OSPERT 2005

Workshop on Operating Systems Platforms for Embedded Real-Time applications

In conjunction with the

17th Euromicro Conference on Real-Time Systems (ECRTS 05)

Palma de Mallorca, Balearic Islands, Spain

July 5, 2005

Sponsored by:

The ARTIST 2 Network of Excellence on Embedded Systems Design

http://www.artist-embedded.org/FP6/

Editor: *Giuseppe Lipari*, Scuola Superiore Sant'Anna, Italy
Copyright ©2005 by the authors

OSPERT 2005

Workshop on Operating Systems Platforms for Embedded Real-Time applications

Program Chair:
Giuseppe Lipari

Program Committee:
Giuseppe Lipari
Gerhard Fohler
Steve Goddard
Hermann Härtig
Michael Gonzalez
Scott Brandt
Ismael Ripoll

Reviewers:
Mario Aldea Rivas
Scott Banachowski
Igor Barsanti
Marko Bertogna
Tim Bisson
Scott Brandt
Michele Cirinei
Gerhard Fohler
Steve Goddard
Michael Gonzalez
José Javier Gutiérrez
Hermann Härtig
Caixue Lin
Giuseppe Lipari
Antonio Mancina
Luca Marzario
Ismael Ripoll
Joel Wu

Preface

Until a few years ago, research on real-time operating systems (RTOS) was considered a closed and sterile field. Today, a number of commercial operating systems provide support for fixed priority scheduling and assessed methodologies used in industry are based on the well-known rate-monotonic scheduling analysis. This was considered enough for most real-time systems. However, since a few years, an increasing number of developers are looking againg at the research on RTOS with novel expectations.

On one side, the widespread success of real-time embedded system technology has raised a number of interesting problems that need to be addressed by RTOS providers. First, novel hardware architectures, consisting of homogeneous and heterogeneous multiprocessors, or reconfigurable FPGAs, are now very appealing for their flexibility and cost. Second, the complexity of current embedded applications is increasing exponentially, requiring quality of service support, dynamic reconfiguration and adaptation. Third, embedded systems have scarce computational resources and memory, and most of them are powered by batteries. Hence, the need for optimizing resource usage to reduce the cost and prolong the autonomy of the system.

On the other side, real-time requirements are now common also in high-end systems and quality of service is one of the most abused keywords. This means that best-effort approaches are no longer sufficient and that some sort of guarantee on the provided service is a strong requirement of any application. Hence, the need to provide (soft) real-time guarantees and resource reservation approaches also in OS for workstations, servers, desktop PC, etc. It is a common feeling among researchers and developers that current RTOS technology does not provide adequate solutions to the above problems.

This workshop is an attempt to bring together researchers, practitioners and developers of RTOSs to discuss the recent advances in RTOS technology and the challenges that lie ahead. The program committee selected 11 papers that were deemed interesting to the RTOS community. These papers are collected here, and will be presented at the workshop. Moreover, the program includes three invited talks: two are on succesful academic projects on RTOSs; and one invited talk by Thorbjorn Jemander of ENEA Epact, a provider of commercial RTOSs.

Finally, to foster the discussion among the participants, the program includes two panel sessions. Session 1, "Kernel Architectures for Embedded Systems" will discuss the problems of providing support for Embedded systems, especially for novel hardware architectures. Session 3, "Real-Time in general purpouse OS", presents the challenges in porting real-time technology on general purpouse systems. At the end of both sessions, participants will be asked to present their points of view on a set of interesting questions. The result of the discussion will be available on the OSPERT web site shortly after the conclusion of the workshop.

I would like to thank the organizers of ECRTS for giving us the opportunity to organize this workshop; the program committee for their help in the organization and the useful discussions; all the reviewers for their effort in making a very nice program; the ARTIST 2 Network of Excellence on Embedded Systems, for supporting this workshop; and all authors that have submitted, as the success of a workshop is mostly due to the authors!

Have a good time in Palma de Maiorca,

Giuseppe Lipari

PC chair

Program

Session 1: Kernel Architectures for Embedded Systems.

Challenges for scheduling media applications on a multiprocessor SoC 07
C. M. Otero Pérez, G. van Doren. Philips Research, Eindhoven.

Impact of Embedded Systems Evolution on RTOS Use and Design. 13
D. Andrews, I. Bate, T. Nolte, C. M. Otero Perez, S. M. Petters. University of York, Philips Research Lab, Eindhoven, The Netherlands, NICTA, Australia.

Operating Systems and Supporting Architectures for Embedded Real-time Systems. 21
N. Audsley, R. Gao, A. Patil, P. Usher, J. Withman. University of York.

Session 2: RTOS Architectures and APIs – I.

The FIRST API 27
Michael Gonzalez Harbour, Universidad de Cantabria, Spain

The need for configurable and flexible scheduling in a RTOS aspiring
to solve contemporary problems. 29
Thorbjorn Jemander. ENEA Epact, Sweden.

An overview of the XtratuM nanokernel. 31
M. Masmano, I. Ripoll, and A. Crespo. Universitad Politècnica de Valencia. Spain.

Kernel Support for Energy Management in Wireless Mobile Ad-Hoc Networks. 37
M. Marinoni, G. Buttazzo, T. Fachinetti, G. Franchino. University of Pavia. Italy.

Session 3: Real-Time in general purpouse OS.

Variable-Rate QoS in the OS Network Subsystem. 47
H. Cheng, X. Liu, and S. Goddard. University of Nebraska-Lincoln, USA.

Developing a Complete Integrated Real-Time System. 57
S. A. Brandt, S. Banachowski, C. Lin, and J. Wu., Univ. California, USA.

A Unified Framework for multiple type resource scheduling with QoS guarantees. 67
L. Palopoli, P. Valente, T. Cucinotta, L. Marzario, A. Mancina. Scuola Superiore Sant'Anna. Italy.

Session 4: RTOS Architectures and APIs – II.

Adding new features to the Open Ravenscar Kernel. 77
S. Urueña, J. A. Pulido, J. A. de la Puente, J. Zamorano. Universidad Politécnica de Madrid, Spain.

The OCERA operating system. 85
Alfons Crespo, Universidad Politécnica de Valencia, Spain

Lightweight RTAI for DSPs. 87
J. Kretzschmar, R. Baumgartl. Technical University chemnitz. Germany.

Power Measurement as the Basis for Power Management 95
D. C. Snowdon and S. M. Petters. NICTA, Australia.

Challenges for scheduling media applications on a multiprocessor SoC

Clara M. Otero Pérez, Giel van Doren
Philips Research Laboratories Eindhoven (PRLE)
{ clara.otero.perez, giel.van.doren}@philips.com

Abstract

Systems on chip (SoC) emerge as the chosen solution to accommodate the flexibility and performance demands of current media applications such as audio and video processing. A multiprocessor SoC combines embedded processors, specialized hardware blocks, and on-chip memories to meet the requirements of current and future products.

Providing scheduling analysis and techniques for such a SoC is a challenge, even more when cost is one of the driving factors for the design leading to resource constrained devices. To meet the performance demands of media applications, SoCs have to be used efficiently.

In this paper we show that processor sharing, keeping data on chip and the use of prefetching are solutions that improve efficiency but raise challenges for future research on real-time scheduling.

1. Introduction

Consumer multimedia devices are becoming increasingly flexible, accommodating late changes in standards or product scope during system design, and allowing in-the-field upgrades with new or enhanced features. To fulfill the flexibility and high processing requirements, consumer electronics vendors increasingly deploy heterogeneous multiprocessors *Systems on Chip (SoC)*. An additional drive for flexibility is the increasing cost of masks and design for the chips. Designing flexible SoCs that support a wide range of products can increase volumes, and as a consequence reduce the mask and design cost per chip.

Within Philips a new multiprocessor SoC is being developed based on the CAKE architecture [1]. This SoC, named Wasabi has a number of programmable DSPs (TriMedia[1]), general purpose CPUs, and hardware IP blocks such as a Memory Based Scaler (MBS), as depicted in Figure 1. The core of this architecture is a shared level two (L2) cache [2], which plays a central role in the communication both across processors as well as between processors and off-chip memory via a high bandwidth double data rate bus. The main role of the L2 cache is to reduce latency (increase processor efficiency) and reduce off-chip bandwidth.

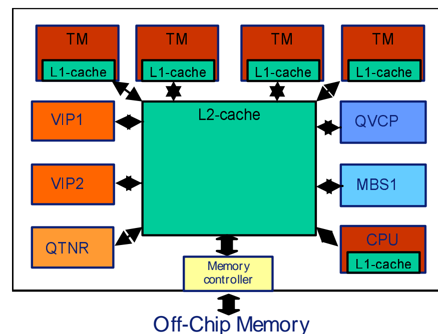


Figure 1 The Wasabi SoC

In this paper we focus on media applications that execute on the SoC, in particular video processing. A video processing application consists of a sequence of processing steps, each of which execute a function on a video frame. Each processing step can be executed in an IP block or as a software task running on a programmable core. High amounts of data (video frames) stream from a processing step to the next.

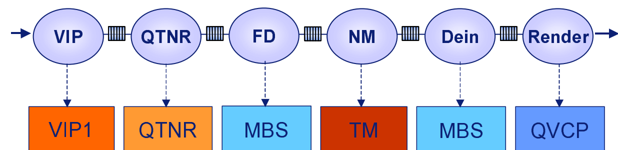


Figure 2 Streaming video application

Figure 2 depicts a processing chain of picture improvement algorithms for an analogue input stream. The buffers between processing steps contain the intermediate data. An analog video frame is collected by the video input (VIP) IP block; temporal noise reduction is performed by the Quality Temporal Noise Reduction (QTNR) IP block. The film detection (FD)

[1] TriMedia is a family of Philips DSP cores optimized for A/V processing

is done on the Memory Based Scaler (MBS). The TriMedia executes the Natural Motion software algorithm whereas deinterlacing (DEIN) is done by the MBS. Finally, the Quality Video Coprocessor (QVCP) renders the frame.

The bandwidth between two communicating video processing steps is large. For example, for high definition video (1920x1080), interlaced (30 frames per second), format YUV using 2 bytes per pixel, the generated bandwidth is:

30 * 1920 * 1080 *2 = +/- 120 Mbyte/s

The data (buffers) can reside either on-chip, in the L2 cache, or off-chip. When intermediate buffers are off chip every frame has to be transferred twice over the bus, once for writing and once for reading, resulting in 2*120MB/sec for the above example. This bandwidth requirement scales linearly with the number of processing steps. In case of having multiple chains with multiple processing steps this rapidly reaches the available bandwidth limit.

However, significantly increasing off-chip memory bandwidth is undesired for high volume consumer devices, which are cost-driven. High bandwidth to off-chip memory is:

- expensive to implement. It requires more pins and more chips to widen the data path.
- expensive in power consumption.

As a result, the bus to off-chip memory remains a scarce resource, a potential bottle neck, and has to be taken into account explicitly when realizing applications on such a SoC. However, keeping all data on chip is also not an option due to on-chip memory cost. As a consequence, the on-chip memory is also a scarce resource in such a SoC and has to be used efficiently.

The following section describes how efficient resource utilization drives the realization of distributed applications in a SoC. Section 3 discusses the derived scheduling challenges. Finally, directions for solutions are presented in Section 4.

2. Efficient resource usage on a SoC

The challenge in realizing a video application on a resource constrained multiprocessor architecture is to use the resources cost-efficiently.

For cost reasons, expensive resources such as processor, memory and bus bandwidth often have to be shared. Reducing the number of processors on a chip, decreases the silicon area and as a consequence the cost of producing the chip. As an example, it is more cost-efficient in silicon area to have one faster IP block, such as the MBS, than to have two slower ones. Therefore, *processors are shared* among processing steps on the same chain as well as among different chains.

In many cases, the sharing of resources introduces interference reducing efficiency. For example cache trashing and bus congestion. The work in [3] studies the efficient use of the shared cache and the bounding of this interference.

In the case of processing resources, efficiency depends on the *Stalls* that occur, relative to the capacity (*C*) of the processor, as shown in Equation 1. Stalls cycles are wasted processor cycles, since the processor is waiting for instructions or data before it can proceed.

$$Eff = \frac{C - Stalls}{C} \quad (1)$$

As a direct conclusion, the efficiency increases by reducing the number of stalls. In a processor with a cache the stalls depend on the number of cache misses and the latency to get the data in case of a miss, as shown in Equation 2.

$$Stalls = misses \times latency \quad (2)$$

The latency of a miss depends on where in the architecture the miss occurs. In the Wasabi chip, a miss can occur in the L1-cache and in the L2-cache. When data is not in the L1-cache (L1 miss), but it is available in the L2-cache the latency is small, e.g. 8 cycles. This latency is relatively constant since the on-chip interconnect is over dimensioned. A miss in the L2-cache (L2 miss) results in a latency of at least 30 cycles. The latency varies depending on the bus load (increasing as the busload increases), as depicted in Figure 3. The latency seen by a processor is influenced by other processors behavior, due to bus sharing.

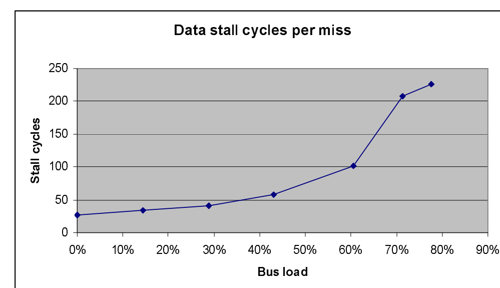


Figure 3 Latency increase with bus load.

Having the right data available at the right time can decrease the number of misses. A standard approach to achieve that is to increase the cache-sizes. However, it does not help much for video data. One HD video frames is already too large to fit in the L2 cache completely, and a SoC is typically processing a number of frames simultaneously.

A solution to reduce the number of stalls is to keep the huge amount of video data that stream between processing steps on-chip. It both reduces the load on the bus (and thus the latency for others), and reduces the L2-misses. These buffers have to be small due to the limited amount of shared on-chip memory (L2 cache). However, *keeping the data on-chip* that is communicated between a producing and consuming processing step, tightly couples the execution of these steps due to the small intermediate buffers.

A second solution to decrease the number of stalls is to *prefetch* data. Prefetching reduces the number of misses, by ensuring that the right data is available when it is needed. One way to realize prefetching is by predicting which data will be required next, based on the current memory accesses. Special hardware can perform this task [4]. Another way is that the processing step explicitly indicates what data is currently needed, and what data will be needed next.

Prefetching can be done at the level of the L1-cache or L2-cache. Since the size of the L1-cache is much smaller, less data can be prefetched in advance. Prefetching decreases the number of misses, but it does not decrease the load on the bus.

A third solution to decrease the number of stall cycles is to exploit on-chip memory by the algorithms executed by the processing steps. However, this optimization is algorithm dependent and falls out of the scope of this paper.

3. Scheduling challenges

As presented in previous section, efficiency and cost issues require solutions such as processor sharing, keeping data on chip and prefetching. In this section we explore which are the requirements that these solutions impose on the scheduling.

3.1. Processor sharing

Multiple research studies investigate the scheduling of applications on homogeneous and heterogeneous multiprocessor systems as in [5], [6], [7], [8]. These studies assume traditional task sets and well-behaved data flow graphs that do not correspond with practical solutions. For example, tasks are assumed to be ready to run (not generate idle), buffers do not cause tasks to block or latency to off-chip memory is independent on the total bus load. A heterogeneous multiprocessor SoC contains a wide variety of processing element that behave very differently. Many IP blocks are not shareable, as for example the video input unit (VIP) which can only process one stream at the time, receives data continuously at a given rate, and is not allowed to block. Among the shareable processing units we can distinguish two orthogonal properties to describe their behavior:

- Pre-emptive vs. non pre-emptive.
- Fine vs. coarse granularity.

For shareable processors the granularity of the processing plays an important role. The coarser the granularity, the less scheduling flexibility. For example a non-preemptive Noise Reduction IP block, has the granularity of a complete frame. The reason is that in the beginning of a frame the internal state is reset, and as a consequence it does not have be able to save/restore its internal state. Making such an IP block pre-emptive requires additional hardware to save/restore the state of the IP block and the generated overhead for the actual state saving/restoring.

For pre-emptive processors, there is a trade-off between the low overhead generated by coarse pre-emption granularities and the flexibility provided by the fine grain.

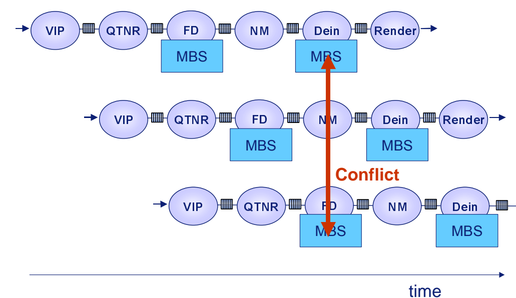


Figure 4 Conflict when sharing processors on pipelined execution

Furthermore, IP blocks can be fully dedicated to a single function, like MPEG-2 decoding, or multi function, as for example the MBS, which can execute film detection, scaling or de-interlacing functions. Depending on the processing power of the IP block, multiple processing steps of a single chain or parallel chains can be mapped on a single IP block.

The execution time varies depending on the function. If two processing steps request the same IP block to perform a tasks it might make a difference which one is executed first for the schedulability of the system. For example, when a long processing step is ready to run, it might be better to wait a bit for the short one to be released and executed than to start the execution of the long one causing the short one to miss its deadline. Figure 4 depicts such a situation for a single chain of which the execution is pipelined. The MBS is needed by two different processing steps from the same chain. This has to be solved by:

- speeding up the MBS clock sufficiently and schedule the two processing steps sequentially, or
- duplicating the MBS IP blocks, or
- make the MBS IP block pre-emptable.

The first solution is preferred for cost reasons. Static scheduling and prediction techniques are needed in this case to decide on the execution order. However, there is a trade-off between high processor speeds and latency on the bus. Temporal high busloads generated by high processor speeds can increase the latency of all bus traffic.

3.2. Keeping data on-chip

Figure 5 presents the producer/consumer case of tasks working on video data. In the ideal case the L2 cache contains the whole frame and the processors are used efficiently. However this solution is prohibitive as the L2 cache is a scarce resource. For example for HD video a frame is almost 4MB, which is larger than the whole cache. Instead, small buffers are used to keep data on-chip. This is possible because most data is produced/consumed in order. Small buffers result a tight-coupling between processing steps meaning:

- Consumer must check for the availability of data to read or wait otherwise
- Producer must check for the availability of space to write the data or wait otherwise.

Depending on the processor type on which the processing step is executing, waiting results on the processor idling or selecting another task to execute.

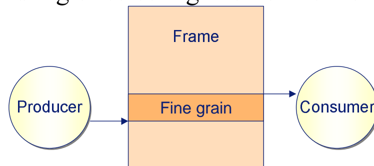


Figure 5 Tight-coupling synchronization to keep small buffers on-chip

In case the producer and consumer are scheduled on the same processor, a large number of context switches occur due to the tight coupling between them.

In case the producer and consumer are scheduled concurrently on different processors, the relative speeds have a considerable impact on the scheduling. If the speed of the tasks are comparable the buffers can be small and both tasks can execute smoothly (without interruptions). If the speeds are different the slowest task dictates the overall speed and the fast task has to wait for the next data.

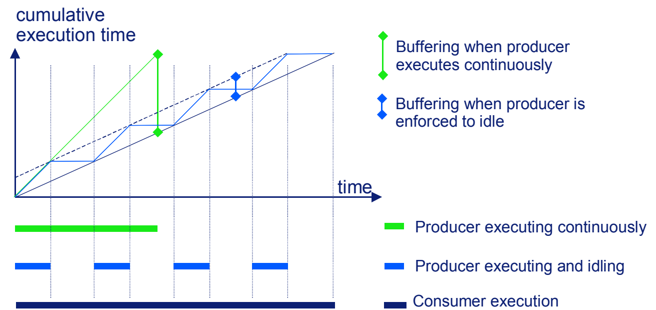


Figure 6 Effect of tight–coupling synchronization on scheduling.

