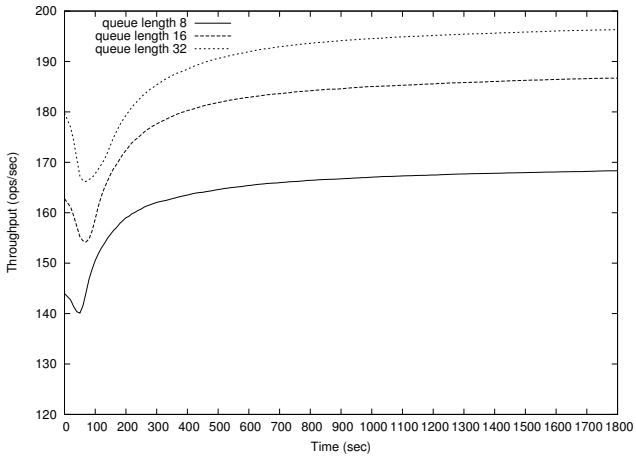


Fig. 6. Adaptive behavior of our scheduler: performance improves with time. Note: the Y axis starts at 120 ops/sec.

optimize throughput. As the matrix gets filled with the latency numbers, throughput starts to improve. Interesting is a hollow in the beginning of the graph. This happens because when the matrix is mostly empty, the scheduler reorders requests by LBN while still trying to extract the required information, which hurts performance temporarily. As time passes, the number of request sequences that have ordered LBNs (and still are not in the matrix) decreases, which leads to the drop in the throughout. After the matrix is filled with missing values, the throughput starts to grow and surpasses the original number.

Thus far we discussed random workloads, but what about workloads that are more sequential? Sequential workloads are characterized by larger request sizes or requests that arrive in serial order. When the request size is around 400KB, transfer times reach the same magnitude as access times. If two subsequent requests are adjacent to each other, access time becomes almost zero. All that make I/O reordering less effective because it only optimizes access time. This is true for all disk schedulers, including ours. However, our scheduler does not hurt performance of sequential workloads, so keeping the scheduler enabled for sequential or mixed workloads is completely valid.

D. Deadlines Enforcement

To show that our scheduler from Section III-C can efficiently enforce request deadlines, we designed the following experiment in Filebench [18]. We created 8 low-priority and 8 high-priority threads. All requests submitted by the low-priority threads were assigned 200ms deadline. We set the deadline for requests from the high-priority threads to 100ms. We collected the maximum response time and throughput for each thread at the user level. Table II shows the results for four schedulers: Earliest Deadline First (EDF), Guaranteed matrix (G-MATRIX), and hardware schedulers with queue sizes 4 and 64. We present the results of hardware schedulers to demonstrate that they cannot guarantee response times,

Sched.	Max response (ms)		Aggr. Thrpt. (ops/sec)
	100ms deadline	200 ms deadline	
EDF	99	198	130
G-MATRIX	86	192	172
TCQ4	407	419	169
TCQ64	955	1,272	236

TABLE II
RESPONSE TIMES FOR LOW (200MS) AND HIGH (100MS) PRIORITY THREADS

though show a good throughput. We do not present the results for other OS I/O schedulers because those schedulers were not designed to enforce deadlines, so their poor results are expected.

EDF enforces deadlines well: maximum response time is always lower than the deadline. This is how it should be because EDF is the most optimal algorithm if the overtime is the only optimization criterion. However, it does not optimize for throughput. Our G-MATRIX scheduler also enforces the deadlines, but its throughput is 32% higher. The hardware schedulers ignore the order in which the OS submits requests and violates the deadlines. As you can see from the table, the maximum response time for TCQ4 is twice over the deadline and 6–9 times worse for TCQ64. By breaking the deadlines, TCQ64 significantly improves throughput (by 37% compared to G-MATRIX). We explained this phenomenon earlier in Section V-C.

VI. RELATED WORK

Improving disk efficiency was a main focus of early disk scheduling algorithms, that observed that ordering of I/O requests can significantly improve disk performance. Most of the algorithms were designed to minimize the disk head movement to reduce seek times, such as SCAN [19] and Shortest Seek Time First (SSTF), while other algorithms tried to minimize total positioning delays by minimizing rotational latencies as well as seek times [14], [20]. Coffman et al. analyzed the Freezing SCAN (FSCAN) policy compared to FCFS, SSTF, and SCAN [21]. The FSCAN scheduler freezes the request queue before the start of each arm sweep. This improves response time and fairness compared to SCAN. Hefri performed extensive theoretical and simulation-based analysis of FCFS and SSTF, showing that SSTF exhibits the best throughput under almost all workloads but its response time variance can be large, delaying some requests by a substantial amount of time [17]. Geist et al. presented a parameterized algorithm that represents a continuum of scheduling algorithms between SSTF and SCAN [22].

All the studies mentioned above assumed that a scheduling algorithm has access to the detailed physical characteristics and current state of the drive. Since modern hard drives hide their internal details and expose only a limited Logical Block Number (LBN) interface, these algorithms had to be implemented in the disk controller's firmware, which is only possible by drive vendors. Our scheduling approach brings detailed device characteristics to the upper layer so that better scheduling can be performed by the OS. Due to the closed-source nature of disk drives' firmware, researchers mostly used simulators (such as DiskSim [12] or specially written ones) or theoretical calculations to demonstrate the performance of their algorithms. All the experiments in this paper, however, were conducted on real hardware without emulation.

Other disk schedulers optimized disk performance by using LBN-based approximation of seek-reducing algorithms [23]. Linux is the richest OS in terms of I/O schedulers, and it includes noop, anticipatory, deadline, and CFQ schedulers. All of these schedulers rely on the regular LBN interface [24]. Our reordering scheme is based on more accurate information about disk I/O latencies and it is more efficient than LBN-based approaches.

Many real-time schedulers aim to optimize performance while meeting real-time guarantees by combining a reordering algorithm to optimize disk efficiency, such as SCAN, with Earliest Deadline First (EDF) real-time scheduling. Some of them use LBN-based reordering [6], [5] and others rely on the detailed knowledge of the disk [25]. Our approach for accurate I/O latency estimation and our reordering scheme is complementary to many of these scheduling algorithms, and can be used to improve overall disk efficiency in these schedulers.

Reuther et al. proposed to take rotational latency into account for improving performance of their real-time scheduler [4]. The authors used a simplified disk model and as a result they were only able to calculate *maximum* response times. The implementation Reuther et al. had was for the Dresden Real-Time Operating System [26]. Our implementation is for much more common Linux kernel and consequently we were able to compare our scheduler to other schedulers available in Linux. Michiels et al. used the information about disk zones to provide guaranteed throughput for applications [27]. However, they were mainly concerned about throughput fairness among applications, not response time guarantees. Lamb et al. proposed to utilize the disk's rotational latency to serve I/O requests from background processes [28]. This is another way to increase disk utilization, providing high throughput while enforcing response time deadlines [29].

Yu et al. conducted the study similar to ours [30]. They examined the behavior of several Linux I/O schedulers running on top of a command-queuing capable device. Their analysis provided the evidence of possible redundant scheduling, I/O starvation, and difficulties with prioritizing I/O requests when command queuing is enabled. The authors also proposed a mechanism for overcoming these problems. Their idea is to switch command queuing on and off depending on the value of a specially developed metric. The disadvantage of their approach is that the metric has a number of tunable parameters which are difficult to set appropriately. Moreover, the appropriate values depend on the hardware specifics. Conversely, our approach is simpler and more general: it moves all scheduling decisions to the OS level, works for virtually any device and performs all tuning automatically. The CPU usage of

our solution is negligible because we use computationally lightweight approximation algorithms, and our memory usage can be limited by the administrator.