Figure 6 depicts the situation when the producer is faster than the consumer. When the producer executes continuously the required buffering is greater than when it is enforced to wait for space to be available. This type of execution introduces inefficiency on the producer processor. In the case the producer is a non pre-emptable processor, it idles and this time is wasted. This property can be used by for example a power aware scheduler to regulate the speed of the producer to save power. If the producer is pre-emptable, the amount of context switches introduces inefficiency by:

- Reducing the overall utilization. The operating system has to execute the context switch code, save registers etc…
- Evicting the task data from the cache, so that when the task is scheduled to run again the data has to be loaded.

For example for HD video, which requires 120MB/sec assuming a realistic data size to keep on-chip of 64KB and double buffering, the context switching frequency generated would be up to 3840 Hz (120 Mbyte/s / 32 Kbyte).

In this case the holes generated by the tight coupling are very small and very frequent. Initial work on taking the idling into account on the schedulability test has been done in [9] for fixed priority scheduling. The main challenge to cope with this behavior is in how to use the holes efficiently. On the one hand, if the holes are not filled in, cycles are wasted and efficiency is low. On the other hand, if the holes are filled by another task the cache might be trashed and the penalty when the task is started again is high. There is a trade-off between reducing stalls by keeping data on chip and the overhead generated by context switching.

3.3. Prefetching

As mentioned in previous section, prefetching can be used to have the right data available at the right

moment. During integration it should be decided what to prefetch and what to keep on-chip. This functionality must not be hardcoded in the algorithm in a way that makes the algorithm hardware specific. The algorithm must indicate when the data is required, and a separate part of the system decides how to react on this info, e.g. by prefetching. However, two considerations should be taken into account:

- For effective prefetching, the data prefetched in the cache should be used before it is replaced. This can be done by ensuring that it is used shortly after it has been prefetched, or by preventing it from being replaced.
- Prefetching can result in peak bandwidths on the bus. The prefetch requests should be separated from other requests on the bus to prevent that the other requests get higher latencies due to the temporarily increased busload. Such a separation might even result in lower latencies for the other requests.

In the first case, the challenge for the scheduling would be to take into account prefetching before pre-empting a task that just got all its data. The second case does not impose requirements on the scheduling of the processing steps, but on the scheduling of the requests on the bus. It is outside the scope of this paper. A reservation mechanism such as the one sketched in [10] would be useful.

4. Solution direction

There are a number of scheduling techniques that aim at resolving the challenges identified in the previous section. Different scheduling techniques might be applied in different parts of the SoC. It is required that these techniques can cooperate, and not work against each other. Preferred scheduling techniques are techniques that provide isolation to achieve more predictable and composable systems.

Exploit processor sharing among processing steps. There are several scheduling algorithms known in literature that study the sharing of (non) pre-emptable processors using static and dynamic scheduling techniques. However, to address the challenges presented in the previous section these techniques have to be augmented by:

- Using the possibility of manipulating the processor speed.
- Considering the effects of the bus load on the latency.

Enable tight coupling. To be able to decide on the synchronization granularity and the buffer size for keeping the data on-chip, techniques have to be provided to analyze the following tradeoffs:

- Cache usage versus context switching frequency
- Processor speed versus the amount of context switching

Furthermore, dynamic scheduling techniques must support co-scheduling on multiple processors and effective use of holes left by the tight coupling

Achieve effective prefetching. Prefetched data in a cache can be lost due to an OS decision to switch to another task. An interesting research area is to study whether the OS can reduce the context switch overhead by using preferred preemption points or actively be involved in prefetching. For example, the OS has the knowledge when that task will be scheduled again, and can start prefetching data for the next task just before the next task is actually scheduled.

5. Acknowledgment

The authors would like to thank Liesbeth Steffens, Jos van Eijndhoven and Chun Wong for their review of this work.

References

[1] P. Stravers and J. Hoogerbrugge, "Homogeneous multiprocessing and the future of silicon design paradigms", *in Proceedings of the International Symposium on VLSI Technology, Systems, and Applications(VLSI-TSA)*, Apr.2001

[2] J. van Eijndhoven, J. Hoogerbrugge, J. Nageswaran, P. Stravers, and A. Terechko, "Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications," in *Dynamic and robust streaming between connected consumer electronic devices,* P. van der Stok, Ed., 2005.

[3] A. Molnos, M.J.M. Heijligers, S.D. Cotofana, and J. van Eijndhoven, "Compositional memory systems for multimedia communicating tasks", in *Proceedings of Design Automation and Test in Europe (DATE)*, Mar. 2005, Munich, Germany.

[4] T.R. Halfhill, "Philips Powers Up for Video", *in Microprocessor Report*, pp. 1-6, Nov.2003

[5] S. Baruah, S. Funk, and J. Goossens, "Robustness results concerning EDF scheduling upon uniform multiprocessors", *in Computers, IEEE Transactions on*, vol. 52, no. 9, pp. 1185-1195, Sept.2003

[6] S.K. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors", *in Computers, IEEE Transactions on*, vol. 53, no. 6, pp. 781-784, June2004

[7] M. Maheswaran and H.J. Siegel, "A dynamic matching and scheduling algorithm for heterogeneous computing systems", in *Proceedings of Seventh Heterogeneous Computing Workshop (HCW 98)*, pp. 57-69, May 1998.

[8] M. Jersak and R. Ernst, "Enabling scheduling analysis of heterogeneous systems with multi-rate data dependencies and rate intervals", in *Proceedings of the 40th conference on Design automation*, pp. 454-459, ACM Press, 2003.

[9] N.C. Audsley and K. Bletsas, "Fixed priority timing analysis of real-time systems with limited parallelism", in *Proceedings of 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, pp. 231-238, July 2004.

[10] C.M. Otero Perez, M.J. Rutten, E.F.M. Steffens, J. van Eijndhoven, and P. Stravers, "Resource reservations in shared-memory multiprocessor SoCs," in *Dynamic and Robust Streaming in and between Consumer-Electronic Devices,* P. van der Stok, Ed. Kluwer Academics, 2005.

Impact of Embedded Systems Evolution on RTOS Use and Design

David Andrews[*], Iain Bate[†], Thomas Nolte[‡], Clara M. Otero Pérez[§], Stefan M. Petters[¶]

Abstract

In this paper, we discuss how the evolution of embedded systems has impacted on the design and usage of Real-Time Operating Systems (RTOS). Specifically, we consider issues that result from the integration of complex requirements for embedded systems. Integration has been identified as a complex issue in various fields such as automotive, critical systems (aerospace, nuclear etc) and consumer electronics. In addition, the pressure on time-to-market, the emergence of multi-site development, and the ever-increasing size of software stacks are driving radical changes in the development approaches of modern applications. These complex requirements have placed greater requirements on Operating Systems with respect to how interfaces are defined and how resources are managed. These requirements are expanded and justified through the course of this paper. The requirements are then discussed in the context of emerging solutions from a number of domains.

1 Introduction

The demand for increased levels of functionality and dependability within small, lightweight embedded and real time systems has been steadily increasing for a number of years. While practitioners (academic and industrial) have been attempting to manage and deal with the complexity of modern embedded systems, the open issues are slowly becoming apparent to the average user. There have been numerous examples of projects not being fielded (e.g. Nimrod Early Warning aircraft) or fielded systems with problems arising associated with the interdependencies of complex requirements (e.g. Ariane 501, Mars Pathfinder). In the consumer electronics domain, integration problems have led to longer time-to-market and unresolved issues becoming visible for the end consumer (TV resetting, DVD recorders hanging). A common characteristic of all these examples are that they are the result of emergent properties resulting from integration and that are difficult to identify and verify under sterile laboratory conditions. An example of an emergent property related to real-time is deadlock and priority inversion when blocking on shared resources. This puts stringent requirements on the RTOS to provide better mechanisms for supporting integration in complex architectures and infrastructures using well defined abstractions and interfaces.

In this paper discuss open challenges for run time kernels (Section 2) and implications and limitations on Operating Systems (Section 3) and then discuss proposed solutions in the context of three applications domains: avionics, automotive, and consumer electronics (Section 4). Finally, the paper discusses some potential ways forward (Section 5).

2 Embedded systems development

Although embedded real time (RT) systems platforms and software are tailored for specific application domains such as consumer electronics, automotive and avionics they all share common problems. Problem examples include timing overruns due to effects such as blocking, unexpected time dependent calculations, and difficulties in understanding the implications of changes. These and other issues can be traced to conflicts in functional decomposition of high level requirements into the existing capabilities of desktop operating system semantics adopted for the embedded systems domain.

The existing open problems are a concern as greater and greater demands are being placed on precision and reliability in the growing breadth of application domains within systems that are becoming larger and more complex. Some of the challenging new trends in designing embedded systems are:

[*] David Andrews is at the Information and Telecommunication Technology Center, University of Kansas, Lawrence, KS, USA. email: dandrews@ittc.ku.edu

[†] Iain Bate is in the Real-time Systems Group, Department of Computer Science, University of York, York, YO10 5DD, UK. email: iain.bate@cs.york.ac.uk

[‡] Thomas Nolte is at MRTC, Department of Computer Science and Electronics, Mälardalen University, Västerås, Sweden. email: thomas.nolte@mdh.se

[§] Clara M. Otero Pérez is at the Philips Research Laboratories Eindhoven (PRLE), The Netherlands. email: clara.otero.perez@philips.com

[¶] Stefan M. Petters is at the National ICT Australia Ltd., Sydney, Australia. email: smp@cse.unsw.edu.au

- *Complexity* - Greater levels of functionality together with legacy code and lack of abstractions. The complexity of these issues is derived from:
  - Consumer electronics systems with the convergence of storage requirements, connectivity, and increased integration of functionality (camera, mp3, connectivity for consumer electronics).
  - Automotive systems that are integrating more functionality to decrease cabling and numbers of processors.
  - Avionics sector weight is a major issue. Size and weight issues are driving the movement away from federated systems to integrating functionality on fewer units.
- *Flexibility* - late changes, software download, reuse.
- *Dependability* - the level of integrity required in both failure and non-failure cases have increased. This has been brought about not just due to the fear of losing valuable sales (e.g., Intel adopted more formal approaches after their floating point unit problems on the early version of the Pentium processor) but also because of legislative pressure.
- *Connectivity* - on the systems level we have system integration where there is greater pressure on systems to work together, e.g., mobile phones to communicate with laptop computers etc..
- *Modularity* - needed to help provide maintainability (see below) but also to support concurrent and multi-site development of systems and subsystems. Concurrent and multi-site development is exacerbated as more projects are managed as partnerships and/or using global software development teams.
- *Maintainability* - there is a move away from monolithic development as it makes change difficult and does not support reuse strategies such as Product Line Architectures.
- *Upgradeability* - there is a need to be able to upgrade systems in the field. The upgrades need to be performed by both experts and naive users.
- *Size and power* - there is pressure towards smaller devices that can run over batteries for longer periods of times.

Early embedded systems were mostly uni-application, uni-processor systems point designs developed by teams co-located and targeted for systems with available power. In contrast, embedded systems are being developed to support more than one application domain and must support the upgrages and the addition of new applications in the field. This increases the need for standards and componentization within the solution requiring more abstract interfaces. Initially the need for greater flexibility implied additional functionality within software, e.g., engine control systems were converted from hydro-mechanical systems to computer-based systems. Now, with reconfigurable logic components, additional functionality is being specified at the hardware level.

At the same time there has been a great deal of technology improvements such as the availability of practical Real-Time Operating Systems (RTOS), 'novel' devices such as Field Programmable Gate Arrays (FPGA) or hyper-threading processors, Systems on Chip (SoC), Network on Chip (NoC), middleware etc.. These trends lead to novel approaches for both hardware and software. These type of solutions support a number of processors, which are often not uniform (e.g., general purpose and signal processing processors). These multiprocessor SoCs are deployed to cope with the market demand for high performance, flexibility, and low cost. NoCs are similarly used. A comparably new trend is the use of asynchronous logic in FPGAs. This is mainly driven to speed up the operation. However, this requires very detailed models on timing behaviour.

To achieve a cost effective solution, expensive resources, such as memory and processor time, are shared among concurrent applications. In the consumer electronics domain, given the dynamic load fluctuations of these applications, worst case resource allocation becomes prohibitive. The allocation of resources below average needs implies that applications have to get by with occasional overloads, reducing system reliability. In the automotive domain, the number of Electronic Control Units (ECUs) is high, driving costs, power usage and integration complexity up, proposing a new era of sharing of ECUs between several subsystems. More powerful, but fewer ECUs allow for automotive subsystems to share an architecture of ECUs. Due to the safety-critical nature of many automotive and avionic applications resource allocation are still based on worst case scenarios. However, integration problems emerge that needs to be treated, i.e., subsystem integration issues.

Component based technology is considered a prime approach to address the problem of time to market and the perceived advantage of reusing code and hardware regarding cost and reliability. The call for increased functional integration on fewer units leads to RT and non RT parts working side by side on the same hardware. This adds complexity in the timely delivery of results, and the security and reliability of operation. The emerging of standards based on collaborations between competitors in the respective area is something, which is now commonly deployed in hardware and software. The standards are used to encourage competition between suppliers, or at least provide means for a

second source and hence reduce cost. Instead of traditional top-down development, systems are built bottom-up from a collection of independently developed components and subsystems.

Industrial development has changed to address this complexity. Development happens not any more in a single office but is spread around the world to make effective use of capabilities within a company, multi-site development. This requires different means of development as this obviously has an impact on communication. In order to reuse existing developments legacy hardware and software are deployed. Thus the effort is shifted from the development of new subsystems into the integration and support of legacy subsystems. The use of Commercial Off The Shelf (COTS) components and the outsourcing of well defined components to subcontractors is an attractive means to reduce the in-house development effort. Recently some industries have moved to open source developments. The public scrutiny by enthusiasts is considered a good way of making software reliable.

3 Implications and limitations on Operating Systems

The recent developments on embedded systems introduce new requirements on the infrastructures and consequently on the RTOS. Current RTOS techniques suffer from a number of limitations that have to be addressed.

Developing and testing system components and subsystems is a complex task in itself. However the main challenge appears at integration time, where emergent properties arise as resource sharing causes unpredictable behaviour. A system could potentially consist of a wide diverse of subsystems where the system integrator has varying possibility of control of function, reliability, resource usage, performance and so on. However, there are a number of legal and policy issues. One example is the potential infection of in-house code with public licenses like the GNU Public License (GPL). However the added complexity has meant that the problem of understanding basic components has increased dramatically, never mind the problems of understanding how they might be integrated and the resulting emergent properties.

Scheduling techniques tend to only concentrate on the timing aspects of systems. Although it is acknowledged that in recent years there has been some work on expanding scheduling to deal with other properties such as power. The key problem though is the majority of systems have a large number of properties and objectives to be satisfied. Some of the interactions between properties and objectives can be quite subtle, which means they are often over looked. For instance, in the design of avionics systems there is a link between the variations in when tasks execute and mechanical stress. The reason being is variations in timing lead to errors in data, causing noise and instability on signals, which leads to the moving surfaces of the aircraft (e.g., flaps) being moved more than necessary and hence mechanical stress. To date, little work has been done on truly multidisciplinary design, which has lead to a lack of available analysis techniques. Even if appropriate techniques were available, it is questionable how flexible and scalable the analysis would be for larger, different or more complex systems. The need to support multiple properties suggests that techniques and need to be more aware of the overall system problem and the environment that it is operating in. At the same time there is still a need for the RTOS to have appropriate abstractions from the rest of the system.

Furthermore, in order to cope with maintanance, bug fixes, and system extensions during the life time of an embedded system, these systems provide interfaces for interoperation. These interfaces may be maintanance ports in a car or specific command sequences issued to a sattelite. Furthermore some of these interfaces are an essential part of the functionality of systems, like networking in mobile phones or sattelites. These interfaces can be misused either deliberate maliciously or accidental and thus raise issues in the area of security and possibly safety. One scenario in the mobile phone industry, for example, is a reprogramming of the radio modem of a phone. This could lead a mobile phone to be used as a cell jammer. Paired with a clever written virus to distribute the code, similar to the recent attacks via bluetooth, this can produce serious damage to the mobile phone infrastructure.

Many of the techniques are biased to the worst-case 'hard' real-time systems. However a great deal of systems only have a few hard real-time requirements. Therefore designing the system for the absolute worst case is often un-realistic and results in fragile solutions that are prone to change. A key issue is designing for the worst case wastes valuable resources most of the time, which with current market pressures is not practical. Again, components and techniques need to be designed with QoS in mind whilst not disregarding the importance of selective rigidity. Some kind of Quality of Service (QoS) support might be required.

Finally, there is a lack of first principles/guidelines to build embedded systems and how the functionality is mapped to software tasks or hardware blocks. There is no structural way, no rule of thumb and often, the reasons for certain mapping are not understood.

In the next section, we will consider what some of the key requirements are and which emerging techniques are suited to meeting the requirements.

4 Emerging solutions

The problems can be distilled into the following requirements placed on the way systems are developed and in par-

ticular the infrastructures:

- Provide appropriate well-defined abstractions and interfaces.
- Provide design and analysis techniques to account for the complex interactions.
- Support flexible and robust execution. This is from two perspectives; change and failure.
- Partitioning is an essential ingredient to support integrity in systems and fault containment.
- Reduce the trusted computing base, which is the amount of code, which needs to be trusted to keep the system operational.
- Appropriate means for deciding whether components are mapped onto either hardware, software or a mix of the two (i.e., IP cores hosted on FPGAs).

The above is now discussed in the context of three application domains, namely avionics, automotive, and consumer electronics.

4.1 Avionics

Significant work has been performed within the avionics domain to achieve the stated objectives. The main body of work has been performed under the banner Integrated Modular Avionics (IMA) [1, 6, 7]. This work has been driven by the need to support incremental certification and technology transparency. Figure 1, which is based on the civil IMA standard (ARINC 653 [1]), shows the typical structure of an IMA architecture.

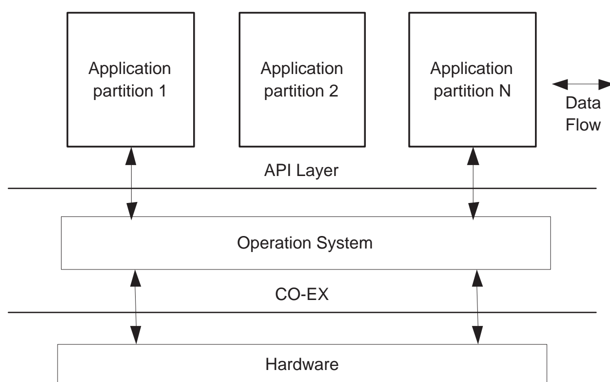


Figure 1. A typical structure of an IMA architecture.