VII. Conclusions

Hardware schedulers are capable of providing excellent throughput but a user cannot control response times of individual requests. OS schedulers can strictly enforce response time deadlines but their throughput is significantly lower than what a disk can provide. The ability to estimate disk latencies at the OS level allows to achieve higher throughput while enforcing the deadlines. We designed and implemented an I/O scheduler that collects a matrix of service times for and underlying disk drive. It then performs request scheduling by finding an approximate solution of a corresponding TSP problem. The design of our scheduler incorporates a number of trade-offs between CPU usage, memory usage, universality, and simplicity. The scheduler does not require a distinct learning phase: it collects hardware information on the fly and performs better as more information becomes available. We successfully tested our scheduler on a variety physical disks and showed it to be up to 28% more efficient than other schedulers. Compared to hardware level scheduling solutions, our scheduler enforces deadlines as requested by the processes.

VIII. Future Work

We plan to work towards memory footprint reduction in our scheduler. We believe that pattern recognition techniques are especially promising in this respect because latency matrices contain a lot of fuzzy patterns which regular compression algorithms cannot detect. We plan to work on extending our scheduler to a wider set of devices, specifically solid-state drives and hybrid devices. We expect that matrix design will require modifications to reflect storage devices with significantly different hardware characteristics. Virtualization is another direction for future work, because virtualization layers tend to further perturb, or randomize, I/O access patterns. We successfully tested a prototype of our scheduler inside a virtual machine and the scheduler was capable of detecting significant latency differences in the underlying storage.

References

[1] A. Gulati, C. Kumar, and I. Ahmad, "Storage workload characterization and consolidation in virtualized environments," in *Proceedings of 2nd International Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[2] D. Sears, "IT Is Heavily Invested in ERP, Application Consolidation Rising," 2010, www.eweek.com/c/a/IT-Management/IT-Is-Heavily-Invested-in-ERP-Application-Consolidation-Rising-244711/.

[3] C. Li, G. Peng, K. Gopalan, and T. cker Chiueh, "Performance guarantee for cluster-based internet services," in *in The 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[4] L. Reuther and M. Pohlack, "Rotational-position-aware real-time disk scheduling using a dynamic active subset (das)," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, 2003.

[5] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn, "Efficient guaranteed disk request scheduling with fahrrad," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08. New York, NY, USA: ACM, 2008, pp. 13–25. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352595

[6] A. L. N. Reddy and J. Wyllie, "Disk scheduling in a multimedia i/o system," in *Proceedings of the first ACM international conference on Multimedia*, ser. MULTIMEDIA '93. New York, NY, USA: ACM, 1993, pp. 225–233. [Online]. Available: http://doi.acm.org/10.1145/166266.166292

[7] M. J. Stanovich, T. P. Baker, and A.-I. A. Wang, "Throttling on-disk schedulers to meet soft-real-time requirements," in *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 331–341. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2008.30

[8] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer, "Benchmarking file system benchmarking: It *is* rocket science," in *Proceedings of HotOS XIII:The 13th USENIX Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011.

[9] B. Dees, "Native Command Queuing - Advanced Performance in Desktop Storage," in *Potentials*. IEEE, 2005, pp. 4–7.

[10] "Tagged Command Queuing," 2010, http://en.wikipedia.org/wiki/Tagged_Command_Queuing.

[11] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *IEEE Computer*, vol. 27, pp. 17–28, 1994.

[12] J. S. Bucy and G. Ganger, *The DiskSim Simulation Environment Version 3.0 Reference Manual*, 3rd ed., January 2003, www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-CS-03-102.pdf.

[13] D. Rosenkrantz, R. Stearns, and P. Lewis, "Approximate Algorithms for the Traveling Salesperson Problem," in *15th Annual Symposium on Switching and Automata Theory (SWAT 1974)*. IEEE, 1974, pp. 33–42.

[14] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," in *Proceedings of the Winter Usenix*, 1990.

[15] A. Frieze, G. Galbiati, and F. Maffioli, "On the Worst-case Performance of Some Algorithms for the Asymmetric Traveling Salesman Problem," *Networks*, 1982.

[16] A. Somasundara, A. Ramamoorthy, and M. Srivastava, "Mobile Element Scheduling for Efficient Data Collection in Wireless Sensor Networks with Dynamic Deadlines," in *Proceedings of the 25th IEEE Real-Time Systems Symposium*. IEEE, 2004, pp. 296–305.

[17] M. Hofri, "Disk Scheduling: FCFS vs. SSTF revisited," *Communication of the ACM*, vol. 23, no. 11, November 1980.

[18] "Filebench," http://filebench.sourceforge.net.

[19] P. Denning, "Effects of Scheduling on File Memory Operations," in *Proceedings of the Spring Joint Computer Conference*, 1967.

[20] D. Jacobson and J. Wilkes, "Disk Scheduling Algorithms based on Rotational Position," Concurrent Systems Project, HP Laboratories, Tech. Rep. HPLCSP917rev1, 1991.

[21] E. Coffman, L. Klimko, and B. Ryan, "Analysis of Scanning Policies for Reducing Disk Seek Times," *SIAM Journal on Computing*, vol. 1, no. 3, September 1972.

[22] R. Geist and S. Daniel, "A Continuum of Disk Scheduling Algorithms," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, February 1987.

[23] B. Worthington, G. Ganger, and Y. Patt, "Scheduling Algorithms for Modern Disk Drives," in *Proceedings of the ACM Sigmetrics*, 1994.

[24] J. Axboe, "Linux Block I/O — Present and Future," in *Proceedings of the Ottawa Linux Symposium*, 2004.

[25] H.-P. Chang, R.-I. Chang, W.-K. Shih, and R.-C. Chang, "Gsr: A global seek-optimizing real-time disk-scheduling algorithm," *J. Syst. Softw.*, vol. 80, no. 2, pp. 198–215, February 2007. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2006.03.045

[26] H. Hrtig, R. Baumgartl, M. Borriss, C. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schnberg, and J. Wolter, "Drops: Os support for distributed multimedia applications," in *Eighth ACM SIGOPS European Workshop*, 2003.

[27] W. Michiels, J. Korst, and J. Aerts, "On the guaranteed throughput of multi-zone disks," *IEEE Transactions on Computers*, vol. 52, no. 11, November 2003.

[28] C. R. Lumb, J. Schindler, and G. R. Ganger, "Freeblock scheduling outside of disk firmware," in *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*. Monterey, CA: USENIX Association, January 2002, pp. 275–288.

[29] Y. Zhu, "Evaluation of scheduling algorithms for real-time disk i/o," 2007.

[30] Y. Yu, D. Shin, H. Eom, and H. Yeom, "NCQ vs I/O Scheduler: Preventing Unexpected Misbehaviors," *ACM Transaction on Storage*, vol. 6, no. 1, March 2010.

A Dataflow Monitoring Service Based on Runtime Verification for AUTOSAR OS: Implementation and Performances

Sylvain Cotard*
Renault S.A.S.
1 Avenue du Golf
78280 Guyancourt, France
Email : sylvain.cotard@renault.com

Sébastien Faucou*, Jean-Luc Béchennec†
LUNAM Université. Université de Nantes*, CNRS†
IRCCyN UMR CNRS 6597
44321 Nantes, France
Email: sebastien.faucou@univ-nantes.fr,
jean-luc.bechennec@irccyn.ec-nantes.fr

***Abstract*—We have recently proposed a data flow monitoring service based on runtime verification for AUTOSAR [1]. This service aims at detecting unexpected communication patterns in complex in-vehicle embedded multitask real-time software. The service uses monitors automatically generated from formal models of the system and of the expected communication patterns. These monitors are statically injected in the kernel of Trampoline, an open source RTOS based on AUTOSAR OS specification. In this paper, we present the implementation of the service and its performances. We propose and evaluate several memory optimization techniques aiming at minimizing the memory footprint of the monitors.**

I. Introduction

AUTOSAR (AUTomotive Open System ARchitecture) is a system architecture standard promoted by car makers, equipment suppliers, and software and hardware providers. Release 4 of the standard introduces a support for multicore processors. Such processors have arisen in the embedded market in recent years and are currently promoted by microcontroller manufacturers such as Freescale, Infineon, TI etc.. Despite their undeniable advantages in terms of performance, multicore architectures bring a set of problems that may jeopardize the dependability of embedded applications by increasing their complexity. The main motivation of this work is consequently to support real-time software developpers to maintain a high level of dependability while using these new architectures.

AUTOSAR software systems are composed of communicating real-time tasks. The global behavior of such a software system depends on the individual behaviors of the tasks and on the relative order of the communication operations performed by each task. This order results from the scheduling of the tasks. One problem brought by multicore architectures is the difficulty to accurately predict all the possible schedules of a system. Two reasons can explain this difficulty. First, many results established in the context of uniprocessor real-time scheduling do not scale to multicore systems [2], [3]. Second, multicore architecture complicates the evaluation of the Worst-Case Execution Time (WCET) of the tasks [4], [5]. By the very nature of multicores, multiple tasks can be executed simultaneously, potentially leading to inter core interferences that are difficult to take into account. In addition the architecture of modern processors – often superscalar with a memory hierarchy – further complicates the analysis.

Recently we have proposed a data flow monitoring service based on runtime verification [1]. The service target an AUTOSAR-like plateform, which is a static application. It could be possible to use the monitoring service in a dynamic system while the application architecture is static. The goal of this service is to detect unwanted communication patterns in complex in-vehicle embedded multitask real-time software. For this purpose monitors are automatically generated from formal models of the system and of the expected communication patterns. These monitors are statically plugged in the kernel of Trampoline, an open source RTOS based on AUTOSAR OS specification. Let us underline that the service we propose is fully independent of the multitask architecture. It can be used within a time-triggered or an event-triggered system as long as the tasks communicate through system calls (which is mandatory in a critical system where memory protection must be used).

For now, since trampoline is a monocore real time operating system, the implemented service is only suitable for this kind of system. In the near future, the service will be extended to multicore architecture.

In this paper, we present the implementation of the service and its performance. We propose and evaluate several memory optimization techniques aiming at minimizing the memory footprint of the monitors.

The paper is organized as follow: in section II we give an overview of the service with a simple use case; in section III we describe related works; in section IV we briefly describe the monitor synthesis techniques and tool; in section V we describe the implementation of the service; in section VI we give some results concerning the performances of our implementation; in section VII we conclude the paper.

II. Illustration with a simple use case

Consider the software system described by Figure 1. It is composed of three concurrent tasks T_0, T_1 and T_2 that

communicate through two shared blackboard buffers b_0 and b_1. Let us consider that T_2 reads data from both b_0 and b_1 to make a coherency check of the results produced by T_1. We want to verify at runtime that **when T_2 starts reading, both buffers are synchronized and stay synchronized until it has finished**. The buffers are synchronized if the data currently stored in b_1 has been produced with the data currently stored in b_0. To implement this verification, each send and receive operation in the system is intercepted and forwarded to a monitoring service. The service uses a monitor to compare the system operations with the specification and outputs a verdict.

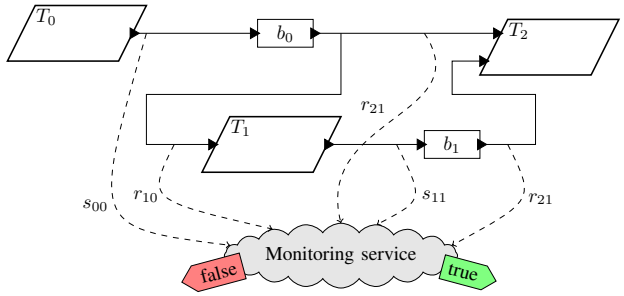


Fig. 1. Monitoring architecture. s_{xy} are sending events by T_x to b_y and r_{xy} are receiving events by T_x from b_y.

The service must detect and report all the errors in the communication patterns as soon as possible (as soon as T_2 performs a read operation while the buffers are not synchronized) and must not report false positive. The implementation of the service has also to meet non-functional characteristics to fulfill the requirements of industrial real-time embedded systems as listed for instance in [6]:

- it has to accomplish its function introducing a minimal overhead;
- after the monitor inclusion, it must be possible to check if the system is still able to respect its timing requirements;
- in industrial contexts, components have to be qualified (or certified) with respect to standards (AUTOSAR and ISO26262 in the automotive context).

To address the functional constraints and the possibility to qualify the implementation, we rely on formal methods to automatically generate the monitors and we adopt a fully static implementation approach conform to the AUTOSAR industrial standard. To address the other non-functional constraints, we include the monitor in the kernel of the RTOS to avoid useless switches between user and kernel modes produced by each system call. We also propose and characterize alternative implementations of the monitors leading to different memory footprint and time overheads. These informations can then be used as inputs for the schedulability analysis.

III. Related Works

Many theoretical works exists on the theoretical foundations of runtime verification and its application to distributed systems. Fewer works deal with actual implementation of runtime verification for (industrial) real-time embedded systems.

The MaC (Monitoring and Checking) system and its implementation for the Java platform are described in [7]. At design-time, monitoring scripts are written with an extension of LTL (linear temporal logic, see below), as well as the glue between these scripts and the monitored system. At runtime, instrumentation in the application code sends information to an automatically generated event recognizer. Then, the event recognizer forwards relevant events to a checker that executes the monitoring script. We follow a similar methodology with a much more domain-specific approach. This implies that we do not need to instrument the application code and we do not need an event recognizer. Moreover, offline processing of the specification allows us to obtain very simple and very efficient monitors in the form of Moore machines. We obtain a solution which is less intrusive and with very small overheads to fulfill the stringent requirements of our application domain.

Copilot [6] is a DSL (domain specific language) to program monitors for hard real-time systems. Copilot programs are compiled into C code and executed by recurrent tasks scheduled alongside application tasks. These programs sample the shared variables and check properties of the data flows. We also target properties of the data flows. At the moment we focus on temporal aspects while Copilot addresses both temporal and functional aspects. To ensure the synchronization between the application and the monitors, Copilot assumes that application tasks conform to a specific programming pattern. Our solution does not suffer from this restriction.

In [8], an high-level overview of an other architecture for fault-tolerance in AUTOSAR is presented. This architecture uses monitors automatically generated from specifications written in SALT, a LTL-like specification language. We reuse the monitor synthesis technique used by SALT. However, our solution differs greatly at the implementation level. Our monitors are plugged inside the RTOS kernel. This allows to minimize the time overhead and to bypass the scheduler in order to achieve the lowest detection latency.

IV. Monitor Synthesis

A. Runtime Verification

Runtime verification is a lightweight formal method that consists in deciding on-line if the current run of a monitored system is conform to some specification. The specification is generally expressed with a temporal logic. Monitors are automatically generated from the specifications. Monitors are simple machines that observe the system and tries to evaluate in a finite time if all the possible extensions of the current run are models of the specification or if none of the possible extension is a model of the specification. A monitor is inconclusive as long as some possible extensions are models of the specification whereas some other are not.

We use runtime verification tools to build the monitors that will be used by our service. The process that we use has three steps.