The architecture features two key abstractions / interface layers, which are between the applications and the operating systems (APEX), and then the operating system and the hardware (COEX). Other key components of this architecture is that it represents a move away from federated systems (where a single computing device supports a single application) to modular systems where multiple applications may be supported on a single device. However more than that, the IMA architectures are being developed to support multiple criticality applications on a single device, which means there is a strong requirements for both temporal and spatial partitioning. Thie requirement is resolved through a mix of hardware support and the OS (by checking virtual memory look ups). Other complexities related to the operating system is the use of ”blueprints” that provide location transparency between communicating applications. The blueprints have to provide fast reliable resolution of references and be alterable to support reconfiguration. Other key initiatives related to IMA is the need for modular timing analysis to help support change. One solution proposed to this is the adoption of Reservation-Based Analysis (RBA) [9]. More recently work has commenced on assessing the parts of the hardware infrastructure that can be mapped onto Programmable Logic Devices, e.g., FPGAs. The aim of this work is to reduce chip counts and allow functional to be customised so that it can be made dependable. The IMA OS work represents a good example of work that fits with the requirements that have been identified during this paper. A number of IMA OS are in development and production but there are some key challenges still including making the OS calls more efficient and providing better support for dynamic reconfiguration.

4.2 Automotive

In the automotive domain, the embedded systems are distributed; hence the communications play a key role in the development process all the way from the design, to implementation and integration.

Traditionally, many OEMs have their own standard platforms for developing their embedded computer systems. This is not good from an integration point of view, when several subcontractors are required to adopt platform depending on which OEM that currently is its customer. The solution here is the effort towards standardization of non competitive elements by the initiation of several large consortia in order to agree on a common scalable electric / electronic architecture (e.g., AUTOSAR [2]) and a common scalable communication system (relying on FlexRay [8] together with existing standards such as CAN [10], LIN [11] and MOST [12]).

Looking at communications, automotive systems distribute data over fieldbuses. One way to do this is, e.g.,

based on specifications of how specific messages are to be used and what data and signals they are to contain (e.g., the J1939 [17] used in the truck and bus applications). This specification is then respected throughout the automotive system lifetime, resulting in a clear but somewhat inflexible networking interface.

Opposite to this early and static specification, the Volcano system [4], currently used by Volvo Car, provides tools for packaging data (signals) into message frames, both for CAN and other networks possibly interconnected with gateways. On top of this specification and signal packaging, it is possible to perform timing analysis of the system from a network point of view, and code can be generated for easy interfacing to data and signals. The Volcano approach allows for a greater degree of flexibility, compared to fixed specification of how data and signals are packed into message.

OSEK/VDX [13], which is a collection of widely used standards for automotive systems, specifies a scalable real-time operating system OSEK/VDX OS [16], communications with transparent communication services OSEK/VDX COM [14], and a network manager OSEK/VDX NM [15] allowing for easy integration of subsystems developed by different OEMs. OSEK/VDX provides reusability and portability of software by using abstract high level interfaces. OSEK/VDX COM allows for communications on a high level abstraction, without detailed knowledge on communication transmitters and recipients locations.

The latest automotive software standard is AUTOSAR, by the AUTOSAR consortia, scheduled to be complete in 2006. The goal of AUTOSAR is to create a global standard for basic software functions such as communications and diagnostics. From an integration point of view, AUTOSAR provides a Run-Time Environment (RTE) routing communications between software components regardless of their locations, both within a node and over networks. Tools allows for easy mapping of software onto the existing architecture of nodes (Electronic Control Units (ECUs)). This mapping is depicted in Figure 2. AUTOSAR is working towards integration of standardized tools relying on, e.g., operating system standards such as, e.g., OSEK/VDX OS, and various communication standards as, e.g., OSEK/VDX COM, FlexRay, CAN, LIN and MOST.

The function integration over the network is a less complex task compared to the integration at the application level. Looking at application level, while designing and specifying the automotive system, model based development is used by some OEMs. Component based development is not used systematically, however, possibly by subcontractors of specific subsystems. Also, the introduction of AUTOSAR will increase the usage of component based software development.

To further increase the flexibility of the development process, some OEMs use a physical model at an early stage for implementation, integration and testing of parts and subsystems. This physical model is used together with modelling tools, such as Statemate and MATLAB/Simulink, to simulate parts and subsystems, environments and specific runtime scenarios. Models of subsystems allows for integration at an early stage in the development process. However, an issue is the exchanging of models between subcontractors and OEMs since these models need to have proper abstractions, not revealing too little or too much information.

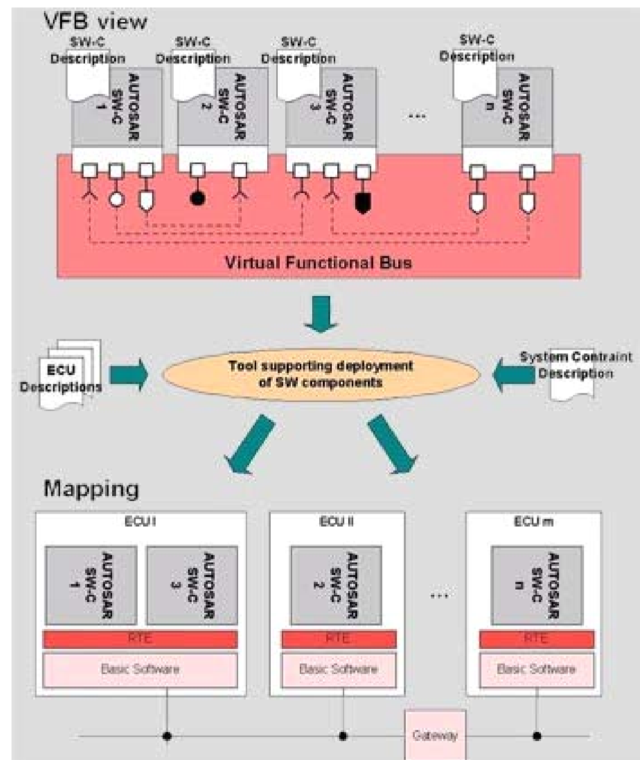


Figure 2. AUTOSAR Virtual Functional Bus and ECU mapping [3].

Furthermore, there are the issue of litigation, if subsystems of different subcontractors are integrated onto a single ECU. In the case of a major, but isolated fault it is important to clearly identify the faulty component. Compartementalisation of applications serves on one hand to isolate faults and on the other hand allows an easier identification of the faulty application.

4.3 Consumer electronics

The consumer electronics companies start to recognize the need for industrial standards to cope with the new trends in software and hardware. Mastering system complexity is

not any more the task of a single engineer or a single company. A number of initiatives have been initiated that bring together various CE companies in an attempt to achieve industry-wide standards that benefit all. In this spirit, the Universal Home API (UHAPI) [18] is a hardware independent API that aims at developing and maintaining sustainable CE products. This API favourers the growth of the ecosystem around the products by enabling independent software vendors (ISVs) to create middleware and applications components that easily interact.

Another initiative that directly relates to the OS is the Consumer Electronics Linux Forum (CELF) [5], which advocates for a open source platform for consumer electronics (CE) devices. CELF intends to leverage the benefits of the open source community and process to maximize the re-use of common solutions to common problems.

On the other hand the use of commodity operating systems on embedded devices introduces the problem of millions of lines of code needed to be trusted not to be breakable via denial of service attacks or spreading viruses. This calls for removing any functionality, which does not need to be priviled from the kernel and moved into the user space, supported by proper partitioning.

5 Way forward

Surely, the current challenges facing real time operating systems within these and other embedded applications domains are challenging at best, and will only continue to grow. How should developers and designers of RTOS's proceed to meet the growing challenges? Several issues are clear and must be considered immediately for inclusion in next generation RTOS's.

First, RTOS designers should consider meeting the growing requirements provided in technology growth by exploiting and not fighting Moore's. Hardware/software co-design of RTOS's have historically provided increased scheduling precision. As Moore's law provides a doubling of transistor capabilities every three years, this can be exploited to offer a scaled increase in RTOS performance and capability that cannot be equaled in pure software solutions. With current software based RTOS's, increased functionality requires more lines of sequential code exacerbating already difficult maintenance of critical sections and additional timing overhead in context switching and operating system processing. By migrating portions of the operating system into hardware, Moore's law enables a migration from the temporal to the spatial domain, and enables functionality to increase concurrenty within the transistors.

Second, appropriate abstract interfaces must be formalized to support the rapid seamless insertion of additional hardware and software application components within a system centric framework. This capability is foundational to many of the existing issues, including dealing with increased complexity through higher level abstractions and supporting component reuse to increase times to market for hardware as well as software components. A higher level abstract interface also brings the benefits of abstract type checking into the hardware/software co-design domain, which provides additional dependability, modularity, and maintainability.

Third, security must be elevated to a first class design constraint for RTOS's. Fundamental issues of atomic sequencing between secure states should be considered as both a hardware and software issue in order to eliminate classic time of check to time of access breeches. Current monolithic operating system organizations have also shown the vulnerability of single supervisor mode, unlimited access to global state information. Thus next generation RTOS development should consider built in compartmentalization of operating system functionality, and provide a framework for the development of both soft and hard secure IP within systems that support unsecure components. However, this can not be solved by the RTOS alone. Restricting access to global states inevitably means memory and device access protection. This requires processors to be equipped with memory management units.

Fourth, scheduling should be expanded to include system resource utilization in meeting application timing deadlines. To support expanded schedulability, RTOS's will be required to perform fast non-invasive resource monitoring and scheduling of dynamically time varying reconfigurable resources to meet more complex and interdependent functional requirements.

6 Summary

In this paper we have collected some of the main development trends in embedded systems for the automtive, avionics and consumer electronics domains. Increasing complexity require new approaches to system composition for both hardware and software. In the hardware side, flexibility is enabled by the use of heterogeneaous Systems on Chip, Networks on Chip and FPGAs. For the software, components are beeing developed multi-site and multi vendor. For cost-efficiency reasons, the system resources are being shared introducing unpredictability in the integrated system. To still maintain the traditional "-ilities" some of the limitations on current RTOS have to be addressed. Some of this limitations include

- Lack of first principles
- Interference due to resource sharing not explicitly considered by analisys techniques
- Security

- Lack of QoS support

Finally, emerging solutions for the application domains were discussed.

References

[1] ARINC. *ARINC 653: Avionics Application Software Standard Interface (Draft 15)*. Airlines Electronic Engineering Committee (AEEC), June 17th, 1996.
[2] AUTOSAR. Homepage of Automotive Open System Architecture (AUTOSAR). http://www.autosar.org/.
[3] AUTOSAR Web Content, V22.4. Available 2005-06-06 from: http://www.autosar.org/.
[4] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a revolution in on-board communication. *Volvo Technology Report 98-12-10*, 1998.
[5] Consumer Electronics Linux Forum. http://www.celinuxforum.org/.
[6] R. A. Edwards. ASAAC phase I harmonized concept summary. In *Proceedings ERA Avionics Conference and Exhibition*, London, UK, 1994.
[7] D. Field and J. Kemp. The ASAAC Programme, NATO HQ, Brussels, July 20th, 2000. Available from: http://www.safsec.com/safsec_files/resources/nato_p 1.ppt.
[8] FlexRay Consortium. http://www.flexray.com/.
[9] A. Grigg and N. C. Audsley. Towards the timing analysis of integrated modular avionics systems. *In Proceedings ERA Avionics Conference and Exhibition*, pages 3.1.1–3.1.12, 1997.
[10] ISO 11898. Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. *International Standards Organisation (ISO)*, ISO Standard-11898, Nov 1993.
[11] LIN Consortium. LIN - Local Interconnect Network. http://-www.lin-subbus.org/.
[12] MOST Cooperation. MOST - Media Oriented Systems Transport. http://www.mostcooperation.com/.
[13] OSEK/VDX. Open Systems and the Corresponding Interfaces for Automotive Electronics. http://www.osek-vdx.org/.
[14] OSEK/VDX-Communication. Version 3.0.3, July 2004. http://www.osek-vdx.org/mirror/OSEKCOM303.pdf.
[15] OSEK/VDX-Network Management. Version 2.5.3, July 2004. http://www.osek-vdx.org/mirror/nm253.pdf.
[16] OSEK/VDX-Operating System. Version 2.2.2, July 2004. http://www.osek-vdx.org/mirror/os222.pdf.
[17] SAE J1939 Standard. The Society of Automotive Engineers (SAE) Truck and Bus Control and Communications Subcommittee. *SAE J1939 Standards Collection*, 2004.
[18] Universal Home API (UHAPI) Home Page. http://www.uhapi.org/home.

Operating Systems and Supporting Architectures for Embedded Real-Time Systems

Neil Audsley Rui Gao Ameet Patil Paul Usher Jack Whitham
Real-Time Systems Group,
Department of Computer Science, University of York, York YO10 5DD, UK
Email:{neil,rgao,appatil,usher,jack}@cs.york.ac.uk

ABSTRACT

Distributed embedded real-time systems (DERTS) require efficient Operating Systems (OSs) and supporting hardware. However, usual approaches involve using relatively static OSs, that have limited configurability. Whilst this may provide a clean OS API that abstracts the underlying hardware and provides resource management, this does not necessarily lead to the most efficient overall system implementation for a particular ERTS. This paper examines four complementary approaches that together enable efficient application specific OS implementations for distributed DERTS.

1. INTRODUCTION

The implementation of an efficient Distributed Embedded Real-Time System (DERTS) necessitates the tuning of supporting Operating System (OS) and hardware architecture in order to provide the performance required by the application. However, this must be achieved whilst maintaining OS application interfaces which provide abstraction of the OS and hardware platform, together with resource management of system resources (including hardware and software, eg. OS I/O buffers). A key challenge therefore, is the specialisation of the OS and platform to the needs of a particular DERTS implementation. This paper describes a multi-faceted approach to this problem involving:

- *Application-Specific Resource Management Policies*:
  provision of reflection within the operating system to enable run-time application specific resource management policies (for scheduling, memory, power etc.);
- *Efficient Remote Resource Access*:
  recognition and provision of distributed resource access at low-level in the operating system to enable low resource systems to efficiently utilise remote resources;
- *Hardware Support for OS Functions*:
  establishment of a platform architecture that efficiently supports operating system and application functions by providing ASIP (application specific instruction set CPUs) and co-processor elements, thus allowing direct implementation of key operating system services on hardware, e.g. communications.

The context assumed by the paper is that of systems on chip (SoC) where the hardware platform consists of a number of separate fixed functions implemented on the same device (e.g. CPU, RAM, communications), together with programmable hardware elements (eg. FPGA) that allow custom coprocessors to be implemented within the platform. This platform architecture is consistent with many existing platform architectures, eg. FPGAs with embedded CPUs [2], hybrid CPUs [1], ASIP platforms [7]. These platforms are being increasingly used for low-cost, low-resource embedded real-time systems, but allow application-specific functions to be implemented directly in hardware, thus providing massive flexibility.

This paper describes a number of threads of work currently being undertaken at York, all contributing to the challenge of efficient implementation of DERTS. Section 2 provides background and motivation for the approach adopted within the paper. The remaining sections provide further technical detail on different aspects of our approach.

2. MOTIVATION

The design of a Distributed Embedded Real-Time System has to geared towards application specific needs. The system is tuned to perform optimally for the application it is being developed for. However, tweaking the hardware or the Real-Time Operating System (RTOS) for every such application is a very costly approach. Conventionally, the design concentrates on standard hardware and using a general purpose RTOS that provides standard interfaces (eg. POSIX) to develop the applications.

This approach fails to satisfy the application specific requirements, thereby leading to poor application performance. Both the platform and RTOS are built for the generic case rather than application specific requirements. From the perspective of the RTOS, to be application specific requires it to undergo dynamic (Eg. at run-time) or static (Eg. at compile time) changes. For example: given that no single scheduling policy can satisfy all the different application domains, the RTOS needs to implement a scheduling policy best suited for the application it executes, potentially application-specific. From a platform perspective, fixed components provide difficulties in efficient system implementation. Exploiting technologies that allow the platform to better match the actual needs of the RTOS and application are essential for efficient implementation. For example: utilising architectures allowing ASIP instructions and co-processors enables key parts of the application to be more efficiently executed (ie. less time/power).

For RTOSs, there have been several different approaches proposed to counter this problem in the form of APIs, component based development, etc. However, these approaches

provide solution to only a part of the problem. For example: there exists API for changing scheduling policy in the RTOS, but no API for changing any other module like memory management. Component based RTOSs do address the problem but there are problem with integrating the various component to work together. Or if the component are designed to work together then the amount of flexibility provided by the system is lost in the process.

Also note that, the RTOS usually comes with additional functionality that the application does not need. For example: in a networking environment where security is not a major issue, the application might not need the TCP/IP stack at all for network communication. However, it is compelled to use it since the RTOS cannot pack network packets without TCP/IP thus introducing a constant overhead to network communication.

There is a need for a dynamic system (including RTOS and hardware) that can change itself to meet the application specific requirements. In remainder of this paper provides an integrated approach that considers both RTOS and platform aspects of this challenge.

3. REFLECTIVE RTOS

Reflection is a mechanism by which a program code or application becomes 'self-aware', checks its progress and can change itself or its behaviour dynamically at runtime or statically at compile time [10]. This change can occur by changing data structures, the program code itself, or sometimes even the semantics of the language its written in. To facilitate this, the application or program code has to have knowledge about the data structures, language semantics, etc. The process by which this information is provided to it is called *'Reification'*.

The Reflection model consists of a base-level and one or more meta-level forming a structure called *'Reflective tower'*. The code in the meta-level is responsible to analyse the reified information, intercept the necessary calls from or to the base-level and affect any change if required. A protocol defined so as to establish a mechanism by which the meta-level entities introspect (analyse), intercede (Eg. by intercepting calls to or from base-level) and affect change to the base-level is called the *'Meta-Object Protocol (MOP)'* [8]. In reflection the meta-level code can form a causal link (two objects are said to be causally linked to each other when a change initiated by one affects the other [10]) with the data structures in the base-level to affect a change directly.

The mechanism of Reflection has been widely used in object-oriented programming, object-oriented databases, artificial intelligence, virtual machines, etc. [10]. In terms of OSs, reflection is used to allow applications to access key OS data structures to obtain information pertaining to the current system performance and resource management policies (Eg. scheduling). An application is then able to modify or introduce new policies into the RTOS with the help of reflective system modules that intercept certain events or function calls to change the overall behaviour of the application and the system. The Reflective OS on the other hand is able to obtain critical information from the applications and change its structure/behaviour dynamically to adapt to the application.

A general purpose RTOS is always built without the knowledge of applications that would execute upon it, rather it is built for the general case. Such RTOS comes with extra functionality which real-time embedded applications may or may not require (Eg. networking, graphics). Recent trend has been to use a component based RTOS where the RTOS is built by combining only the required components for a particular application. This however poses a problem with integrating the different components together. Dependencies arise between different components thereby again adding extra functionality that was not required. Developing components that work together adds several restriction to their development methodology there by compromising the overall system flexibility.

Another approach taken to overcome this problem is by providing APIs which the application can use to change certain policies in the RTOS to their specific requirements. For example: in MaRTE [11] OS, the applications use the API to change the scheduling policy being used to schedule the application threads. On the other hand, SHaRK [6] provides with a similar API to develop applications that use their own scheduling policies. Evaluation results of these approaches show a considerable amount of overhead added to the system there by making the approach infeasible [11].

Also, changing scheduling policy depending on application requirement is not the only thing that needs to be changed in an RTOS. Each and every module in an RTOS may need to undergo a change (Eg. memory management, networking, graphical display etc.). What we need is a generic solution covering all the aspects rather than just a fraction of it (Eg. scheduling API). We have designed and implemented a reflective real-time OS – DAMROS [3, 4]. Fig. 1 shows our proposed generic reflective OS framework [9].

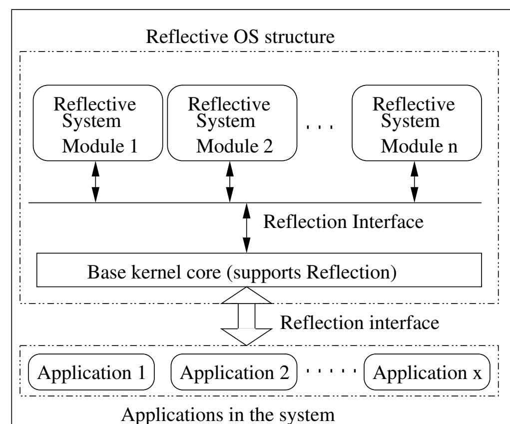


Figure 1: Generic Reflective OS framework

The framework consists of a Base kernel core that provides support for reflection in the form of reflection interface for system modules/applications to reify information, introspect and intercept the base-level. The System modules (eg. scheduler) are designed to be completely reflective. A reflective system module (eg. a reflective scheduler) makes use of the interface provided to analyse reified information and take intuitive steps to intercept and change behaviour of the base-level module. Fig. 1 shows several reflective system modules as well as the applications using the in-kernel reflection interface. Similar to the reflective system modules, the applications can also be reflective. The meta-level code in the reflective applications (not shown in fig. 1) can

analyse the reified information from system and change the behaviour of the application.

The reflective system modules (see fig.2) implement a generic policy at the base-level. For example: in case of a reflective scheduler, a simple round-robin scheduling policy or an optimised scheduling policy can be implemented at the base level. Depending on the application requirement, the meta-level code can then change this base-level policy to an application specific one either at run time or statically.

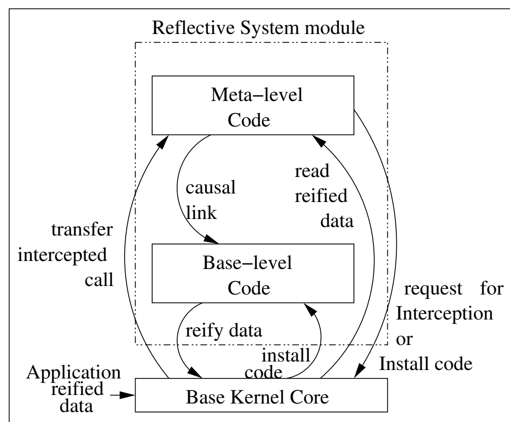


Figure 2: Reflective System module

The advantage of this approach is that we can develop a system perfectly tuned for the application in hand. During testing, the system with default policies can be used to run applications that are allowed to make changes to suit their needs. Depending on the performance metrics obtained after the changes brought in, we can either change the default policy permanently or keep the existing one for deployment.

4. EFFICIENT REMOTE RESOURCE ACCESS

Typical RTOS implementations support distribution by adding network capabilities, e.g. network protocol stack, without any architectural change. This approach may be appropriate for systems that have sufficient resources to meet all the application's functional and non-functional requirements. But for low resource systems such an approach often imposes a significant overhead (eg. applications wishing to access remote resources cannot do so without significant part of the involvement of local and remote nodes).

A key contention of this paper, is that they should be no distinction between local and remote resources i.e. remote resources are accessed and used in the same manner as local resources. This is especially important for systems where it is impractical, or infeasible, to equip all nodes with sufficient resources to meet their worst-case requirements, e.g sensor networks. To illustrate this consider memory access in a multiprocessor (e.g. SMP) or distributed memory systems. In such systems, accessing remote memory uses the same mechanisms as local memory, noting that contentions have to be managed. For other resources, this efficient mechanism is not available. Usually, the access is via a remote network operation via network protocol stacks, imposing a heavy overload, both locally and remotely. Clearly this has implications on an embedded real-time system's ability to meet any form of timing requirement.

To address the issues raised above, an efficient remote resource access mechanism has been proposed [13]. Essentially, the approach reduces the overheads of the network stack and virtual file system (often used to name and access remote resources) as much as possible. This is achieved by structuring the RTOS functionality on a distribution layer, which provides efficient access to remote resources. This permits the virtual file system to directly access the device drivers and devices on remote nodes, so bypassing the inevitable overhead of the remote OS, (part of the) network stack and virtual file system. Clearly, resource control and management within such a system becomes distributed. In turn, network stack overhead can be reduced by using location aware techniques, whereby local nodes can be accessed far more efficiently and remote nodes, and without utilising full TCP/IP protocol stack.

5. HARDWARE PLATFORM

When an embedded system is built on a flexible platform, potentially including field-programmable elements, itself can be customised to match the requirements of the RTOS and the application [14]. Application Specific Instruction Processors (ASIPs) provide a popular approach for doing this. Frequently executed sections of an application task can be optimised by replacing a number of machine instructions with a single instruction that carries out the same job, but does it faster and with fewer memory accesses. Careful use of this optimisation technology can provide a substantial speed increase [12]. An ASIP is illustrated in Figure 3.

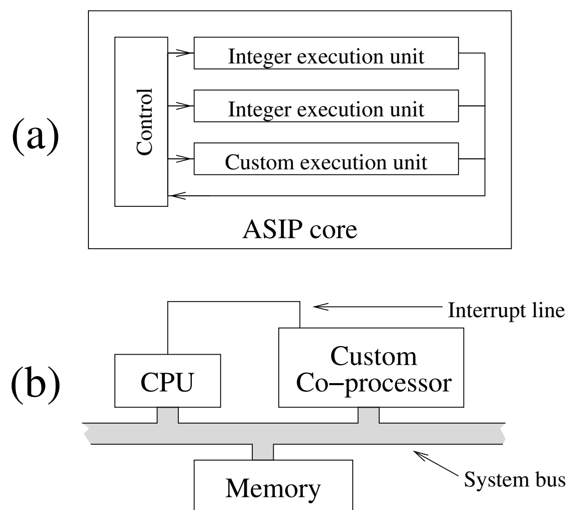


Figure 3: (a): an ASIP core provides customisable execution units on the processor data path. (b): a customised co-processor is generally an additional memory mapped device on the system bus, connected to an interrupt line which is used to signal task completion.

In contrast, conventional CPUs provide a fixed set of instructions and functions. Instructions and functions assist the operating system by providing fast context switch and memory management, for example. However, these are provided in a generic form that are not tuned to the specific requirements of an application.

It is noted that manufacturers of the tools used to customise ASIPs to particular applications have not yet fully considered the implications of running an RTOS on an ASIP. The tools provided are targeted at finding the frequently executed parts of single-threaded applications with full control of the system. This simple model does not take into account the effects of task interaction: for example, there is no attempt to relate the profiling information to task timing analysis. The choice of optimisation point must make use of this information if optimisations are to reduce response times and improve overall performance.

ASIP improvements can be targeted at the RTOS itself, but this is unlikely to be useful, as many of the operations carried out by an OS are I/O or memory bound. ASIPs can optimise operations only with the processor. However, improvements to an RTOS can allow a developer to obtain information about task interaction, and help decisions about ASIP optimisation to be made. A task-aware online profiler is required, and this can be made available within the RTOS as an extra task.

A wider class of optimisations can be made by adding a customised co-processor to complement the main processor. The customised co-processor essentially replaces all or part of a task that would otherwise execute as software within the main processor, and may reduce power consumption and make more CPU time available to other tasks. A co-processor architecture is illustrated in Figure 3. The co-processor can execute its task in parallel with another task on the main processor, and signal completion using an interrupt. This approach requires special driver support in the RTOS, and again it is important to determine which parts of which tasks are best handled by co-processors: information which can only be obtained by careful analysis of task interaction. Analysis is more complex than the ASIP case, as the use of the co-processor is likely to be exclusive.

Future RTOSs will require profiling features and support for co-processors in order to allow systems to make effective use of the hardware that they use. Substantial improvements to execution time and response time can be made, provided that optimisations are applied correctly.

5.1 Example RTOS Component as Coprocessor

For architectures and platforms that allow application-specific components to be programmed into hardware, e.g. into a coprocessor, a key consideration is the choice of functions that are placed into a coprocessor. An obvious component is that of communications, where a coprocessor implementation can prevent interrupts due to the communications subsystem reaching the CPU, and can enable application processing and communications to occur in parallel.

Communication between ubiquitous devices has become a major challenge in recent years. Due to enormous growth in the communications sector, network protocols keep changing over a period of time. Thus, making the older devices incompatible with the new ones. Communication via TCP/IP does provide a standard, but it is too costly and heavy to implement on a small device. New technologies like Zig-Bee, Bluetooth, etc. pose a very light weight solution. However, they are not inter-operable. Meaning – a device with bluetooth capability cannot communicate with a device with zig-bee capability. A way around this problem is to make use of an intermediate device which has bluetooth as well as Zib-Bee capability. Then again, there is no traditional or standard way of doing it when it comes to multiple vendors and multiple devices. There is a great demand for communication between different heterogeneous devices.

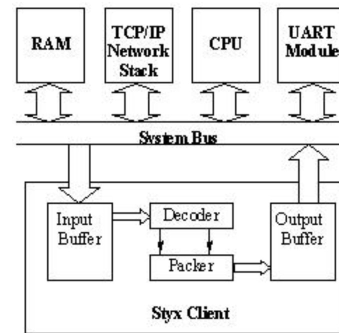


Figure 4: Styx Hardware Coprocessor Architecture – Client

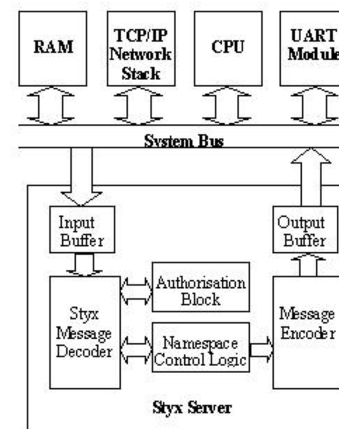


Figure 5: Styx Hardware Coprocessor Architecture – Server

The Styx [5] networking protocol can offer a general solution to this problem. Styx is an application layer protocol that will work on any standard communication medium like ethernet (either using TCP/IP, UDP, etc.), serial line, radio, bluetooth, zig-bee, etc. Styx requires that the underlying communications media provides reliable in order messages. Applications that want to take advantage of Styx are provided with a simple interface, namely files, using standard open, read, write, and close operations. Styx presents all the networked (ie. remote) resources and devices as files to the application. The application on the other hand communicate with the network devices or make use of the network resource by just reading and writing to files.

As in all UNIX flavours, files have their own security. The same applies to the Styx files. Another unique feature of Styx is that it allows temporal isolation between different application on a system using the same network resources. Any change made by one application to the Styx namespace will not affect other applications.

Current work is implementing the Styx component as an FPGA coprocessor (within an OpenRisc architecture). The architecture of the component is given in Figures 4 and 5.

The server handles requests from the client, decoding the specifc request (using the Message Decoder), checking that the client has permission for that operation (using the Authorisation Block). Data is manipulated in the namespace (using the Namespace Control Logic), perhaps to read or write, then the reply message is packed and sent (using the Message Encoder). The implementation (combining server and client) is small, being achieved in under 35K gates currently.

6. CONCLUSIONS

In this paper we have briefly described a multi-faceted approach to the provision of application specific embedded real-time operating systems. This is based upon an operating system that enables applications to reflect on their performance, and to change resource management policies dynamically to reflect the current state of the application. In conjunction with a reflective operating system structure, we provide an efficient method for remote access of resources, by considering distribution within the RTOS structure. This enables remote resource access with minimal network stack and virtual file system overhead. Fundamental to this approach, is the implementation of the operating system upon a flexible platform that can be tuned to provide application specific functions within the hardware. This is considered at the instruction level (ie. ASIP) where key parts of the application and RTOS can be accelerated within single instructions; and at the coprocessor level, where larger application and RTOS functions (eg. communications) can be implemented directly in hardware.

7. REFERENCES

[1] Excalibur embedded processor solutions, Online: http://www.altera.com.

[2] Virtex-pro fpgas, Online: http://www.xilinx.com.

[3] A. Patil and N. Audsley. VRHS: an Application Specific Reflective Hierarchical Scheduler. In *Submission to SOSP 2005*, Brighton, UK.

[4] A. Patil and N. Audsley. Implementing Application-Specific RTOS Policies using Reflection. In *Proceedings of the 11th IEEE Real-time and Embedded Technology and Applications Symposium*, pages 438–447, San Francisco, 2005.

[5] S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey, and P. Winterbottom. The inferno operating system. *Bell Labs Technical Journal*, 2(1), 1997.

[6] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[7] M. H. S. Hauck, T. W. Fry, and J. Kao. The Chimera Reconfigurable Functional Unit. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[8] J. Malenfant, M. Jaques, and F.-N. Demers. A Tutorial on Behavioral Reflection and its Implementation. In *Proceedings of the Reflection 96 Conference, Gregor Kiczales, editor, pp. 1-20, San Francisco, California, USA*, April 1996.

[9] A. Patil and N. Audsley. An Application Adaptive Generic Module-based Reflective Framework for Real-time Operating Systems. In *Proceedings of the 25th IEEE Work in Progress session of Real-time Systems Symposium*, Lisbon, 2004.

[10] Patrick Rogers. *Software Fault Tolerance, Reflection and the Ada Programming Language*. PhD thesis, University of York, UK, October 2003.

[11] M. A. Rivas and M. G. Harbour. POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 67–75, June 2002.

[12] Tensilica Corporation. Accelerating existing C code (accessed 1 Apr 05). `http://www.tensilica.com/html/-accelerating_existing_c_code.html`.

[13] P. Usher and N. Audsley. Improving the Efficiency of Remote Resource Usage in Distributed Real-Time Systems. In *Proceedings of the 25th IEEE Work in Progress session of Real-time Systems Symposium*, Lisbon, 2004.

[14] J. Whitham and N. Audsley. ASIP Instruction Selection for Real-Time Systems. In *Submission to ISSS+CODES*, 2005.

The FIRST Application Programming Interface

(Invited Talk)

Michael Gonzàlez Harbour
Dpto. de Electrónica y Computadores, Universidad de Cantabria, Santander, Spain
e-mail: mgh@unican.es

Abstract

Scheduling theory generally assumes that real-time systems are mostly composed of activities with hard real-time requirements, many systems are built today by composing different application or components in the same system, leading to a mixture of many different kinds of requirements with small parts of the system having hard real-time requirements and other larger parts with requirements for more flexible scheduling, taking into account quality of service. Hard real-time scheduling techniques are extremely pessimistic for the latter part of the application, and consequently it is necessary to use techniques that let the system resources be fully utilized to achieve the highest possible quality.

In this talk, we presents a framework for a scheduling architecture that provides the ability to compose several applications or components into the system, and to flexibly schedule the available resources while guaranteeing hard real-time requirements. The FIRST Scheduling Framework (FSF) is independent of the underlying implementation, and can run on different underlying scheduling strategies. It is based on establishing service contracts that represent the complex and flexible requirements of the application, and which are managed by the underlying system to provide the required level of service.

*This work has been funded by the Commission of the European Communities under contract IST-2001-34140 (FIRST project)

The need for configurable and flexible scheduling in a RTOS aspiring to solve contemporary problems

(Invited Talk)

Thorbjörn Jemander
Enea Epact AB, Teknikringen 8, Linköping, Sweden

Abstract

As embedded systems increase in size and complexity, the demands of the underlying operating system get greater for each generation. For a long time, the embedded systems remaied rather small, and simple operating systems were used, if any. In modern embedded systems there are hundreds of processes on each CPU, all put together to minimize the number of chips and hence reduce cost. This means that, today, there is a number of conflicting demands of the various processes on a single CPU. There are hard and soft deadlines, combined with non-real-time processes. There must the a high through- put, yet quick response times and deterministic execution times. There is a fixed priority scheme, yet the priorities are changed run-time. All these problems have to today be solved with, in principle, a fixed priority scheduler lacking awareness of fundamental concepts as deadline, execution time, process time buget etc. This is not adequate for the problems at hand, and therefore modern RTOSes have to supply mechanisms to a more flexible, configurable and timing- and/or load aware scheduling.

Curriculum Vitae

Thorbjörn Jemander is a real-time systems specialist at Enea Epact, working with design and analysis of real-time systems. He is the systems architect of the Enea Advanced Scheduling Framework and is engaged in the promotion of real-time analysis tools and methodology. In 1991 he earned a BSc in computer engineering, 1996 a MSc in applied physics and electrical engineering, 2001 a PhD in solid state quantum physics.

An overview of the XtratuM nanokernel *

M. Masmano, I. Ripoll, and A. Crespo
Universidad Politécnica de Valencia, Spain.
{mmasmano,iripoll,alfons}@disca.upv.es

Abstract

This paper presents a new nanokernel (XtratuM) which is aimed for executing several operating systems (where, at least, one of them is a real-time operating system) in the same hardware with temporal and spatial isolation.

Simplicity is the main idea behind of its design, therefore XtratuM can be defined as a thin layer of software which abstract the essential devices to run a kernel: the memory, the timers and the interrupts.

Besides, this paper presents the ARINC specification 653-1, an interface specification which allows to build high-reliability applications, being this interface a solid candidate to be the future XtratuM interface.

Keywords: Nanokernel, real-time, embedded system

1 Introduction

The word XtratuM derives from the latin term *substratum* referring to the layer of material which gives support to the upper layers.

In computer science, this term is used to designate a software hiding the programming complexity of the lower levels and supplying a functionality to the upper levels. From this point of view, XtratuM can be defined as a layer which is directly inserted between the hardware and others Operating Systems (OSes), easing the programming of these OSes as well as allowing to execute all of them in an isolated and concurrent way.

An Operating System (OS) can be defined as an abstraction layer between the physical hardware and the applications [6, 8, 7], which hides the underlying hardware (processor, physical memory, storage mediums, etc) and provides a high-level abstraction of the machine instead (file system, process, threads, etc).

The use of an OS provides a big quantity of benefits to the applications mainly because the complexity of the hardware is hidden, simplifying the design and increasing the portability of the applications. Before the apparition of the OSes, the applications themselves were the responsible of setting up and managing the underlying hardware. Hence, these applications used to be designed for an machine with an specific configuration (quantity of memory, storage space, speed, etc), and therefore stopping working when this configuration was changed.

The design of an OS tend to be mainly guided by the requirements of the applications that will be run on it. It means that each OS just enables the use of a concrete range of applications, whereas it is not optimum or even it is useless when is used by applications with different requirements. For instance, the Windows OS was designed to supply a powerful, intuitive graphic interface, easy to use for the beginners. Windows is a clear example of an OS which does not provide any hard real-time capability (it is important to note that Windows offers the possibility of using an scheduler based on fixed priorities, but it does not turn windows into a hard real-time operating system).

Another example is the OSes used in the mobile phones, these kind of OSes are designed to be executed on embedded systems with low resources (battery, processing power, memory and storage space). Often these systems tend to implement a poor graphic environment. Therefore heavy graphic applications can not be executed on mobile phones' OSes. A last example could be the real-time OSes whose timing behaviour is deterministic. These systems are outstanding to execute applications with timing requirements. However, these OSes use to

*This work has been supported by the European Commission project number IST-2001-35102 (OCERA).

lack in suppling a friendly user interface.

Current processors are more and more powerful allowing to execute more and more complex applications. These complex applications usually can be divided in several parts with different requirements. An example of this can be a engine control program with a graphic monitor. The application can be split in two parts: the control algorithm which interacts with the engine, with hard real-time and fault-tolerance requirements (where using a hard RTOS is compulsory) and the monitor, with graphical requirements (where it would be nice to have a desktop OS with graphic capabilities like Windows or Linux).

Enhancing a general purpose OS with real-time capabilities has already been tried with disappointing results. The most satisfactory results have been obtained by the RTLinux approach, adding a software layer (a hard RTOS) beneath a general purpose OS (Linux or FreeBSD). This software layer virtualises interrupts and executes the general purpose OS in the background as the lowest priority task of the system.