In a first step, we describe the system that we want to monitor in the form of finite state automata. Each state of each automata is associated with a property. Each automata reacts

to a set of events of the system. The model of the whole the system is obtained by computing a synchronized product of the automata. A state of this product is thus associated to a vector of atomic properties.

In a second step, we use the monitor synthesis technique proposed by Bauer *et al* and described in [9]. The specification is given as a LTL formula [10]. If the property is monitorable, a monitor is computed in the form of a Moore machine. The input alphabet of the Moore machine is the set 2^{AP} where AP is the set of atomic proposition associated with the states of the model of the system. The output alphabet is the set $\{\top, \bot, ?\}$. The monitor outputs $?$ as long as it can not conclude if the run is or not a model of the property. It outputs $\top$ as soon as it detects that all the possible continuation of the run are models of the property (therefore the run will be a model of the property). It outputs $\bot$ as soon as it detects that none of the possible continuation of the run is a model of the property (therefore the run will not be a model of the property). Notice that when the monitor outputs either $\top$ or $\bot$, the monitoring is finished. The synthesis algorithms and the corresponding correctness proofs are given in [9].

The third and last step consist in composing the model of the system and the monitor in order to obtain a monitor that reacts to events of the system.

B. *Tool support:* Enforcer

In order to compute the monitors, we have developed the prototype tool *Enforcer*[1]. Enforcer implements the three steps described above. It relies on the tool *LTL2BA*[2] [11] for a step of the monitor synthesis procedure. The figure 2 shows the inputs and outputs of *Enforcer*.

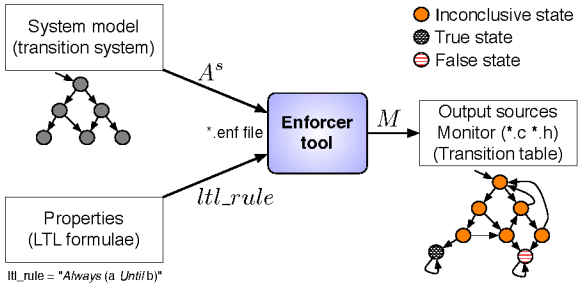


Fig. 2. Monitor synthesis with *Enforcer*

Let us illustrate the whole process on the requirement express in section II. The system is composed of three dependent tasks and two buffers as illustrated in Figure 1. The buffers are manipulated through five primitives: s_{00}, r_{10}, s_{11}, r_{20}, r_{21}. In this system, it is expected that once T_2 start reading, both buffers are synchronized and stay synchronized until it has finished.

[1]*Enforcer* is developed by the group *systèmes temps réel* at IRCCyN. It is distributed under GPL licence. It is available here: http://enforcer.rts-software.org

[2]*LTL2BA* is developed by Denis Oddoux at LIAFA and Paul Gastin at LSV. A modified version is included in the *Enforcer* package.

1) The enforcer entry file: The *Enforcer* file for this system is given on Figure 3. Two automaton are required to model the system: m_sync represent the synchronization between b_0 and b_1. At the beginning, we consider b_0 and b_1 are synchronized. Once T_0 sends a data in b_0, the buffers are desynchronized. The synchronization process required that T_1 reads b_0 before writing the data to b_1. m_t2, corresponds to the behavior of task T_2. T_2 reads data from b_0 and b_1 without any constraint on the order. However, when T_2 reads b_0 (resp. b_1), it is systematically followed by a read from b_1 (resp. b_0).

Thus, the specification will express the constraint that once T_2 read b_0 or b_1, m_sync stay in the $sync$ state until T_2 completes. This is done in the **property** section. The **reset** section is used to specify what the service should do once the monitoring is finished: either stop this monitor or reset this monitor.

```
rule syncho {
  automata {
    m_sync {
      states {sync, nosync, inprog};
      events {s00, s11, r10};
      transitions {
        in sync {
          when s00 then nosync;
          when s11 then sync;
          when r10 then sync;
        }
        in nosync {
          when s00 then nosync;
          when r10 then inprog;
          when s11 then nosync;
        }
        in inprog {
          when s00 then nosync;
          when s11 then sync;
        }
      }
    } // end m_sync
    m_t2 {
      states {begin, firstb0, firstb1};
      events {r20, r21};
      transitions {
        in begin {
          when r20 then firstb0;
          when r21 then firstb1;
        }
        in firstb0 {
          when r21 then begin;
        }
        in firstb1 {
          when r20 then begin;
        }
      }
    } // end m_t2
  }
  property = always ((m_t2.firstb0 or m_t2.firstb1)
      implies (m_sync.sync until m_t2.begin));
  reset = true;
}
```

Fig. 3. *Enforcer* model of a simple system

From this property, *Enforcer* computes the monitor presented in figure 4. This monitor has three states. Once T_2 start reading, the monitor goes to the $St2$ state (inconclusive

output). The monitor stay there until T_2 completes or until an error occurs. This is the case when the two buffers become desynchronized (false output).

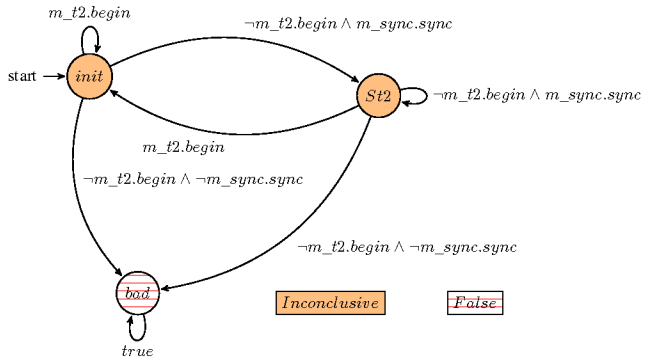


Fig. 4. Intermediate monitor

In a last step, *Enforcer* composes the monitor with the model of the system. The resulting monitor is given on figure 5. This monitor is a generated code that will be statically injected in the kernel of Trampoline as exposed in the next section.

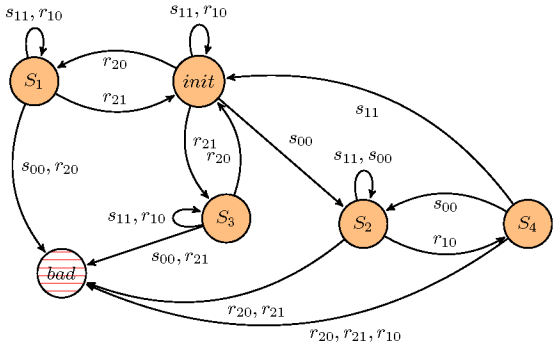


Fig. 5. Final monitor of the system

V. Implementation in the Trampoline RTOS

A. Runtime Support

1) Overview of the Trampoline RTOS: The service has been implemented in *Trampoline* [12], an AUTOSAR OS compliant open-source RTOS. AUTOSAR uses two means to communicate data between the tasks/ISRs: a) When the sender and the receiver do not share the same memory section, the IOC (Inter OS-Application Communication) is used; b) When the sender and the receiver share a memory section, the RTE (RunTime Environment) does a straight data copy between variables (without switching to kernel mode). In a fault-tolerant system, each task/ISR should have its own protected memory section and so we assume that the IOC is always be used.

The implementation of the IOC in *Trampoline* is not yet available but there is an implementation of the OSEK internal communication services (OSEK COM). Since the IOC is very close to OSEK COM, it can be used as a basis for our implementation.

With OSEK COM, tasks communicate thought message objects. There exist two kinds of message object: sending message object – smo – and receiving message object – rmo. A sending message object does not store any data and is connected to one or more receiving message object (1:N communication scheme). A receiving message object is a buffer. It can be either queued (mailbox), or unqueued (blackboard). Availability of a new message may be notified to the application by four means: task activation, event setting, callback and flag setting.