This approach (running a hard real-time OS jointly with general purpose OS within the same machine) shows that running several OSes on the same machine enhances the execution of the applications with disperse requirements.

Some existing techniques to execute several OSes in the same machine are described below:

1. Running several OSes (guest OSes) on the top of another OS (host OS) through a virtualisation program: This technique consist in, on the top of a host OS, creating a complete virtual hardware machine. Thus, allowing the execution of the guest OS even if it were compiled for a different physical hardware architecture. The advantage of this kind of approach is that the guest OS can be directly executed (no modifications to the OS are required). Nevertheless, this technique also presents an important drawback: the guest OS can not be directly executed by the real processor but it has to be interpreted by the virtualisation program with a lost of performances. Therefore this approach is not useful to satisfy complex application requirements. Examples of virtual machines are: VMWare [9], Plex86 [3], win4lin [5].

2. Multiplexing the physical hardware between several OSes (guest OSes): Usually, OSes are internally structured as a set of building blocks (see figure 1). These blocks can be divided in two categories, the device drivers: network-card drivers, PCI bus drivers, SCSI drivers, USB drivers, etc. And the abstractions of the hardware: memory manager, virtual filesystem infrastructure, network stack, etc. This method is implemented inserting a software layer beneath of the guest OSes. This software layer[1] takes over the real hardware and provides a virtual one to the guest OSes. An important disadvantage of this approach is that the current hardware architectures can not be completely virtualised, therefore the device drivers of the guest OSes have to be hacked to deal with virtual drivers rather real ones. Examples of implementations of this software layer are: Fiasco [4], Adeos [10], RTLinux [11], etc.

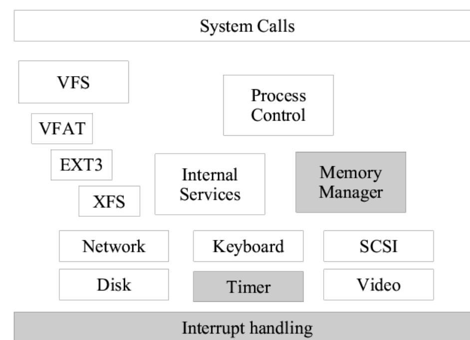


Figure 1: Classical operating system internal structure (simplified).

The results achieved using the second approach have encouraged us to design a new nanokernel, called XtratuM, which has been explicitly designed to support the execution of at least one real-time OSes jointly one or more general OSes.

2 The ARINC specification 653-x

The ARINC specification 653-x defines a general-purpose APEX (APplication/EXecutive) between the OS of an

[1] There are several ways of call this layer depending on the way it virtualise the hardware, its size and other implementation details. This denomination can be nanokernel, picokernel, and exokernel[2].

avionics computer resource and the application software. Therefore, this specification was designed to be used by high-reliable applications.

The main aim of the ARINC specification 653 is to define a group of substandards (phases), each of them focused in a different degree of functionality-criticality of the system.

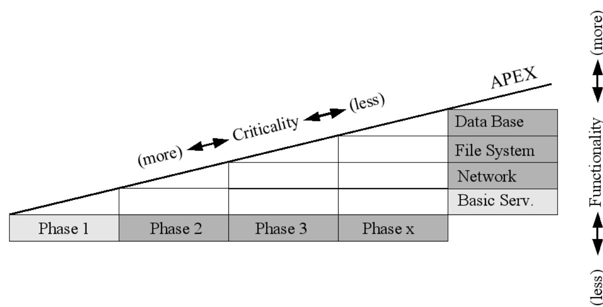


Figure 2: ARINC 653's phases.

The figure 2 shows the services which are offered in each phase of the standard ARINC 653, depending on the criticality of the system and the desired functionality. Currently, only the ARINC 653-1 (filled with light grey), phase for the most critical systems with the least possible functionality. The rest phases (filled with dark grey) are still in development and have not been defined yet. All services defined in these still-in-development phases have been written down as possible examples.

Basic services offered by the ARINC 653-1 (the first phase, and currently the only one implemented) are the following:

- Partitions management: a partition is described as a program, composed by code and dates with a single, isolate memory address space.
- Processes management: in the ARINC 653-x, a process is described as an execution unit within a partition.
- Time management: these services permit to read current time, programming a timer, stopping an existing timer, etc.
- Memory management: the ARINC specification 653-x does not supply any service to allocate or release dynamic memory. All memory is statically allocated at compilation time.
- Inter-partition communication: These services define the mechanisms used to communicate some partition between them.
- Intra-partition communication: These services define the mechanisms used by several processes belonging to a partition to communicate between them.
- The health monitor: the health monitor is the mechanism proposed by the ARINC specification 653-x to recover or kill partition after a fail has happened.

3 XtratuM architecture

The main idea behind of the design of the architecture of XtratuM is to virtualise the minimal possible parts of the hardware to achieve the execution, in a concurrent way, of several OSes. Where some of these OSes (or all of them) could be real-time kernels.

However, unlike some existing nanokernels (for example, the L4 μ-kernels family[1]) XtratuM does not virtualise the whole hardware architecture, but it just multiplexes the most essential parts of the hardware to execute in a concurrent ways several OSes. Each OSes should be aware how to use the parts of the hardware which have not been virtualised by XtratuM.

XtratuM basically offers the following virtualisations for the guest OSes:

- Interrupts: Taking over interrupts on a computer is a synonymous of controlling the whole machine. Once XtratuM is started up, it is the only one who really controls hardware interrupts and, of course, the only one who is able to disable/enable real interrupts. An API is offered, enabling to the guest OSes to deal with the virtual interrupts which allows basically enabling/disabling virtual interrupts, installing interrupt and exception handlers, and so on.
- Timer: Providing a timer is not necessary to execute concurrently several OSes. However, to simplify the porting of an OS, XtratuM provides *at least* one virtual timer. The exact number of timers implemented by XtratuM depends on the available number

of hardware timers. For example, when XtratuM is executed in the intel x86 architecture (supposing the APIC timer available) will offer two different virtual timers: the classic PIT and the APIC timer. Besides, to work with these virtual timers, XtratuM also provides a high-level API to deal with them.

- Virtual Memory: Currently, XtratuM is only able to create a memory map per OS, enabling memory isolation among different OSes. This facility is still under development and lacks of many features like a sharing memory mechanism or increasing/shrinking the initial allocated memory.

Therefore, currently, from the guest OSes point of view, XtratuM just provides a high-level API to handle a timer and the interrupts. Each guest OS has to be aware about the rest of the existing hardware and how to share it. For instance, two different guest OSes which are going to use the serial port at the same time have to cooperate between them to avoid a race condition on it use.

New features like an inter-OS communication mechanism and a sharing resource protocol is being implemented and will be released soon.

3.1 XtratuM's scheduling issues

In its first releases, XtratuM implements an scheduling policy based on fixed priorities, where each domain has to indicate its priority at the creation moment. The reason of this decision is because, initially, XtratuM was though to execute a general purpose OS jointly with a hard RTOS in the same computer. In this conditions, triggered interrupts will be reissued to the domain depending on the domain's priority. The main benefits achieve because of the use of this policy is the low overhead and its implementation simplicity.

Nonetheless, the existence of more than one domain with timing requirements would make more desirable the use of a different scheduling policy. At this moment we are studying the possibility of introducing policies which guarantee the use of the CPU to each existing domain with timing constraints.

4 Implementation details

Implementing a nanokernel from the scratch is an arduous, hard task with a great amount of work to be carried out: programming a booting code for the targeted architecture, implementing new drivers, etc.

Nonetheless, as demonstrated in the Adeos nanokernel paper[10], all this work can be greatly simplified.

The nanokernel can be designed/implemented avoiding to start from the scratch but from a previously existing kernel (considered in Adeos as the root domain). The nanokernel is built around the infrastructure of this existing OS (the root domain). Using this approach, Adeos avoids the management of the virtual memory or the great majority of the existing devices, it just take care of interrupts of the system and supplies an scheduler to schedule the domains (guest OSes). Even the loading of the domains, as well as Adeos itself is implemented via the modules of the Linux kernel, therefore overriding the necessity of implementing a loader.

Taken advantage of this approach, the first versions of XtratuM have been built using the infrastructure supplied by the Linux kernel. Basically, these first versions consist of:

1. A patch for the Linux kernel. This patch modifies the Linux kernel in two ways: replacing all the disabling/enabling interrupt instructions by calls to XtratuM and inserting several hooks in the Linux kernel code. These hooks will be used later to virtualise the interrupts and the hardware timers.

2. The XtratuM nanokernel itself. Provided as a piece of software which has to be inserted inside Linux through the Linux kernel module mechanism. A XtratuM boot loader has been avoided using the Adeos approach, that is, XtratuM is loaded into the system as a Linux kernel module (sharing the memory map with Linux). However, XtratuM differs on the method used to load the guest OS. In Adeos the domains are also loaded as Linux kernel modules. XtratuM uses its own loader to create a specific memory map for each guest OS, enabling memory protection between the different OSes.

Besides, XtratuM virtualises the interrupts in a simi-

lar way as Adeos does. The IDT[2] entries, which contain the Linux's interrupt handlers, are replaced with the XtratuM's interrupt handler addresses. Once the IDT has been modified, interrupts are completely managed by XtratuM.

5 Examples of use

Next are some examples of how XtratuM can be used:

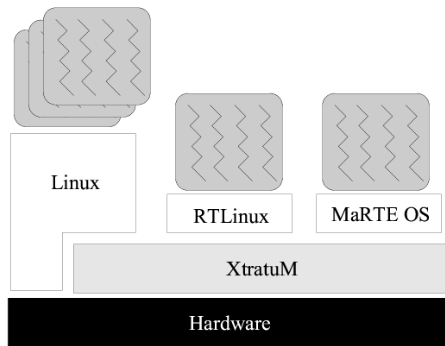


Figure 3: XtratuM running redundant systems.

Figure 3 represents a system were Linux is the background operating system; RTLinux/GPL playing the role of master real-time operating system and running the controlling application; and MaRTE OS also running the same application (but coded by a different developers group and using a different programming language Ada) but the application does not effectively send the actions to the hardware but compares its our results with those generated by the RTLinux/GPL domain. In case of a mismatch in the actions computed by both applications, or if a domain raises an exception, XtratuM can stop the buggy domain. Note that in order to know which is the faulting domain it might be possible to need a third domain.

XtratuM, when compiled with booting code (which will be developed soon) can be used to run the real-time operating system in several hardware processors with minor code changes. We are currently working on the ARM (Xscale) porting of XtratuM.

[2]Interrupt descriptors table, in the x86 architecture is the place where the address of the interrupt handlers are stored.

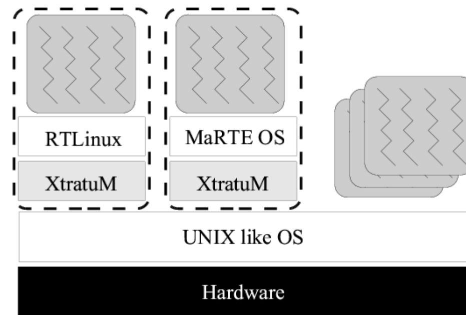


Figure 4: Using XtratuM to test the operating system, or application.

Another practical use of XtratuM framework is to run your RTOS on the top of a Linux system as a regular Linux process. The idea is to compile the guest operating system as a normal ELF executable and then run it the same way as ls or bash does. In this scenario, XtratuM used the POSIX signals and timers facilities provided by the host operating system as if they were interrupts and timers devices.

It is a very restricted and unrealistic system that can only be used for testing and to speed up the code development. It can also be used for teaching purposes.

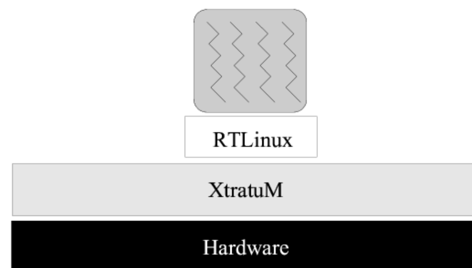


Figure 5: XtratuM as an Stand-alone system.

6 Conclusions

This paper presents the basic architecture of a new nanokernel called XtratuM which allows to execute several applications (application or operating system plus applica-

tions) with temporal and spatial isolation.

Currently, a prototype is ready where the Linux kernel is a domain executed in the XtratuM's memory space.

Although XtratuM was initially designed to permit the execution of a real-time operating system (as RTLinux/GPL) jointly with a general-purpose Operating System (Linux), currently, other configurations have been explored:

- Use of MaRTE OS replacing RTLinux/GPL as the real-time operating system.
- Execution of several real-time operating systems jointly with Linux. The approach requires a modification in the current XtratuM scheduling policy.
- Directly execution of an application without any kernel beneath it. For instance, a real-time application using a cyclic executive.

As future work, we are considering and working on three improvements:

- Adding new scheduling policies to XtratuM. These new policies will support more than only one application with timing requirements.
- Building an stand-alone version of XtratuM. Linux has to be executed as any other partition, with its own memory address. In the current version of XtratuM a fault in the Linux kernel is translated to a crash of the whole system.
- Replacing current non-standard XtratuM's API with a standard specification as ARINC 653-1.

References

[1] L4 microkernels specification. http://l4ka.org/projects/pistachio/l4-x2-r2.pdf.

[2] Mit exokernel operating system. http://www.pdos.csail.mit.edu/exo.html.

[3] The new plex86 x86 virtual machine project. http://plex86.sourceforge.net/.

[4] Herman Hartig et al. Fiasco microkernel. http://os.inf.tu-dresden.de/fiasco/overview.html.

[5] NeTraverse. Win4lin. https://www.netraverse.com.

[6] A. Silberschatz and P. B. Galvin. *Operating Systems*. Addison Wesley Longman, 1999.

[7] A. S. Tanenbaum. *Modern Operating Systems. Second Edition*. Prentice Hall, 2001.

[8] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems. Design and Implementation. Second Edition*. Prentice Hall, 2000.

[9] Inc. VMWare. Vmware workstation. http://www.vmware.com/.

[10] Karim Yaghmour. Adaptive domain environment for operating systems. http://www.opersys.com/ftp/pub/Adeos/adeos.pdf.

[11] Victor Yodaiken. The rtlinux manifesto.

Kernel Support for Energy Management in Wireless Mobile Ad-Hoc Networks *

Mauro Marinoni, Giorgio Buttazzo, Tullio Facchinetti, Gianluca Franchino
University of Pavia, Italy
{mauro.marinoni, giorgio.buttazzo, tullio.facchinetti, gianluca.franchino}@unipv.it

Abstract

Effective power management in wireless networks of mobile robots requires a proper support from the operating system, which must allow the application to dynamically configure the onboard resources to save energy consumption while guaranteeing the required real-time and performance constraints. In this paper, we present the kernel mechanisms necessary to achieve an integrated power management approach, in which energy saving is achieved at different levels of the architecture, including the processor, the communication device, and the robot peripherals, like sensors and actuators.

1. Introduction

The use of coordinated teams of small robots has several interesting applications, including monitoring, surveillance, searching, and rescuing. On the other hand, the use of small robot systems introduces several new problems that need to be solved for fully exploiting the potential benefits coming from a collaborative work. Most of the problems are due to the limited resources typically available on a small mobile robot. In fact, cost, space, weight, and energy constraints, impose the adoption of small microprocessors with limited memory and computational power. In particular, the computer architecture should be small enough to fit on the robot structure, but powerful enough to execute all the robot computational activities needed for achieving the desired level of autonomy. Moreover, since such systems are operated by batteries, they have to limit energy consumption as much as possible to prolong their lifetime.

In a wireless ad hoc network of mobile robots, energy can be saved at different architecture levels. At the operating system level, suitable scheduling and resource management algorithms can be adopted to execute tasks at the minimum speed that guarantees the required performance constraints. At the network level, the transmission power of each node can be set at the minimum level that guarantees a given degree of connectivity. At the application level, specific devices can be turned off, or configured at a proper operating low-power mode (if any), when they are not used for a sufficiently long interval of time. Also servomotors can be driven to drain less current when the robot joints are set in a configuration that does not demand high torques.

Models to describe the battery charge behavior have been proposed to help in deriving new approaches to the battery usage. Benini et al. [5] proposed a flexible discrete-time battery model that accurately describes the dynamic battery operation, allowing a careful system design accounting for realistic battery lifetime values. In a different work [4], the authors achieved a significant battery lifetime improvement by steering the power absorption in a multi-battery pack. This solution may be easily adopted when servomotors are used as actuating devices.

In the context of real-time systems, different energy-aware algorithms have been proposed to minimize energy consumption in the processor. They basically exploit voltage variable processors to minimize the speed while guaranteeing real-time constraints [15, 2, 3, 9].

In many cases, however, the approaches proposed in the literature are based on simplifying assumptions, like negligible overhead or continuous dynamic voltage scaling, which make them unusable in real applications, especially in those embedded systems based on small microcontrollers with very limited operating modes. In some cases, only two modes are available, so power management can only be achieved by switching between the two modes using suitable strategies. Recently, Bini et al. [6] proposed a method for analyzing the feasibility of real-time applications that execute by alternating two speeds, taking overheads into account.

On the network side, energy-aware algorithms have been mainly focused on the MAC level. Some others considered energy conservation in routing problems. Ye et al. [16] proposed the Sensor-MAC (SMAC) protocol, which is divided in two phases: a sleep period and an active period. In the sleep period the nodes switch their transceiver off, by putting it in sleep mode. In the active period, the nodes turn

* This work has been partially supported by the Italian Ministry of University Research under contracts 2003094275 (COFIN03) and 2004095094 (COFIN04).

its transceiver in the receiving mode to listen for incoming communications, or in transmission mode to initiate a communication. Each node can choose its sleep/active schedule, therefore sleep and active periods have to be locally synchronized between nodes. To synchronize them, nodes exchange $SYNCH$ messages, which contain the identi cation number of the sender and the time of its next sleep. The protocol is carrier sense multiple access with collision detection (CSMA/CD), so the synchronization does not have to be very strict. The active period is divided in two parts: the rst part is used by nodes to send their $SYNCH$ messages, if any, while the second part is used for the request to send messages (RTS), if any.

T-MAC [13], like SMAC, adopts synchronized sleep/wakeup cycles to allow nodes to operate at low duty cycles while maintaining network connectivity. In order to reduce latency, T-MAC proposes a $future - request - to - send$ (FRTS) scheme to inform a node, on the third hop, that there exist a message for it by sending a FRST packet. Hoesel and Havinga [14] proposed a MAC protocol, LMAC, based on a TDMA scheme. Time is divided into slots, whose size is suf - cient to send entire messages. Each node can have only a time slot, during which the communication is collision-free. This implies that energy is not wasted for managing collisions and accessing the radio channel. The scheduling algorithm is distributed and each message is dived in two parts: a control unit and a payload unit. The control unit includes several data, such as node identi er , data size, and a sequence slot number to maintain synchronization between nodes. To save energy, each node that is not addressed for communication turns its radio off until the next slot. Moreover, two nodes switch their transceiver off when the communication between them n- ishes.

Yu et al. [17] proposed the Geographic and Energy Aware Routing (GEAR) algorithm, which considers energy ef cienc y. Based on the fact that in the sensor networks a query is often geographical, GEAR propagates a query to the appropriate geographical region using energy-aware and geographically informed neighbor selection heuristics to route a packet towards the target region. Within a region, it uses a recursive geographic forwarding technique to disseminate the packet.

What is missing in the literature, however, is an integrated framework for energy-aware control, where different strategies can be applied at different levels of the architecture, from the hardware devices to the operating system, up to the application level.

In this paper, we present a system wide approach to energy management applied to all the architecture levels and integrated with the scheduling algorithm to guarantee real-time constraints. The method is tailored for an embedded

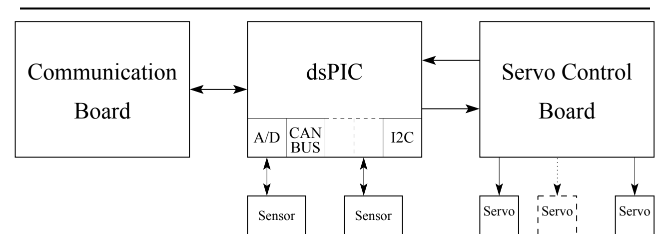


Figure 1. Block diagram of the main robot components.

robot controller consisting of a dsPIC 30F601x family microcontroller, capable of driving more than 20 servomotors, and a wireless communication board with different power/transmission modes.

The rest of the paper is organized as follows. Section 2 presents an overview of the system architecture, describing the degrees of freedom available in each component to achieve some form of power management. Section 3 illustrates the methods we propose at different architecture levels to limit energy consumption while still meeting real-time constraints. Section 4 focuses on the kernel support required to provide e xible power management services to real-time applications. Finally, Section 5 states our conclusions and future work.

2. System description

The system under consideration consists of a team of mobile robots that have to cooperate for achieving a common goal. Each robot can be either a classical wheeled vehicle or a legged walking machine actuated by servomotors, equipped with proximity and special-purpose sensors, a processing board, and a wireless communication subsystem. Hence, each robot can be seen as a mobile node of a wireless ad hoc sensor network. A block diagram of the main robot components is illustrated in Figure 1.

In the following sections we describe the characteristics of each component installed on each robot unit, focusing on the features that may enable the implementation of energy-aware control strategies.

2.1. Onboard microcontroller

The onboard processing unit is a Microchip dsPIC [10], which seamlessly integrates the control attributes of a microcontroller (MCU) with the computation and throughput capabilities of a Digital Signal Processor (DSP). It is a 16-bit microcontroller where most of the 24-bit wide instructions are executed in 1 cycle up to 30 MIPS. The model selected for prototyping includes a program memory space of 144 KBytes, a data memory space of 8 KBytes,

and a non-volatile data EEPROM of 4 KBytes. The MCU presents a full-features software stack, up to 41 interrupt sources, and 5 16-bit counters with 32-bit working mode. The DSP engine features a high speed 17-bit by 17-bit multiplier, a 40-bit ALU, two 40-bit saturating accumulators and a 40-bit bidirectional barrel shifter, and performs divisions in a 19-cycles loop. In terms of peripherals the chip supplies Capure/compare/PWM functionality, 12-bits Analog-to-Digital Converters (A/D) with 100 Ksps conversion rate and up to 16 input channels. Connectivity is provided through a full range of channels: I2C, SPI, CANbus, USART and Data Converter Interface (DCI), which supports common audio Codec protocols, as I2S and AC'97.

The microcontroller allows the application to choose among three different clock sources: an external oscillator up to 40 MHz with an internal PLL circuit to boost the frequency up to 120Mhz, an internal clock of 8 MHz, and a low power clock of 512 KHz. A postscaler can be applied to the selected source to slow down the frequency of a factor of 4, 16, or 64 to obtain the system clock. Once the clock frequency is selected, it is possible to set the supply voltage at the lowest level that supports such amount of MIPS. For example, slowing down the clock to one-third of its maximum value the power supply could be lowered to 2.5 Volts. It is also possible to create a set of frequency/voltage pairs to be used as power states in dynamic voltage scaling (DVS) algorithms. Changing the clock source is an action that is performed with a latency of 10 periods of the new clock.

The MCU has two reduced power modes, idle and sleep, which can be entered through the execution of a speci c instruction. In the idle mode the CPU is disabled, but the system clock source continues to operate. Peripherals continue to operate, but can optionally be disabled. In the sleep mode, the CPU, the system clock source, and any peripherals that operate on the system clock source are disabled. This mode consumes less power, but requires a delay from 10 μs to 130 μs when exited, whereas the idle mode has no wake up delay.

A method is provided to disable a peripheral module by stopping all clock sources supplied to that module. When a peripheral is disabled with this feature, it is in a minimum power consumption state. When the command is sent, the interested module is disabled after a delay of 1 instruction cycle. Similarly, when the wake up command is given, the target module is enabled after a delay of 1 cycle.

2.2. Communication board

In the market there are many transceivers that are suitable to build small radio devices that can be used to realize sensor nodes. There are many smart features that can be exploited to design energy-aware transmission protocols. Some of them are listed below:

- RSSI (Receiving Signal Strength Indicator) is a value proportional to the strength of the received RF signal. It can give a greedy esteem of the distance from the source, if the transmission power is known.
- Different levels of transmission power. They can be exploited, in conjunction with the RSSI, to save energy, adapting the transmission power to the distance between source and sink nodes.
- Different operating modes. The most common modes are: $sleep$, $receiving$, and $transmission$. Each mode consumes different level of energy. When a transceiver is on sleep it consumes less power than in others modes, but the time to switch between modes is different. For example, switching from $sleep$ to $transmission$ takes more time than switching from $receiving$ to $transmission$. Moreover, switching between modes consumes energy too. This latter consideration is important in the communication protocols design.

The characteristics described above can be found in several devices available on the market. A couple of devices suitable for our class of embedded system are the CC1000 and the ATR86RF211.

The CC1000 is a chip produced by Chipcom, with a transmission rate of 78,5 Kbaud/sec, a variable transmission power from -20 to 10 dBm, a minimum supply voltage of 2.1V, and a RSSI output pin for signal strength acquisition. It has two operating modes: power-up and power-down. In the power-down mode, it consumes no more than 1 μA. The transceiver can be set in the ISM (Industrial, Scienti c and Medical) and SRD (Short Range Device) frequency bands at 315, 433, 868 and 915 MHz, but can easily be programmed by a microcontroller to operate at other frequencies in the 300-1000 MHz range.

The ATR86RF211, produced by Atmel, operates in the ISM band (from 400 MHz to 930 MHz), with a FSK (Frequency Shift Keying) modulation, a data rate of 64 kbps, and eight digitally selectable power levels. The maximum transmitter power is 14 dBm in the 433 MHz frequency band. Its power saving features are: *power down* mode, *sleep* mode, and stand-alone *wake up* procedure. It consumes no more than 0.5 μA in power down mode and no more than 3 μA in sleep mode. It is a multi-channel transceiver with fast frequency shifts (less than 50 μs for a 100 KHz shift). This feature is suitable to implement an ef cient *frequency hopping* transmission protocol. The AT86RF211 is also well adapted to battery operated systems as it can be powered with only 2.4V. It can be controlled by means of a three wire interface, either by a microcontroller or by a DSP. Finally it has an RSSI output pin in order to acquire the strength of the received signal.

2.3. Servomotors

The motors considered in this work are the Hitec HS-475HB, which include an internal position control loop that allows the user to specify angular positions through a PWM input signal. This feature simpli es the external circuitry and avoids sending feedback signals to the motor control unit. The internal feedback loop imposes an angular velocity of 250 degrees per second and the motor is able to generate a maximum torque of 9.6 kg·cm with a voltage of 6 Volts. Motors are connected to a board that provides them with the required power supply and the input signals coming from the control layer. The torque applied to the motors can be estimated by monitoring the current drained by the servo. Such a current is read by using a Maxim MAX471 chip, which produces an output voltage proportional to its input current.

Monitoring the servo current absorption can also be exploited to obtain a level of feedback in servomotors control. In fact, as described in Section 3.3, the current absorption is related to the exerted torque and can be used to detect and compensate angular position errors.

3. Power management

Hardware and software components cooperate to reach the following main goals: low power consumption, onboard sensory processing, real-time computation, and communication capabilities. The robot is built with generic mechanical and electrical components, making the low-power objective more dif cult to be satis ed. Nevertheless, the adoption of power-aware strategies inside the software modules signi cantly increased the system lifetime. To achieve signi cant energy saving, power management is adopted inside every system module and needs to be coordinated at the operating system level.

3.1. DVS management

Using the DVS capabilities described in Section 2.1, it is possible to implement a power-aware scheduling algorithm for a discrete set of clock frequencies. The possibility to put the processor in a sleep state can be useful for various purposes. The simplest strategy is to put the processor in this state if there is no work to be executed. The sleep mode can be exited upon the arrival of an interrupt from a peripheral or from the system timer for a task activation. The sleep state can also be considered as an actual operating mode, and the corresponding zero speed can be included in the set of available speed levels.

The simplest approach for integrating power-aware scheduling and real-time constraints is to x the clock speed to a value computed off line to guarantee system feasibility. Due to the limited number of allowed frequencies for the system clock, this solution may cause a waste of power consumption. A better result can be achieved by alternating two clock frequencies to reach the ideal value requested by the theoretical calculation. In fact, alternating two clock states, as a PWM signal, one can approximate a speed level that is not available in the processor. In this way, an optimal speed level can be computed to guarantee real-time constraints while minimizing energy consumption, as proposed by Bini et al. [6]. Another approach is to calculate a different speed for each task and set the system clock to the appropriate value when a context switch occurs. In addition to the off-line calculation of the working frequency, it is possible to implement an on-line reclamation technique to further decrease the frequency using the unused computation time.

Another possibility for reducing energy consumption at the system level is to use the capability of the MCU to put each single device in different working modes: each one with well-known energy requirements. It is also possible to put a device in a non working state and decide whether the peripheral could work while the CPU is in the idle state.

3.2. Communication board

In a mobile node, a part from the electromechanical components, the radio module is the component that usually consumes most energy. When designing a communication protocol for such networks, one must consider the main sources of energy waste. In the following, we brie y describe some of them.

The rst source is *idle listening*, which is due to the energy wasted when listening to the channel to receive possible messages. The second source is *collisions*: when two or more nodes try to send a message at the same time, some collisions are experienced and the corrupted packets have to be retransmitted, causing more energy consumption. This is particularly true in carrier sense multiple access (CSMA) systems. The third source is *overhearing*, occurring when a node picks up packets that are not sent to it. Another source is due to *protocol overhead*: simple protocols need less energy to operate. Some protocols introduce additional control packets (e.g., RTS/CTS packets in the IEEE 802.11 [1]) to solve the hidden node problem [12]. Such added control packets consume additional energy.

The energy wasted during communication can also depend on the particular approach used at the MAC-level. For example, time division multi access (TDMA) protocols are collision-free, therefore they do not have to consume energy to retransmit corrupted packets. However, they are characterized by poor scalability, high protocol overhead, and require to exchange additional information for clocks synchronization.

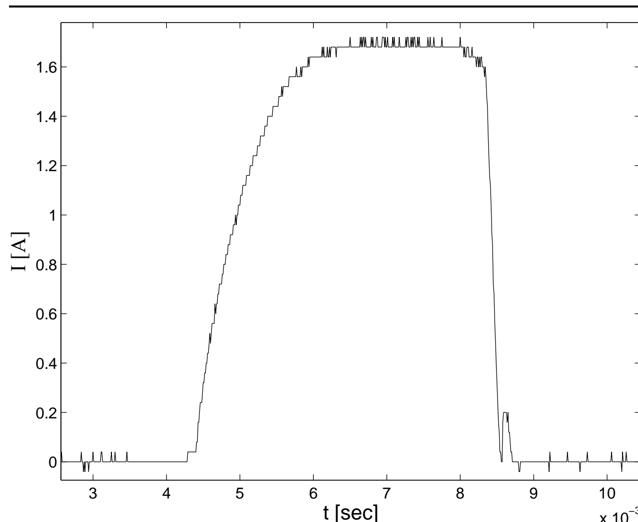


Figure 2. Current absorption per period in a servomotor.

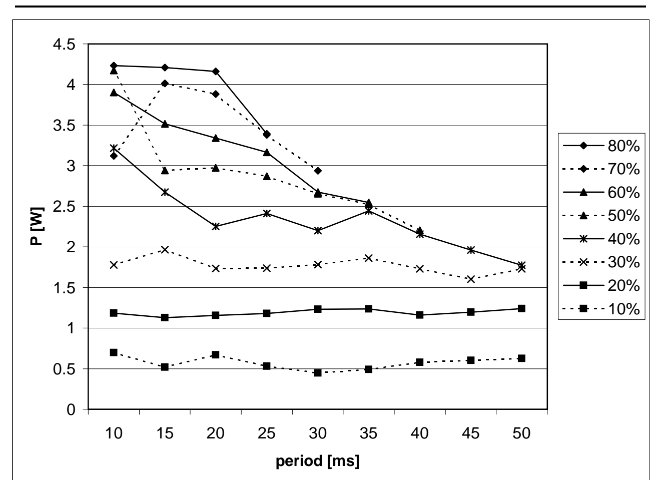


Figure 3. Power absorption per period in a servomotor with different loads percentage of the maximum load.

Transmission power control [8, 11] is useful to guarantee network connectivity, manage density and allow spatial reuse of radio channels. Moreover, minimizing transmission power can also indirectly reduce energy consumption by reducing the channel contention and collision between transmission nodes. Balancing density and connectivity networks maximize spatial reuse of the spectrum. The transmission power control is often encapsulated in the MAC or in the routing protocol.

3.3. Servomotors

In mobile robot systems, the energy consumed by motors is signi cantly higher than the one spent for processing and communication. Hence, a careful management of the motor power can remarkably improve the system lifetime. Servomotors are commonly adopted in robotic applications, since they are cheap and integrate reduction gears and position control to simplify their usage. A servomotor is controlled by modulating the duty cycle of a square wave signal, where the duration of the active pulse de nes the angular position of the shaft. The pulse period does not in uence the shaft angular position, but affects the servo current absorption, since the energy absorption starts at every period and its duration depends both on the load and the period. Figure 2 shows a typical current absorption in a servomotor within a control period.

To assess the behavior of the servo energy consumption, we performed several tests on the Hitec HS-475HB servomotor described in Section 2.3. We carried out several experiments by varying the control period and the load torque. Figure 3 shows the consumed power as a function of the control period for different loads. It is interesting to notice that for loads not exceeding 30% of the maximum value the power consumed by the servo does not depend on the control period, while for higher loads it decreases with the control period.

Figure 4 shows the angular errors between the position set-point and the real servo shaft position as a function of the control period and for different applied torques. For loads higher than 30% of the maximum torque value, the servo is not able to keep the position set point, and the error increases with the control period. However, such an error can be predicted by estimating the exerted torque through the absorbed current and can be corrected by using an external feedback loop.

On the servomotors side, power consumption can be controlled at two different levels. At the application level, the robot system can be driven to reach pre-de ned postures that minimize the torques on the robot joints. This strategy can be quite effective in multi-link robots with several degrees of freedom, such a walking machines and anthropomorphic manipulators. At the signal generation level, a longer control period allows the driving hardware to maintain low-power states for longer time.

4. Kernel Support for Energy Management

This section describes the kernel infrastructure required to support energy-aware strategies at the application level. It consists of two parts: a set of mechanisms inside the kernel that implement the methodologies described in the previous sections and a set of library functions that simplify the user interaction. In the following, the rst part is re-

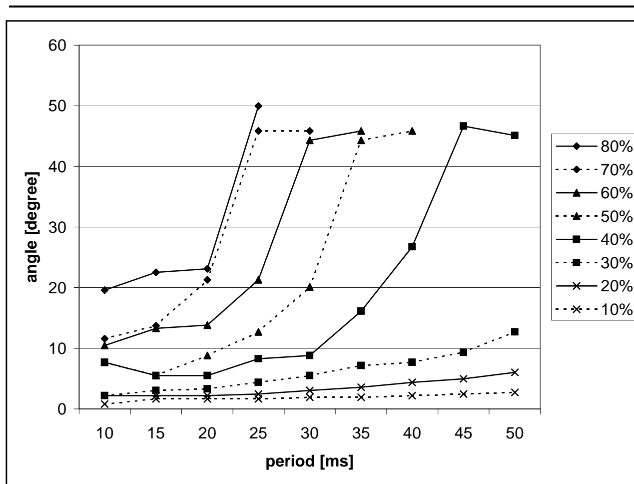


Figure 4. Shaft position errors with different periods and loads percentage of the maximum load.

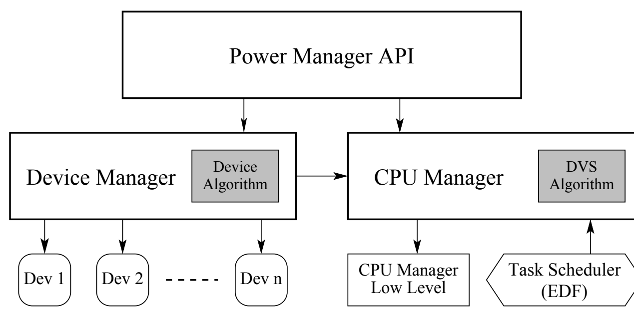


Figure 5. Block diagram of the Power Manager.

ferred to as the *Power Manager*, while the second part is referred to as the *Power Management API*. The main objective of the Power Manager is to achieve high ef cienc y to reduce the overhead introduced in the system, whereas the main goal of the Power Management API is to provide an abstraction layer that allows the user to control the power consumption of the different resources in a simple and uniform fashion. The Power Manager is responsible for selecting the most appropriate power states both for the CPU and the peripheral devices.

A block diagram of the power management architecture is illustrated in Figure 5. The blocks at the bottom represent the elements used for the interaction with the hardware, while the top layer provides the application programming interface (API) for interacting with the user.

- Blocks from *Dev 1* to *Dev n* represent the low-level drivers of each single peripheral that the Device Manager uses to control the power states.
- The CPU Manager at the lower level is in charge of managing the DVS capabilities of the processor working on the frequency-voltage table.
- The task scheduler interacts with the CPU Power Manager through a set of functions invoked at known scheduling events.
- The Device Manager controls the operating modes of the peripherals in order to reduce the power consumption at the system level while guaranteeing consistency of both the operating system and the application. The strategy adopted by the Device Manager is decided by a speci c device algorithm, that can be changed by the user.
- The Device Manager relies on the CPU Manager to integrate the CPU with all others peripherals. It could implement different algorithms to reduce power consumption and work as a bridge between the abstraction used at the higher level and the DVS mechanism at the bottom layer.
- The set of library functions in the API layer allows the application to interact with the power-management infrastructure in a simple and uniform fashion.

4.1. CPU Power Manager

The main goal of the CPU Power Manager is to select the most appropriate voltage level and clock frequency in the processor. The clock management is split into two levels: a lower level related to the hardware and a higher level related to the DVS algorithm. The lower level is in charge of manipulating the fundamental CPU parameters, like clock frequency and voltage level, so it is architecture-dependent, while the higher level decides the time and the value of the CPU parameters, so it depends on the DVS scheduling algorithm.

Since each CPU is characterized by a different set of frequencies and voltage levels, and not all possible combinations are allowed, the set of frequency-voltage pairs supported by the CPU is stored in a table, as shown in Figure 1. Then, a set of kernel primitives allows the DVS algorithm to retrieve some relevant information, such as the minimum voltage compatible with a given frequency, the maximum frequency consistent with a chosen voltage, or the normalized speed ($s = f/f_{max}$) corresponding to the selected mode. Other primitives also allow reading and writing the current frequency and voltage level, and managing the system time by acting on the system tick.