Conforming to AUTOSAR OS specification, Trampoline is a static RTOS. It is not possible to create or delete object at runtime. The configuration is given as a set of initialized objects (task, alarm, isr, resource, event, message, etc.) described offline in ARXML (AutosaR XML) or in a language called OIL [13]. The Trampoline toolchain includes an OIL compiler. This compiler processes an OIL configuration and generates the corresponding source code (files *.c and *.h).

2) Monitor inclusion in Trampoline and Runtime behavior: In order to integrate the service in the Trampoline toolchain, we have decided to slightly extend the syntax of OIL to support the definition of the objects required by *Enforcer* (events, rules, finite set automata). This allows to reference object of the system (task and message objects) in the data that will be sent to *Enforcer*. The global compilation scheme (including the monitor generation by *Enforcer*) contains 4 steps as illustrated on Figure 6:

- (1) – all objects are declared in OIL;
- (2) – the OIL configuration is processed. This leads to the generation of the configuration sources and the monitor description (*.enf file);
- (3) – *Enforcer* is called to generate the monitor files;
- (4) – the final step is the compilation (the makefile is automatically generated by the OIL compiler of Trampoline).

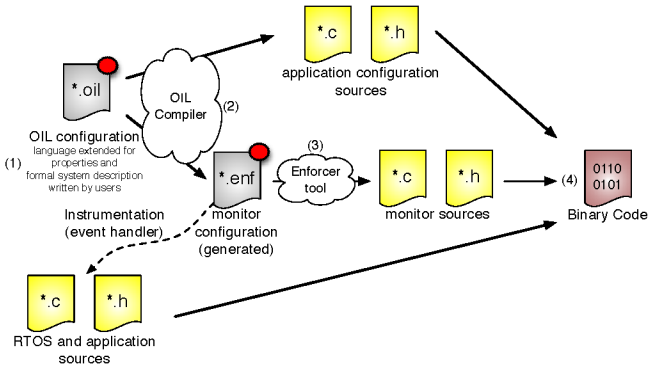


Fig. 6. Compilation chain

We will now illustrate the extension of the syntax of OIL with code from the example of the simple use case described in figure 1

- Monitored events are declared. An event is the combination of a task, a buffer and a service (send or receive a message).

```
enforcer_event s00 {
  // s00: T0 sends data to buf. b0
  task = T0;
  messageobject = b0;
  action = send;
};
```

- The system is described with finite state automata. The initial state is implicitly the first one;

```
automata m1 {
  state = sync; // initial state
  state = nosync;
  state = inprog
  transition sync_to_nosync {
    fromstate = sync;
    tostate = nosync;
    enforcer_event = s00;
  };
  /* etc */
};
```

- The rule declaration contains the property written in LTL and the reset flag. It also references the automata that must be used to generate the final monitor.

```
LTL_rule rule1 {
  automata = m_sync;
  automata = m_t2
  property = "always((m_t2.firstb0 or m_t2.firstb1)
      implies (m_sync.sync until m_t2.begin))";
  reset = true;
};
```

The consistency of the configuration is checked by the OIL compiler. If no error is detected, the *Enforcer* model (*.enf file) is generated. This model is processed by *Enforcer* to generate different source files (c files and headers). These files contain the *monitor configuration*, the *hook handler* and the *event handler* .

The monitor configuration includes the monitor transition table and the data structure used at runtime to run the monitor. The definition of this data structure type is given on figure 7[3]. This definition includes alternative parts corresponding to the different memory optimization policies provided by *Enforcer*. This data structure is stored in RAM whereas the transition table is stored in ROM. The constant fields could be grouped in a distinct data structure stored in ROM to save some bytes.

Among these fields, one can notice the two function pointers `false_function` and `true_function`. These functions are called when the monitor outputs a verdict. If the *reset* flag is set, the monitor is put in its initial state after the execution of one of these functions.

The *hook handler* contains the skeletons of the `false_function` and the `true_function`. The system integrator must fill these skeletons with its code. Some helper functions are provided to identify for instance the monitor and the last event. To stick with the AUTOSAR OS context, the system integrator must choose in the `false_function` between the actions usually proposed in case of a timing protection error: 1) shutdown the OS, this leads to restart the whole ECU; 2) terminate the application owning the task that has triggered the last event. 3) terminate the task that has triggered the last event.

[3]AUTOSAR normalizes storage classes using a set of macros. CONST(*type*,TYPEDEF) is used for a constant definition in a struct declaration.

```
struct TPL_FSM_INITIAL_INFORMATION {
  u8 monitor_id; /* monitor identifier */
#if ENFORCER_OPTIMISATION == NO
  u8 *automata; /* transition table */
#else
  unsigned long *automata;
#endif
  u8 current_state; /* current state */
  u8 nb_event; /* number of events */
  CONST(tpl_false_func, TYPEDEF) false_function;
    /* function called when "false" occurs */
  CONST(tpl_true_func, TYPEDEF) true_function;
    /* function called when "true" occurs */
  u8 false_state; /* false state identifier */
  u8 true_state; /* true state identifier */
  u8 reset; /* reset flag */
#if ENFORCER_OPTIMISATION_1 == YES
  u8 nb_bit_per_state;
  u8 mask; /* mask value */
  u8 nb_bit_per_line; /* transition table size */
#endif
#if ENFORCER_OPTIMISATION_2 == YES
  u8 nb_bit_per_state;
  u8 div_value; /* division value */
  u8 mod_value; /* modulo value */
  u8 mask; /* mask value */
  u8 garbage_size; /* bits lost per line */
  u8 nb_bit_per_line; /* transition table size */
#endif
#if ENFORCER_OPTIMISATION_3 == YES
  u8 nb_bit_per_state;
  u8 div_value; /* division value */
  u8 mod_value; /* modulo value */
  u8 mask; /* mask value */
  u8 nb_bit_per_line; /* transition table size */
#endif
};
```

Fig. 7. Monitor data structure description

Finally, the event handler is the entry point of the monitoring service. Since we focus on monitoring communication patterns, the message sending (SendMessage) and message receiving (ReceiveMessage) system calls of Trampoline have been modified to notify events to the monitors. The notification process uses a table indexed by the identity of the target message object and the identity of the task. The table contains pointers to generated functions. These functions calls the update function of the monitors interested in this event.

The monitor is included in the kernel of Trampoline for performance purposes. As an alternate implementation, notifying events to the monitor could be done in the RTE (so outside of the kernel). However beside the lack of memory protection for the data of the monitor, which is paradoxical for this kind of mechanism, this implementation would lead to an additional execution time overhead especially in a multicore

context. Outside the kernel, the monitor would need to make several system calls to: a) get the task or the ISR id, b) enter a critical section to prevent the monitor to be preempted and c) enter a critical section to rule the concurrency of monitor accesses by the other cores[4]. To sum up, 6 system calls would be needed. Because a system call incurs a $5.5\mu s$ penalty on the target we consider in section VI-A2, sending an event and updating the monitor would take more than $36.5\mu s$ instead of the $3.4\mu s$ for the implementation inside the kernel as shown in table II.

B. Avoiding the overhead for non-monitored component

Communication services on non-monitored components are also intercepted. In that case, the runtime behavior presented above is still valid but the function pointer leads to an empty function. Experiments show that this cost is about $1\mu s$ and represents almost 30% of the monitor execution time when one rule is referenced as presented in section VI-B.

In Trampoline, the communication services are called through two function pointers embedded in the receiving message objects: one for writing and one for reading. These functions depend on the kind of the message object (queued or unqueued). By providing a monitored and a non-monitored version for each of these functions and by setting each receiving message object to the version corresponding to its status (ie. monitored or not), the monitor overhead can be avoided for non-monitored message objects.

A slight modification of the Trampoline kernel is needed to minimize the overhead of the monitoring of the other services like task activation or termination. As in the communication, two versions of the services likely to be monitored are provided. Only one of them includes a call to the monitoring service. Instead of calling the service directly, it is called by using a function pointer embedded in the corresponding descriptor. For instance the function pointer of the task activation service would be embedded in the task descriptor. Monitored tasks would have a pointer to the monitored task activation service and the other tasks would have a pointer to the non-monitored task activation service. The memory footprint of this solution depends on the code size of the service and of the number of monitored services. Time overhead of this modification has been measured on the system presented in VI. It incurs a $0.28\mu s$ penalty over a service without the monitoring capability. This overhead has to be compared to the $1\mu s$ overhead with the empty function approach.

C. Memory Optimization

An automotive embedded systems has a limited amount of memory (RAM and ROM). It has also a limited computing capacity that must be shared among tasks under strict timeliness constraints.

As exposed above, the RAM footprint of the monitoring service is linear in the number of monitors: a descriptor (of constant size) is required for each monitor.

[4]In AUTOSAR, a critical section for the same core and between 2 or more cores is not protected by using the same mechanism.

The computation cost is also linear in the number of monitors that must be updated upon the occurrence of an event. Most of the time, an event will trigger the update of only one monitor (this statement is discussed in section VI-B2). In this case, the computation cost is constant, with an order of magnitude close to other system services (measurements to support this claim are given in section VI-B1).

It is much more difficult to evaluate the ROM footprint because the size of the transition tables depends on the rules. In the worst case, the determinization algorithm used in the monitor synthesis can produce an automaton with 2^n states where n is the number of state of the Büchi automaton computed from the LTL formula. In practice, the ROM greatly varies from one monitor to another but is always several order of magnitude greater than the RAM footprint. As a conclusion, the most critical issue caused by the introduction of the monitoring service is the ROM footprint. We will thus present three coding schemes for the transition tables that allow to mitigate the ROM footprint. These optimizations imply an increase of the RAM footprint and of the computation cost. The increases are slights and the RAM footprint as well as the computation cost are kept constants.

In a real-time context, matrix-based coding of transition tables are obviously more desirable than list-based coding because they allow to obtain the target of a transition in constant time. In the following the transitions tables are coded in s-by-e matrix M, where s is the number of states, e is the number of events and $m_{i,j}$ is the target of the transition from state i by event j. In the first implementation, each state $m_{i,j}$ is coded by a byte. With this implementation, a monitor can have up to 256 states. Figure 8 shows an example of this coding for a transition table with 5 states and 6 events. In this case, 180 bits are "lost" (5 bits each state). The optimization schemes are based on this observation.

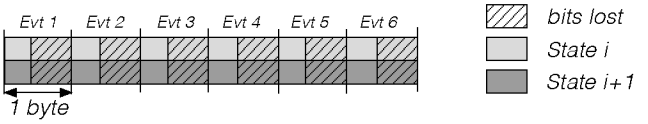
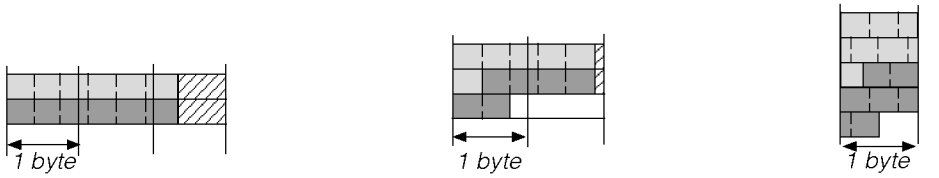


Fig. 8. Initial implementation

In the following, N_l is the number of bits lost, N_s is the number of states, N_e is the number of events, and N_o is the number of bytes needed to store the matrix. N_o is expressed as a multiple of n. The value of n is aligned on byte boundaries: 8, 16, 24 or 32 bits. It is chosen by *Enforcer* to minimize N_l. In order to simplify equations, we denote $N_b = N_e * N_s * \lceil log_2 N_s \rceil$ the total number of useful bits in matrix. The ROM footprint of a coding scheme is characterized by the couple (N_l, N_o). For the initial implementation, it is given by equation 1 and 2:

$$N_l = (8 - \lceil \log_2(N_s) \rceil) \times N_e \times N_s \quad (1) \qquad N_o = N_s \times N_e \quad (2)$$



(a) Method 1 (b) Method 2 (c) Method 3

Fig. 9. Transition table after optimization for each solution

1) Optimization – Method 1: The first scheme is illustrated by figure 9(a). In this scheme, values in the rows are concatenated without taking into account the boundaries between bytes. In order to limit the time to obtain the target state of a transition, a row must fit in a memory word: $\lceil \log_2(N_s) \rceil * N_e \leq 32$. This is a reasonable constraint verified by a large number of monitors. The ROM footprint of this scheme is given by equation 3 and 4:

$$N_l = (n - N_e \times \lceil \log_2(N_s) \rceil) \times N_s \quad (3) \qquad N_o = N_s \times \frac{n}{8} \quad (4)$$

2) Optimization – Method 2: The second scheme is illustrated by figure 9(b). Values in the rows are concatenated without taking into account the boundaries between bytes. Rows can span among memory words as long as a value is not split between two different words. With this technique, at most $N_{ln} = \lceil \log_2(N_s) \rceil - 1$ bits are lost per n bytes. There is no more constraints on the number of states and events of the monitor. The ROM footprint of this scheme is given by equation 5 and 6:

$$N_l = \begin{cases} \frac{N_b}{n - N_{ln}} \times N_{ln} & if\ (N_b\ mod\ (n - N_{ln})) = 0 \\ \left\lceil \frac{N_b}{n - N_{ln}} \right\rceil \times N_{ln} + & \\ (n - N_{ln}) - & otherwise \\ (N_b\ mod\ (n - N_{ln})) & \end{cases} \quad (5)$$

$$N_o = \left\lceil \frac{N_b}{n - N_{ll}} \right\rceil \times \frac{n}{8} \quad (6)$$

To obtain the target state of a transition, a division is required. This has an important impact on the computation cost. The magnitude of this impact is amplified on platforms where no hardware division is provided (this is the case of the platform used to measure the performance of our implementation in section VI).

3) Optimization – Method 3: The third scheme is illustrated by figure 9(c). Values in the rows are concatenated without taking into account the boundaries between bytes. Rows can span among memory words. A value can be split between two different words. No bits are lost except in the last byte. Notice that no division is required to compute the target state of a transition. The ROM footprint of this scheme is given by equation 7.

$$N_l = \begin{cases} 0 & if\ (N_b\ mod\ 8) = 0 \\ 8 - (N_b\ mod\ 8) & otherwise \end{cases} \quad (7)$$

$$N_o = \left\lceil \frac{N_b}{8} \right\rceil \quad (8)$$

4) Discussion: An attempt to represent and compare the three methods in terms of ROM footprint mitigation is given on figure 10. It represents the ratios of bits saved by each optimization method when compared to the initial implementation. This ratio depends on the number of states and the number of events. The graph on the left of figure 10 gives the ratios for 3 events. The graph on the right gives the ratios for 5 events. Each graph gives the ratios for a number of states varying between 4 and 256. Each integer abscissa x corresponds in fact to the set of points $[2^{x-1} + 1, 2^x]$. For the first method, there is no variability in such a set. For the second method, there is an important variability so two series of points are given corresponding to the better and the worst case. For the third method, the variability is negligible.