To simplify the implementation of a power management scheme for the CPU, the operating system also supplies a set of hooks for executing speci c DVS functions upon the occurrence of certain events. A list of most relevant events

	f_1	f_2	f_3	f_4
v_1	1			
v_2	2	4	6	
v_3	3	5	7	8

Table 1. Table storing the frequency-voltage pairs allowed by the CPU.

that may require the intervention of the power manager is reported below:

- **System startup** - This hook allows to setup the environment of the DVS algorithm and its initial state. It can also be used to compute the working frequency based on the task set parameters.
- **Context switch** - This hook is important whenever the DVS algorithm has to modify the clock frequency as a function of the scheduled task, or perform some resource reclaiming based on the unused computation time.
- **CPU idle** - When the ready queue becomes empty, the power manager could set the CPU in a low power consumption mode until a task is activated or an interrupt is raised.
- **Wake-up** - When the system exits from a power-saving status, some action could be executed before the kernel restarts.
- **Power-Management Point** - In some cases, hooks may be explicitly inserted in the applications tasks through a proper system call, to invoke speci c DVS functions. For example, some algorithms proposed in the literature [9] require to insert a function in the task body to calculate the actual computation time with respect to the worst-case one.

Finally, in some other cases, power management may need to be executed at given time instants, hence the kernel must provide a mechanism to activate a DVS function by a timer. For example, this functionality is needed by the Speed Modulation algorithm proposed by Bini et al. [6].

4.2. Device Power Manager

Reducing energy consumption at the system level is possible because most peripheral devices support various operating modes and the MCU has the capability of putting each device in a sleep state. A problem is that some peripherals trash the current job when switched in low-consumption states. For example the A/D converter loses the ongoing acquisition and the UART does not listen to incoming data. To reduce the power consumption without affecting the behavior of tasks, a system-level coordination is required.

To support power management of peripheral devices at the kernel level, a `pstate` type is de ned as an array with size equal to the number of devices in the system. Each element of the array stores the power state of the peripheral, where power states are represented by integers sorted by power consumption.

Three `pstate` variables are de ned in the kernel:

- W^{max} stores the maximum number of different power states each device can manage;
- W^{sys} stores the minimum power level required by each device for the correct kernel operation.
- W^{dev} stores the actual power level set by the power manager for each peripheral device.

Then, two arrays of `pstate` type are de ned for each task τ_i to express its requirements:

- R_i^{run} stores the minimum power level for each device required for the correct behavior of task τ_i, while it is running;
- R_i^{idle} stores the power levels requested by τ_i, when it is not running.

It is worth observing that the proposed approach is general enough to include the CPU in the set of devices used by the task. In this case, the mapping between the power state values inside the array and the real processor behavior is performed by a function provided by the Power Manager.

The arrays de ned above can be used in a static or dynamic fashion, depending on the maximum overhead that can be tolerated in the system. If the overhead has to be minimized, the static approach is more suited, where all the values in the arrays are x ed and computed in the worst case. Otherwise, each task can dynamically change these values during execution, so allowing the power manager to reduce the overall power consumption of the system. As an example of dynamic behavior, a task could request a given power level for the A/D converter only while the acquisition is in progress, and then reset the value to zero. In the static approach, the re-computation of all power levels is needed only during a context switch, while in the dynamic mode the new power state for a device has to be computed every time the running task modi es its power state requirements. If τ_i is the new active task and p is a speci c device, the new value $W^{dev}[p]$ is computed as the maximum between $W^{sys}[p]$, $R_i^{run}[p]$ and the maximum value $R_{max}^{idle}(i,p)$ among all values $R_k^{idle}[p]$ for tasks different than τ_i. That is:

$$W^{dev}[p] = \max\{W^{sys}[p], R_a^{run}[p], R_{max}^{idle}(i,p)\}$$

where

$$R_{max}^{idle}(i,p) = \max_k \{R_k^{idle}[p] \mid \tau_k \neq \tau_i\}.$$

4.3. Power Management API

The interaction between the application and the power management infrastructure occurs through a set of functions that manipulate the set of `pstate` arrays. Every function is implemented to work on a single device. The most important functions perform the following tasks:

- Get the maximum power level allowed by the system to the peripheral;
- Get the minimum power level required for the operating system consistency;
- Get the power level the device is actually using;
- Get the power level required by the task while it is running;
- Require a new power level for the device while the task is in execution; the actual power level for that device will be decided by the power manager based on all task requirements.
- Get the power level required by the task while it is not running;
- Require a new power level for the device while the task is not running; the actual power level for that device will be decided by the power manager based on all task requirements.
- Get the power level required by all the other tasks while not running;

There are also ve functions used to interact with the power-management algorithms.

- Two of them are used to get/set parameters of the CPU Manager to tune its behavior.
- Another function is needed to de ne a Power Management Point, that must be explicitly inserted by the user in the task body.
- The last two functions allow the user to get/set parameters of the Device Manager to tune its behavior.

5. Conclusions

In this paper we presented an integrated approach for achieving energy management in wireless ad hoc networks of mobile robots. We showed that signi cant energy saving can only be obtained by a combined effort at different architecture levels. At the operating system level, speci c power-aware algorithms can be adopted to set the appropriate operational mode to minimize energy consumption while guaranteeing the timing constraints. At the network level, node transmission power can be tuned to guarantees a given degree of connectivity and, at the application level, the control strategies can trade performance with energy consumption, so that the robot can switch to a different behavior to prolong its lifetime when the batteries are low, still performing useful tasks.

We showed how the proposed techniques can be supported at the kernel level to implement e xible energy-aware strategies on the robot resources and on the network.

As a future work, we plan to implement the proposed strategies in the Erika kernel [7], that will run on the dsPIC boards embedded in all robot units.

References

[1] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*. IEEE 1999.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejia Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of the Euromicro Conference on Real-Time Systems*, June 2001.

[3] H. Aydin, R. Melhem, D. Mossé, and P. Mejia Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 2001.

[4] L. Benini, D. Bruni, A. Macii, E. Macii, and M. Poncino. Discharge current steering for battery lifetime optimization. *IEEE Transactions on Computers*, 52(8):985–995, August 2003.

[5] L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi. Discrete-time battery models for system-level low-power design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(5):630–640, October 2001.

[6] E. Bini, G. Buttazzo, and G. Lipari. Speed modulation in energy-aware real-time systems. In *IEEE Proceedings of the Euromicro Conference on Real-Time Systems*, July 2005.

[7] Evidence Srl. *ERIKA Enterprise RTOS*. URL: http://www.evidence.eu.com.

[8] J. Heidemann and W. Ye. *Energy Conservation in Sensor Networks at the Link and Network Layers*. Nirupama Bulusu and Sanjay Jha (editors), 2005. Technical Report ISI-TR-2004-599, USC/Information Sciences Institute, 2004.

[9] R. Melhem, N. AbouGhazaleh, H. Aydin, and D. Mossé. *Power Management Points in Power-Aware Real-Time Systems*. R. Graybill and R. Melhem (editors), Plenum/Kluwer Publishers, 2002.

[10] Microchip Technology Inc. *dsPIC30F family reference manual (DS70046C)*, 2004. URL: http://www.microchip.com.

[11] R. Ramanathan and R. Rosales-Hain. Topology control of multihop wireless networks using transmit power adjustment. In *Proc. of the IEEE Infocom*, March 2000.

[12] F. A. Tobagi and L. Kleinrock. Packet switching in radio channels: Part ii - the hidden terminal problem in carrier sense multiple-access modes and the busy-tone solution. *IEEE Transactions on Communication*, 23(12):1417–1433, December 1975.

[13] T. van Dam and K. Langendoen. An adaptive energy-ef cient mac protocol for wireless sensor networks. In *Proc. of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 1993.

[14] L. van Hoesel and P. Havinga. A lightweight medium access protocol (lmac) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In *Proc. of the 1st International Workshop on Networked Sensing Systems*, Tokyo, Japan, June 2004.

[15] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. *IEEE Annual Foundations of Computer Science*, pages 374–382, 1995.

[16] W. Ye, J. Heidemann, and D. Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. *IEEE/ACM Transactions on Networking*, 12(3):493–506, June 2004.

[17] Y. Yu, R. Govindan, and D. Estrin. Geographical and energy aware routing: a recursive data dissemination protocol for wireless sensor networks. Technical Report UCLA/CSD-TR-01-0023, UCLA Computer Science Department, May 2001.

Variable-Rate QoS in the OS Network Subsystem*

Hui Cheng, Xin Liu, and Steve Goddard
Department of Computer Science and Engineering
University of Nebraska — Lincoln
Lincoln, NE 68588-0115
{hcheng, lxin, goddard}@cse.unl.edu

Abstract

Many distributed applications need support for both real-time computation and real-time communications because of their sensitivity to delay and jitter. It therefore requires the operating system to provide soft real-time support in the processor and the network subsystems. In this work, we present a network subsystem which can cooperate with any CPU scheduler to provide network Quality of Service (QoS) to distributed applications. This system, called VRE-NET, allows applications to reserve bandwidth for the protocol processing capacity of the network subsystem. The bandwidth reservation can be adjusted according to different QoS requirements. Since different applications may have different execution patterns, it is infeasible to predict the resource requirements in general. Instead, we provide an interface for users to associate each application with a rate controller that automatically adjusts the bandwidth reservation. We also provide a default rate controller for typical multimedia applications.

1 Introduction

In recent years, many new network applications have emerged with QoS requirements. These applications, such as distributed multimedia applications, need support for both real-time computation and real-time communications because of their sensitivity to delay and jitter. In practice, a distributed multimedia application needs to send and gather real-time data across the network. To achieve an acceptable performance, it should have sufficient CPU time to code/decode frames. It also demands a certain amount of network bandwidth. The demand varies according to dynamic QoS requirements. Finally, the application should execute with a stable rate and with minimal jitter. It therefore is becoming important for the operating system to provide soft real-time support for these applications to acquire various resources with timing constraints.

A conventional operating system usually separates the scheduling of its subsystems. Each subsystem executes at its own rate and is independent of each other. While they are supposed to provide resources to user processes, none of the subsystems, except the processor scheduler, know anything about user processes. Thus it is hard for a conventional operating system to appropriately allocate all resources to user processes. Unfortunately, for a user process, insufficient support of any subsystem could drastically degrade its performance. For example, consider an online video-player and a file transfer application running on a desktop. To guarantee the performance of the video-player, we assign a higher priority to it than the file transfer application. However, since the packets destined to the file transfer application arrive at a much higher rate than the video-player, the network subsystem processes more packets for the file transfer application than for the video-player. It seems as though the network subsystem assigns inverse priorities to the applications. As a result, the video-player performs poorly even though it has sufficient CPU time.

Users usually only have control over processor scheduling. Conventional Unix/Linux systems provide the *nice* interface to adjust a user process's static priority. Many QoS-supported CPU schedulers, such as BEST [4] and RBE [12], were developed to satisfy the processor resource requirement of multimedia applications. Furthermore, some CPU-schedulers, such as VRE [13, 14] and RBED [5], can automatically adjust a user process's CPU bandwidth according to its dynamic QoS requirements. Therefore, it is becoming easier for users to control processor scheduling, and the important problem is to make the scheduling of other subsystems cooperate with the scheduling of the processor. The challenges include: (1) The scheduling unit for processor scheduling is different from other subsystems. For example, the scheduling unit is a packet for network subsystems, while the scheduling unit is a thread for processor schedul-

*Supported, in part, by grants from the National Science Foundation (EHS-0208619, CNS-0409382).

ing. (2) Users seldom know the exact amount of resources that other subsystems are expected to provide. (3) Even with a stable processor bandwidth assignment, the requirements for other resources may vary from time to time. It is impossible for users to make the adjustments all by hand.

In this work, we provide a real-time solution by enabling network subsystem scheduling to match CPU scheduling. For distributed multimedia applications, CPU and network bandwidth are the two most important resources. Moreover, compared with the receipt of inbound packets, processing of outbound packets turns out to be a trivial problem because user processes will not be blocked for sending packets. Therefore, we only consider the receiving part of the network subsystem in this work.

We propose and implement a variable-rate scheduling model for the OS network subsystem (VRE-NET). The VRE-NET system allows tasks to specify bandwidth needs and create virtual network channels, called VRE-NET queues, with the assigned bandwidth for each task. Remaining tasks share a default queue. Packets in VRE-NET queues have higher priority than packets in the default queue. Therefore, the network subsystem processes packets in VRE-NET queues according to bandwidth assignments and processes packets in the default queue only when no packet in VRE-NET queues is eligible to be processed. In case the system is overloaded, *i.e.*, the total amount of bandwidth assignments is beyond which the network can provide, the network subsystem schedules packets in proportion to bandwidth assignments for user processes. If packets of a task arrive at a rate exceeding the bandwidth of the corresponding virtual network channel, they will be dropped without further network processing.

However, the requirement of network bandwidth for a task may vary from time to time and is hard to predict when these changes will occur. Hence we introduce an automatic rate adjustment mechanism in the VRE-NET system. An application is associated with a rate controller, which periodically monitors the actual bandwidth consumed by the user application. This value can be used to estimate the actual network bandwidth requirement of the application. Thus, the rate controller can make the bandwidth adjustments according to this value. Here we make an assumption that an application will suspend itself when it acquires excess resources. This assumption is practical for most applications. For example, if we set *MPlayer*, a popular multimedia player on Linux, to play a movie at 30 frames per second, then it will only execute with this rate, even when it controls the whole processor and network bandwidth.

Note that when we talk about the bandwidth reservation in this paper, we mean the reservation of the processing capacity of the network subsystem. That is, the VRE-NET system can process incoming packets at a rate according to the bandwidth assignment. It is not the bandwidth reservation over the actual network. However, for some network protocols with flow control mechanisms, such as TCP, the bandwidth reservation of the VRE-NET system can influence the network bandwidth from the sender to the receiver, as we will see in Section 4.3.

Many QoS-supported network subsystem models have been developed to support QoS-sensitive applications. The VRE-NET model offers several advantages over these models: VRE-NET is easily portable to systems that do not support kernel threads; provides an interface for users to associate applications with user-customized rate controllers that automatically adjust the bandwidth reservation; and can easily cooperate with any CPU scheduler.

A full discussion of related work is given in Section 2. The architecture and implementation of the VRE-NET system is presented in Section 3. Section 4 describes how we evaluated our system and presents the results. Section 5 presents our conclusions and describes future work.

2 Related Work

In [7], Druschel and Banga presented the LRP network subsystem for addressing the issues of scheduling anomalies in a monolithic Unix operating system. The network interface demultiplexes incoming packets according to their destination socket, and places the packet directly on the appropriate receive queue. Receiver protocol processing is performed at the priority of the receiving process. For UDP packets, the protocol processing does not occur until the application requests the packet in a receive system call. For TCP packets, a kernel thread is created for each receiving process for the protocol processing. This thread is scheduled at its process's priority and its CPU usage is charged to its process. Later, the LRP model is used in resource containers [2] and QLinux [19], where processes can be grouped together. In these cases, the LRP charges resource usage to corresponding resource containers or classes.

A few limitations of LRP has been identified in [6] and [3]. The limitations include: (1) It is hard to port the LRP to operating systems that do not support kernel threads; (2) The interdependencies between the resource container and the scheduler is very high. Although LRP and resource containers could work well with time-sharing schedulers, it is not clear how the respective techniques can be used in conjunction with real-time or proportional-share schedulers. (3) LRP requires significant changes to the OS kernel.

In [8], Ghosh and Rajkumar proposed the NetR real-time network subsystem based on Linux/RK [16, 17]. The NetR subsystem introduces a "receiving network reserve" mechanism, which lets the application control how many packets should be processed in a period. It also introduced a new kernel thread that replaces the network bottom half and is thereby dedicated to executing protocol processing of arriv-

ing packets. The kernel thread handles packets of different reservations using *Deadline-Monotonic* priorities, which assigns higher priorities to reservations with shorter relative deadlines.

However, NetR may also present significant difficulties. First, it relies on a *resource kernel* [16, 17], which is not available in most operating systems. It also requires the support of kernel threads, as LRP does. Therefore, it has the same portability problem as LRP. Second, it may cause scheduling anomalies for UDP applications. Consider an application that uses non-blocking socket I/O to receive UDP packets. In this case, packets cannot be processed at a rate at which applications are prepared to receive them, no matter how priorities are assigned.

From our perspective, the most relevant related work is the work done within the context of the proportional share scheduling of operating system services by Jeffay et al.[11]. While previous operating system work (*e.g.*, [18, 10, 15]) in proportional share resource allocation considered only the problem of scheduling user processes, [11] addressed the issue of real-time scheduling of internal operating system activities, especially the network protocol processing. The developers were also concerned with the impact of network processing and explicitly scheduled the network processing activity in proportion to the rate at which the process is expected to receive packets. The protocol processing makes progress at the sum of the rates of all processes that are currently receiving packets from the network. Furthermore, the protocol processing activity internally sub-allocates its quantum to packet processing by assigning eligible times and deadlines to packets based on the weights of the user process that will receive the packet.

Several limitations also exist in the work by Jeffay et al. First, kernel activities may be delayed to ensure that user processes execute for a full quantum, which may cause overflow of the input queue. Second, in this model the interrupt routine consists of a loop that removes packets until the input queue is empty or until a maximum number of packets has been processed. However, the “maximum number” is hard to caliberate. The authors determined it by hand timing the loops. Obviously, this value varies for different operating systems or protocols.

Our work differs from these works in several respects.

1. Unlike the LRP and NetR network subsystem, the VRE-NET subsystem does not use a kernel thread to replace the network bottom half (soft interrupt). Therefore, it is easy to port the VRE-NET subsystem to other operating systems.

2. The VRE-NET system does not rely on a specific scheduler. It is transparent to and could cooperate with any CPU scheduler to provide QoS to applications. The CPU scheduler determines the execution rate of a user process. The rate controller will automatically adjust the network bandwidth of the process to match the rate of packets consumption.

3. The VRE-NET system supports both blocking and non-blocking socket I/O. It intercepts system calls that are used to receive data, *e.g. sys_recvfrom*; records the volume of data intentionally read by the user process and only processes the packets at the rate that the process is prepared to receive.

4. The VRE-NET system allows applications to specify the exact bandwidth requirements rather than using weights or priorities. The bandwidth requirement is more straightforward and is only related to the application’s QoS requirements, while the weight will be affected by other factors, such as other processes and the host computer environment.

Due to the similarity of objectives in [11] and this work, it would be nice to conduct experiments to test our VRE-NET system against the proportional-share scheduling system by Jeffay et al. Unfortunately, we were unable to get the code for the proportional-share scheduling system described in [11], and we were unable to reproduce their results.

3 The VRE Network Subsystem

In conventional Unix/Linux network subsystems, the arrival of a network packet is signaled by an interrupt. The interrupt handler then encapsulates the packet in a buffer and queues the packet in a receive queue. Protocol processing of received packets typically occurs in the context of a software interrupt.

Interrupt-driven network subsystems can provide low overhead and good latency. However, it only provides best-effort scheduling and cannot guarantee applications will receive a large enough share of the network bandwidth to meet deadline or timing constraints. In addition, incoming packets are typically processed in a FIFO manner, which provides no isolation between different processes. These systems are not designed to provide applications with QoS guarantees for access to system resources. This leads to poor performance of real-time/multimedia applications. Similar problems also arise in many non-Unix operating systems.

In the VRE-NET system, the receive queue is replaced with several VRE-NET queues and a default queue according to the configuration of the system. As stated in Section 1, we leave the sending mechanism unmodified because the user process will not be blocked by sending packets. The incoming packets are queued per process. Packets destined to other processes will be put in the default queue. We bind the VRE-NET queue with the process to provide QoS guarantees to multimedia applications. Figure 1 illustrates the architecture of the VRE-NET system.