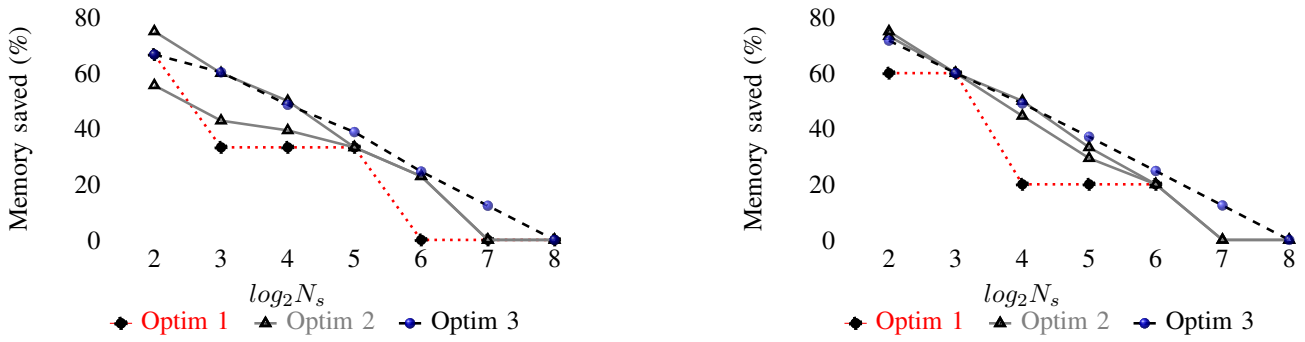
First, the efficiency of all the methods decreases when N_s increases. This is explained by the fact that when 7 or 8 bits are required to code the number of states, the number of bit lost by the initial implementation becomes very low (or even null).

Second, the third method appears to be the most efficient, althouh it can be overtaken in some cases by the second one. The third method looses bits only in the last byte. The second method looses bits on each slice of n bytes (where n is a value computed by Enforcer as exposed above). Thus the second method gets worst compared to the third one when the number of slices becomes important, ie. when the number of events becomes important. The first method is globally less efficient.

VI. Evaluation and Results

A. Testbed Presentation

1) System Description: Measurements are done on the system described in section II. The three tasks T_0, T_1, T_2 communicate using two shared buffers b_0 and b_1. T_0 writes to b_0; T_1 reads from b_0 and writes to b_1; and T_2 reads from the two buffers. We denote $s_{x,y}$ the event: task x writes to buffer y, and $r_{x,y}$ the event: task x reads from buffer y. We assume that both buffers are unqueued (reading does not consume the



(a) Percentage of memory saved ($N_e = 3$)

(b) Percentage of memory saved ($N_e = 5$)

Fig. 10. Memory optimization for ($N_e = 3$ and $N_e = 5$)

data). The corresponding Enforcer model is composed of three automata. m_sync (Fig 11(a)) and m_t2 (Fig. 11(b)) have been introduced above; 11(c) describes the behavior of the buffer b_0.

We consider three alternative properties for this system.

- ***Prop 1:*** when T_2 reads b_1 buffers are synchronized and stay synchronized until T_2 completes;

```
always ( m_t2.firstb1 implies (
            m_sync.sync until m_t2.begin
  )
)
```

- ***Prop 2:*** once T_2 has read b_0, it must wait that the buffers are synchronized before to read T_1.;

```
always ( m_t2.firstb0 implies (
    ( not (m_b0.writeb0) and not (m_t2.begin) )
      until ( m_sync.sync and m_t2.begin )
  )
)
```

- ***Prop 3:*** We always want the scheme: s_{00} then r_{20} then r_{21};

```
always ( m_b0.writeb0 implies ( next (
  m_b0.b0read2 and ( m_b0.b0read2
    implies ( next (
      m_b0.b0read1 and ( m_b0.b0read1 implies (
        next (m_b0.writeb0)
      ))
    )))
)))
```

Table I shows the main attributes of each property. The size column gives the size of the transition table in the initial implementation.

	Number of states	Number of events	Monitor size
Prop 1	8	5	40 bytes
Prop 2	20	5	100 bytes
Prop 3	7	3	21 bytes

TABLE I
PROPERTIES FEATURES

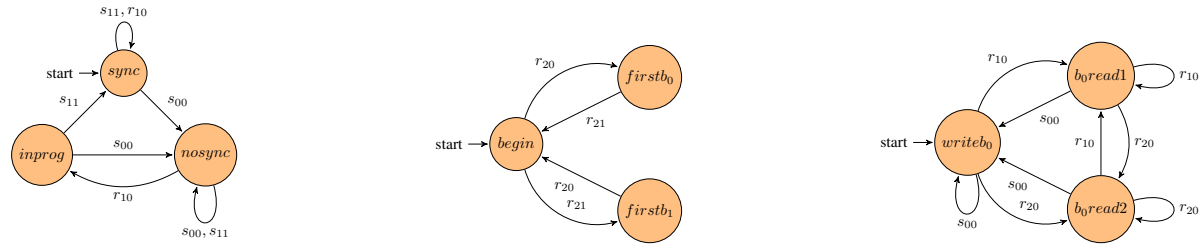
2) Hardware Platform: The implementation targets an Olimex lpc2294 board. The LPC2294 MCU is a 32-bits ARM7TDMI-S chip, running at 60 MHz. The compiler is *gcc* and optimization level is O3. All experiments have been run only once because there is no memory cache on this platform so the execution time is constant. Both program and data are stored in external RAM. Results have been obtained by measuring the number of cycles required to execute the services (i.e. SendMessage() and ReceiveMessage()) when the monitor framework is disabled and then when it is enabled. We took into account the kind of notification triggered by the ReceiveMessage service.

B. Results

1) Monitor Impact with one property: For these experiments, we consider the monitor generated from property *prop2*. The figure 12 shows the time needed to perform the main communication services (send and receive a message) compared to the one needed by the monitor, first when no memory optimization is performed and then for each of the optimization methods. It is worth noting that the monitor execution time does not depend on the communication service called. This is because the operations performed by the monitor are always the same: 1) intercept an event; 2) call the change state function; 3) output a verdict. At last, the time required to analyze the intercepted event is presented. This cost is the part of the monitoring imputed to each call of SendMessage or ReceiveMessage services, even if no monitor is attached to the event.

With no memory optimization, the monitor execution time represents about $33,5\%$ of the time required by the fastest service (i.e. *ReceiveMessage*) and 14% of the time required by the slowest one (*SendMessage* with *ActivateTask* notification). The overhead is reasonable compared to the benefit brought by the monitoring service.

The monitor introduction has also an impact on the memory. We separate the monitor impact on the memory in three categories: The code needed to get the new state after an event interception in ROM; the size of the transition table in ROM; the size of the data structure, in RAM. Table II shows

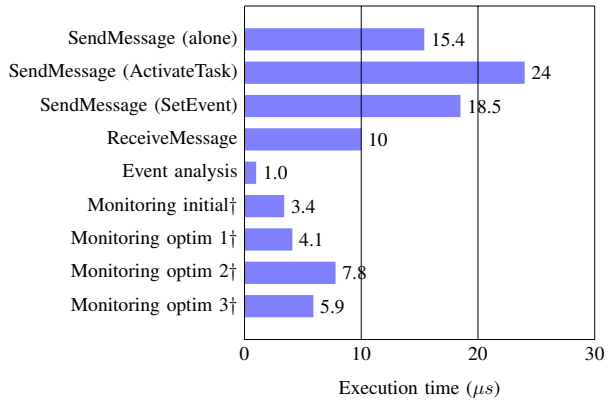


(a) *m_sync*: Buffer synchronization

(b) *m_t2*: T_2 behavior

(c) *m_b0*: b_0 behavior

Fig. 11. Formal model of the monitored system



†The Event analysis time is included in these measures

Fig. 12. Execution time for *prop 2*

	Execution Time	Memory print		
		Transition table ROM	Data structure RAM	Code size ROM
Initial	$3,4\mu s$	100 *bytes*	15 *bytes*	168 *bytes*
Optim 1	$4,1\mu s$ $(+20,5\%)$	80 *bytes* (-20%)	18 *bytes* $(+20\%)$	200 *bytes* $(+19\%)$
Optim 2	$7,8\mu s$ $(+129\%)$	68 *bytes* (-32%)	21 *bytes* $(+40\%)$	244 *bytes* $(+45\%)$
Optim 3	$5,9\mu s$ $(+73,5\%)$	63 *bytes* (-37%)	20 *bytes* $(+33.3\%)$	292 *bytes* $(+73,8\%)$

TABLE II
MEASUREMENTS : IMPACT OF THE OPTIMIZATION POLICY FOR *prop2*

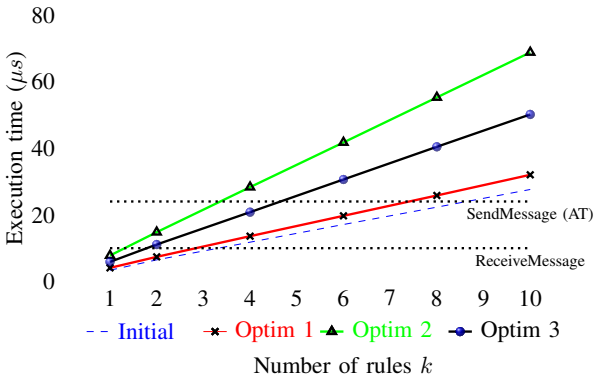


Fig. 13. Execution time for k rules ($k \in \{1, 2, 4, 6, 8, 10\}$)

the results of property *prop2*: computation cost and memory footprint.

As expected, we can observe that the most efficient optimization is the third one. On the other hand, the overhead on the execution time and the code size are increased by almost 74%. It is worth noting that the results are given for the worst case (i.e. when the next state has to be computed by the concatenation of a data from a line i and a data from the following line). However this case will be encountered only occasionally. The mean increase is 44%.

In this experiment, the first optimization solution appears to be the best tradeoff: 20% of the memory is saved while other parameters are gently increased (e.g. only 20% for the execution time). Finally, the second solution does not present any interest.

2) Monitor Impact with n properties: We consider now the monitors generated from properties *prop1*, *prop2*, and *prop3* previously defined. We replicate these monitors in order to obtain configurations where an event is shared by $k \in \{1, 2, 4, 6, 8, 10\}$ monitors.

In figure 13, we plot the execution time taken by the monitoring service to update k monitors. This overhead increases linearly with k. It is composed of a constant part (time required to obtain the set of monitors to update) followed by a linear part (time required to update each monitor). For example, if the second optimization is used and if two monitors evolve on a same event, the monitoring service takes more time to execute than the ReceiveMessage service (non monitored) alone.

As can be seen on the figure, if the number of monitors that must be updated after an event becomes greater than 6, the overhead of the monitoring service becomes prohibitive. Let us underline that most of the time, the monitors of a system watch different parts of the architecture and do not share events.

For the case where some monitors share events, in order to maintain a constant and predictable overhead, a solution consist in composing the monitors. This allows to update all the monitors in one operation (constant overhead). The

drawbacks of this solution are a potential increase of the ROM footprint and a decrease of the quantity of information produced by the service that can be used for diagnosing the system.

As an illustration, let us consider two monitors M^1, M^2, with event sets Σ^1 and Σ^2. We denote $|Q^1|$ and $|Q^2|$ the number of states of M^1 and M^2.

In the case where $\Sigma^1 \subseteq \Sigma^2$ or $\Sigma^2 \subseteq \Sigma^1$, the composition is the intersection of the two monitors. The set Q of states of the composed monitor is such that $|Q| \leq max(|Q^1|, |Q^2|)$. For example, let us consider the monitors generated from properties *prop1* and *prop2*. Monitor size for each property is given in Table I. The total memory taken by these two properties represent $(20 + 8) \times 5 = 140\ bytes$ with the initial implementation. After the composition, there are only 14 states so $14 * 5 = 70\ bytes$ are required.

In the general case (some events are shared but not all), the composition impact on the ROM depends on the ratio of shared events. The set of states of the composed monitor Q is such that $|Q| \leq |Q^1| \times |Q^2|$. The result is in fact very dependent on the application. A dedicated analysis for each situation is required.

The main drawback of the composition concerns the diagnostic. When a composed monitors outputs a negative verdict, it is not possible to detect which property has been violated. This has to be done by the user in the `false_function`. Moreover, a composed monitor outputs a positive verdict only when all the composed properties are enforced.

VII. Conclusion and Future Works

In this paper, we show that runtime verification can be introduced in industrial real-time embedded systems with a minimal execution time overhead and an acceptable memory footprint in monocore context. To do so, we propose to inject monitors in the RTOS kernel. We have developed a tool named *Enforcer* that generates the source code of the monitors that can be injected in the kernel of the Trampoline RTOS.

We have presented the implementation of the monitoring service and a performance evaluation of this implementation. We have proposed 3 optimization solutions to limit the size of the monitor transition table. In most case, these solutions are very efficient, to the price of an extra execution time overhead that can be significant. These solutions give the possibility to the system integrator to make a tradeoff between memory and execution time.

For now, the service targets monocore architectures. However it will be extended to multicore ones. Beside the redesign of the implementation, this extension will have to address the problem of truly simultaneous events. Whether the formal model of the system considers simultaneous occurrences of events or it serializes them remains an open problem that we will undertake in future works.

The service will also be used in different case studies so as to determine how it could be extended to better fulfill the needs of embedded real-time applications.

Acknowledgment

This work has been supported by Renault S.A.S., 1 Avenue du Golf, 78280 Guyancourt - France

References

[1] S. Cotard, S. Faucou, J.-L. Béchennec, A. Queudet, and Y. Trinquet, "A Data Flow Monitoring Service Based on Runtime Verification for AUTOSAR," in *International Conference on Embedded Software and Systems (ICESS)*, 2012.

[2] A. Fedorova, M. Seltzer, and M. D. Smith, "Cache-fair threads scheduling for multicoreprocessors," Harvard University, Tech. Rep. TR-17-06, 2006.

[3] J. H. Anderson and J. M. Calandrino, "Parallel real-time task scheduling on multicore platforms," in *Real-Time Systems Symposium (RTSS)*, 2006, pp. 89–100.

[4] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction caches," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008, pp. 80–89.

[5] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *Real-Time Systems Symposium (RTSS)*, 2009, pp. 68–77.

[6] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A hard real-time runtime monitor," in *International Conference on Runtime Verification (RV)*, 2010, pp. 345–359.

[7] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky, "Monitoring, checking, and steering of real-time systems," in *International Workshop on Runtime Verification (RV)*, 2002, pp. 95–111.

[8] A. Bauer, M. Leucker, and C. Schallhart, "Runtime reflection: Dynamic model-based analyis of component-based distributed embedded systems," in *Modellierung von Automotive Systems*, 2006.

[9] ——, "Runtime verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, 2010.

[10] A. Pnueli, "The temporal logic of programs," *Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 46–57, 1977.

[11] P. Gastin and D. Oddoux, "Fast LTL to Büchi automata translation," in *International Conference on Computer Aided Verification (CAV)*, 2001, pp. 53–65.

[12] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline - an open source implementation of the OSEK/VDX RTOS specification," in *International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2006, pp. 62–69.

[13] OSEK/VDX, "OSEK/VDX - OSEK Implementation Language," OSEK Group, Tech. Rep. v2.5, 2005.