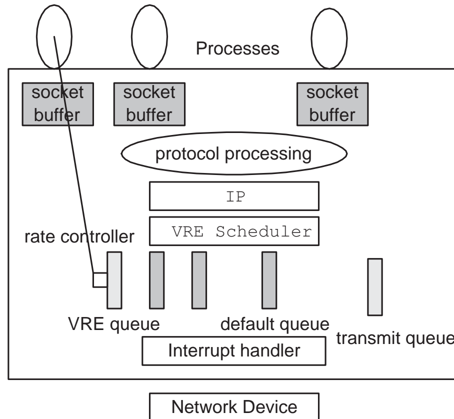


Figure 1. VRE-NET System Architecture

Each VRE-NET queue is assigned a bandwidth and is treated as a real-time task. A real-time task is a task with timing constraints. Each instance of the task is called a job, which should complete before its deadline. Here, the transmission of each packet is treated as a job of the task. Each job is specified by the tuple (v, d) of integer constants. The parameter v denotes the *virtual release time* of the job and the parameter d denotes the *deadline* of the job. The packets with the earliest deadline and with the virtual release time at or before the current system time will be chosen by the VRE network scheduler for protocol processing. The concept of the VRE model and virtual release time is presented in Sections 3.1.1 and 3.1.2.

We also provide a rate adjustment mechanism in our system. In practice, applications can dynamically change their execution rates, and therefore change their resource requirements, including network bandwidth. Our schema is: an application is assigned an initial bandwidth and associated with a rate controller to dynamically adjust the bandwidth. If the process requires more bandwidth than the assigned bandwidth, the rate controller will increase the bandwidth until the user process requires no more bandwidth. On the other hand, if the current bandwidth assignment is more than necessary, the rate controller will re-allocate the task's excess bandwidth to other tasks. The detailed strategy to estimate the bandwidth requirements is presented in Section 3.2.

3.1 VRE-NET Network Subsystem Scheduler

VRE-NET is based on the VRE QoS-Supported CPU scheduling model proposed by Goddard and Liu[9]. The VRE model supports both variable rate tasks and non-real-time tasks. Variable rate tasks reserve a specific execution rate and dynamically adjust the rate. Non-real-time tasks proportionally share the remaining processor capacity as if they were scheduled by a time-sharing system.

Interestingly, the VRE model is well-suited for the scheduling of network processing activity. Real-time tasks reserve a specific bandwidth and associate it with a VRE-NET queue. Non-real-time tasks share the remaining bandwidth and the default input queue. The arrival of packets can be considered releases of real-time jobs. The actual arrival rates of packets is unknown, but the expected arrival rates can be calculated with the expected bandwidth and single packet size. In addition, the expected bandwidth can be adjusted according to dynamic QoS requirements.

3.1.1 Variable-Rate Scheduling Model

A VRE task is described by four parameters $(x_i(t), y_i(t), c_i(t), d_i(t))$, each of which is a variable that changes over time t. $y_i(t)$ is the inteval in which $x_i(t)$ jobs are expected to be released; $c_i(t)$ is the WCET; $d_i(t)$ is the relative deadline, which is typically equal to the period $y_i(t)$. (We assume $d_i(t) = y_i(t)$ in this work.) To effect a rate change, a VRE task can change either its execution time, $c_i(t)$, or its job release rate, $(x_i(t), y_i(t))$. The deadline of a job *j* of VRE task T_i can be represented by Equation (1)

$$D_i(j) = \begin{cases} t_{ij} + d_i(t) & \text{if } 1 \leq j \leq x_i(t) \\ \max(t_{ij} + d_i(t), D_i(j - x_i(t)) + y_i(t)) & \text{if } j > x_i(t) \end{cases} \quad (1)$$

where t_{ij} is the release time of job J_{ij}. All real-time tasks are scheduled using the *earliest deadline first* (EDF) algorithm. The second line of Equation (1) prevents the processor from being saturated by early job releases. A VRE task set is schedulable if there exists a schedule such that each job can complete before its deadline. See [9] for more details. The scheduling of network packet processing using the VRE model is presented in Section 3.1.3.

3.1.2 Virtual Release Time

One problem with the VRE task model is bad response times in some situations, even when all deadlines are met. A real-time task with early job releases and without the competition from other real-time tasks will keep executing and its deadlines will rapidly increase. If another real-time task joins the system, the first real-time task will stop and wait for the new task until the new task's deadline becomes larger than the first one's. The waiting period could be very long for the first task. For users, the first task will appear non-responsive in this period. Note that the VRE schedule is still correct because all tasks meet deadlines.

Although the *waiting period* does not influence the correctness of real-time CPU scheduling, it is not acceptable in network scheduling. An arrival packet is treated as a released real-time job and is the basic scheduling unit for a network scheduler. If a packet has been pending for processing

for a long time, the sender may think that the packet has been lost during the transmission. And worse, for some network protocols, such as TCP, the sender will re-send the packet if it does not receive the acknowledgement in time. This may increase network congestion. In addition, the TCP connection will be disconnected if the sender still does not receive the acknowledgement after re-sending a certain number of packets.

To overcome the problem of bad response times, we introduce the *virtual release time* concept to the VRE-NET system. Each real-time job, *i.e.*, the packet to be processed, is assigned a virtual release time. Only when the system time reaches the virtual release time, is this job eligible to be scheduled. Using Equation (1) to define the absolute deadline, $D_i(j)$, of a real-time job j of task T_i, its virtual release time is given by:

$$v_i(j) = D_i(j) - d_i(t) \quad (2)$$

We claim that the introduction of a *virtual release time* will not harm the schedulability of a VRE system. The proof of this conclusion is beyond the scope of this work but is intuitively obvious since the *virtual release time* simply prevents jobs(packets) from being processed at a rate faster than the current rate expected. Thus the tasks in this system run smoothly and have good response times. For the VRE-NET system, the *virtual release time* enables the VRE-NET scheduler to process packets at a stable rate. This is rather important to network protocols that have their own congestion control mechanisms. For example, a sender and a receiver of a TCP connection typically exchange messages to find an ideal rate to transmit packets. If the network subsystem holds some packets too long, the sender would think that the packets have been lost and would slow down its sending rate. While if the network subsystem quickly processes packets, the sender may speed up. This may introduce more jitter in the network transmission and cause a scheduling anomaly. An example of virtual release time is shown in Figure 2.

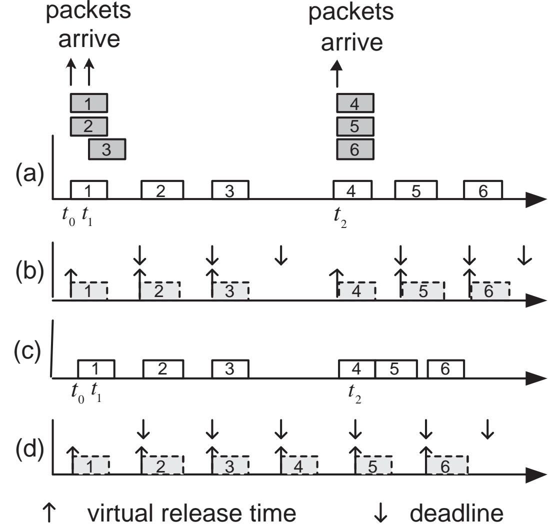


Figure 2. Packets arrive at t_0, t_1 and t_2. (a) shows how the VRE-NET system processes the packets with $q_i(t) = 0$ and (b) shows the virtual release time and deadline for each packet in (a). (c) shows how the VRE-NET system processes the packets with $q_i(t) > 0$ and (d) shows the virtual release time and deadline for each packet in (c)

3.1.3 Packet Scheduling

The VRE-NET scheduler lays between the network device interrupt handler and the software interrupt handler, which does the IP processing for each packet. The task of the scheduler is to choose a packet to be processed. The VRE-NET system records the arrival time for each packet, and calculates the virtual release time and deadline of the packet. It always chooses the packet with the earliest deadline and a virtual release time at or before the current system time.

A VRE-NET queue can be considered a VRE task described as $(1, p_i(t), p_i(t), b_i(t))$, which means 1 packet of the queue will be processed every period of length $p_i(t)$, and the bandwidth of the queue is $b_i(t)$. If the actual arrival time of a packet is $a_i(t)$, the size of the packet is $s_i(t)$, and the deadline of the previous packet of the same process is $D_i'(t)$, then the relative deadline $p_i(t)$ of this packet is given by:

$$p_i(t) = s_i(t)/b_i(t)$$

The deadline D of the packet is given by:

$$D_i(t) = \begin{cases} a_i(t) + p_i(t) & \text{if } a_i(t) \geq D_i'(t) + q_i(t) \\ D_i'(t) + p_i(t) & \text{if } a_i(t) < D_i'(t) + q_i(t) \end{cases}$$

where $q_i(t)$ is a time interval to deal with network jitter. In practice, packets may arrive at a VRE-NET queue in bursts separated by short idle periods. In this case, the reserved bandwidth may not be fully utilized as shown in Figure 2(a). Thus, the $q_i(t)$ parameter is used to reserve an amount of previously unused bandwidth for future packets. It makes the network bandwidth reservation more flexible and accurate, as shown in Figure 2(c). The value of $q_i(t)$ is a tradeoff. While a small value cannot overcome the problem caused by network jitter, a large value may mistakenly reserve too much previously unused bandwidth and pose high workloads to the network subsystem. In our implementation, we let $q_i(t) = 10 \times p_i(t)$ and get good results, as shown in Section 4. The virtual release time $v_i(t)$ of the packet is given by:

$$v_i(t) = D_i(t) - p_i(t)$$

A system interface routine *set_vrenet_params* is used to assign and adjust a real-time task's bandwidth reservation. The following system call assigns the bandwidth of process *pid* to 4,000 kbps.

set_vrenet_params(pid,4000)

The VRE-NET scheduler is triggered in two cases: (1) when the network device interrupt handler enqueues a packet to a queue, the VRE-NET scheduler will be invoked when there is no other task of higher priority; (2) if all of the first packets of queues are not eligible to be processed, *i.e.*, their virtual release time is larger than the current system time T, then the VRE-NET scheduler sets a timer with a expiration period of $min(v_i(t) - T)$ and re-schedules at that time, where $v_i(t)$ is the virtual release time of the first packet of a VRE-NET queue.

3.2 VRE-NET Rate Controller

An application may dynamically change its requirement for network bandwidth from time to time. It follows that the reservation of bandwidth should be changed accordingly. However, we cannot expect users to make the adjustments all by hand since the adjustments might be too frequent and the rules might be too complicated. Thus, we introduce the *rate controller* to automatically accomplish the bandwidth adjustments.

In practice, different applications can have different execution patterns. It is infeasible to construct a single general-purpose rate controller. Thus, we provide an interface for users to implement customized rate controllers. Each application can be assigned a specified rate controller. Meanwhile, we provide a default VRE-NET rate controller. As previously stated, we make the assumption that an application will suspend itself when it acquires excess resources. The assumption is also valid with QoS-supported CPU schedulers, which usually specify an execution rate or reserve a processor bandwidth for an application. The application cannot consume excess resources with the limitation of the processor bandwidth or the specified execution rate.

Let b_{rq} denote the bandwidth requirement of the user process. The rate controller should keep the bandwidth reservation b_{rv} close to the bandwidth requirement b_{rq}, *i.e.*, $|b_{rv} - b_{rq}| \leq \varepsilon$. Our approach to bandwidth reservation is not based on traditional control theory. The primary reason for this is that the target set point b_{rq} is not known in advance.

Instead, the rate controller works by periodically comparing the current bandwidth reservation b_{rv} with the actual bandwidth consumption b_c of the user process. We modify the system calls that are used to receive packets to calculate the volume of data consumed by the process. Theoretically, the bandwidth consumed can exceed neither the bandwidth reserved nor the bandwidth required, *i.e.*, $b_c \leq b_{rv}$ and $b_c \leq b_{rq}$. The rate controller estimates b_{rq} with the following observation: given a sufficient bandwidth reservation, the user process can consume at most the bandwidth it requires; given an insufficient bandwidth reservation, the user process consumes all the bandwidth reserved. That is, if $b_{rv} > b_c$ then $b_c = b_{rq}$. Therefore, the rate controller works by keeping $b_{rv} > b_c$ and $b_{rv} - b_c \leq \varepsilon$. There are three possible cases at runtime,

1. The bandwidth consumed is equal to the bandwidth reserved, *i.e.*, $b_c = b_{rv}$. In this case, the rate controller cannot estimate b_{rq}. Thus the rate controller needs to increase the bandwidth reservation.

2. The bandwidth consumed is a little lower than the bandwidth reserved, *i.e.*, $b_{rv} - \varepsilon \leq b_c < b_{rv}$. In this case, $b_{rq} = b_c$ and $b_{rv} - b_{rq} \leq \varepsilon$. This state is considered a *stable state* for the rate controller.

3. The bandwidth consumed is much lower than the bandwidth reserved, *i.e.*, $b_c < b_{rv} - \varepsilon$. In this case, the rate controller reserved too much bandwidth for the process. The rate controller will decrease the bandwidth reservation to a level that is a little higher than the current bandwidth consumption. In other words, the rate controller will try to achieve the *stable state* by adjusting the bandwidth assignment.

4 Evaluation

In this section, we present experiments designed to evaluate the effectiveness of the VRE-NET network subsystem. The implementation of the VRE-NET network scheduler was done in the Linux 2.6.0 kernel. A default rate controller was implemented as a loadable module. We compare the performance of the Linux system with and without the VRE-NET network subsystem. In all experiments, the standard Linux 2.6 CPU scheduler is used to schedule processes. All data was collected on AMD Athlon workstations. The workstations are equipped with 1.0GHz CPU, 256MB of memory and run Linux 2.6.0.

Our experiments can be partitioned into four categories: the overhead of the VRE-NET system; the reservation of network bandwidth for specific tasks; the isolation of different tasks; and automatic bandwidth adjustment. We repeated each experiment 5 times and present the mean value. The repetition of 5 times is enough because the variation of the results was small. We selected *Mplayer* as the QoS sensitive application. During the experiments, *Mplayer* runs on a client machine and decodes a video file – a 3D PC game demo – from a Web Server using http protocol.

	UDP	TCP
Processing Time without VRE-NET	5538.815	7299.345
Processing Time with VRE-NET	5808.631	7569.161
Overhead of Demultiplexing	72.792	72.792
Overhead of Scheduling	197.024	197.024
Overhead%	4.9%	3.7%

Table 1. The overhead of the VRE-NET system. The unit is nanosecond per packet.

4.1 Overhead Measurements

The first experiment measured the overhead of the VRE-NET system. The overhead comes from two components of the VRE-NET system: (1) the early demultiplexing module, and (2) the VRE-NET packet scheduler.

For this experiment, we first measured the protocol processing time for a packet in the original Linux network subsystem. The processing time is considered from the moment that the network interrupt handler is invoked to process a packet to the moment that the packet is put into a user process's socket buffer. We connected two PCs running Linux 2.6.0 via a 100Mbps Ethernet hub. An application running on one PC sent packets to an application on another PC. We measured the time for the original network subsystem processing 6,000 UDP/TCP packets. The average processing time for a UDP/TCP packet is presented in Table 1.

With the baseline established, we next measured the overhead for the early-demultiplexing and network scheduling in the VRE-NET system. The overhead is independent of the protocol type of a packet because the demultiplexer and the network scheduler do almost the same calculations for TCP and UDP packets. As before, we measured the overhead for 6,000 packets, and the average overhead for a UDP/TCP packet is presented in Table 1. The overhead percentage is calculated with Equation (3). Note that these values may vary in different environments.

$$\text{Overhead\%} = \frac{\text{Overhead of Demultiplexing} + \text{Overhead of Scheduling}}{\text{Processing Time without VRE-NET}} \quad (3)$$

As shown in Table 1, the VRE-NET system causes an overhead of 4.9% for procesing a UDP packet and an overhead of 3.7% for processing a TCP packet. The overhead only affects the CPU time that is used to process incoming packets. Thus the overhead on the whole system should be far less than the 4.9% and 3.7%. For example, we ran a sending application and a receiving application on the original Linux in this experiment senario. The sending application sent UDP packets with a size of 400 Bytes at a uniform rate of 30,000 pkts/s to the receiving application. The throughput was around 96Mbps. We observed that the receiving side used around 16% CPU time in protocol processing. That means, if the applications were running on the VRE-NET system, the overhead of the VRE-NET system put on the whole system should be $16\% \times 4.9\% = 0.78\%$. We repeated the experiment on the VRE-NET system. The receiving application was assigned a bandwidth of 100Mbps. The result showed that the receiving side used around 16.8% CPU time in protocol processing, which verified our estimate. Considering that the VRE-NET system can provide QoS to distributed applications, we claim that the overhead is acceptable.

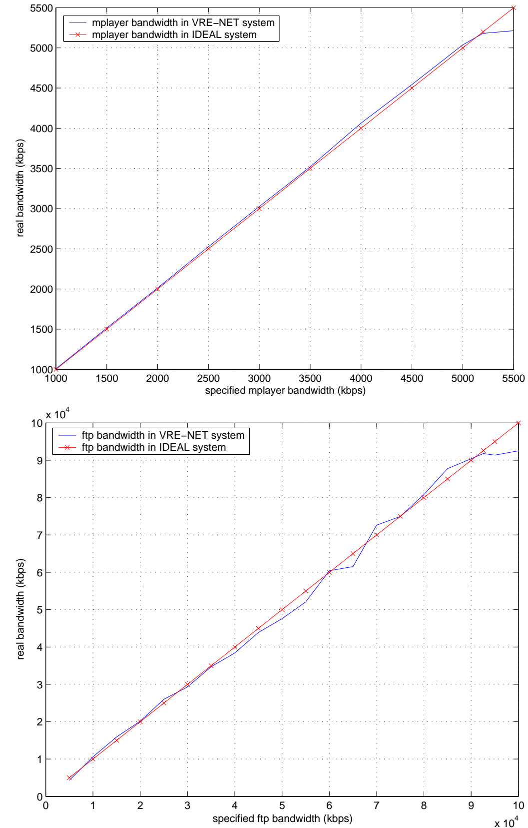


Figure 3. The actual bandwidth under different bandwidth reservations, compared to the ideal situation in which the actual bandwidth is equal to reserved bandwidth.

4.2 Bandwidth Reservation

The purpose of this experiment was to show the effectiveness of the bandwidth reservation mechanism of the VRE-NET system. The experimental setup comprised of a client, a HTTP server and a Ftp server. The three hosts are connected via 100Mbps ethernet. In the first experiment, *MPlayer* read data from the Web server and played videos on the client host. We assigned different bandwidth reservations to *MPlayer* and recorded the actual bandwidth. The

second experiment was the same as the first one except that we ran a ftp application to download a 1GB file from a Ftp server, which needs a much higher bandwidth than *MPlayer*.

As shown in Figure 3, the actual bandwidth closely approximates the assigned bandwidth in the VRE-NET system. For *MPlayer*, we specified it to decode frames at 30fps. The actual bandwidth remained unchanged when the assigned bandwidth was larger than around 5,300kbps, which was the maximum bandwidth that *MPlayer* needs to execute at 30 fps. For the file transfer application, the actual bandwidth turned out to be unchanged around 93,000 kbps, which was the maximum bandwidth that the 100Mbps network can provide. Figure 3 shows that the bandwidth reservation is accurate.

We can see from Figure 3 that the bandwidth reservation is less accurate when the specified bandwidth is higher. This inaccuracy comes from the resolution of the kernel clock. The Linux kernel creates a default kernel clock with 1 millisecond resolution, but the packets may arrive at a rate much higher than 1,000 packets per second. The precision afforded by the kernel clock is insufficient to execute network scheduling. However, the incoming packets are kept in the VRE-NET input queues and can be processed in batches. Therefore this inaccuracy is bounded, as shown in Figure 3. Of course, a higher resolution clock can be used if necessary to achieve a tighter bound of the allocation error.

This experiment also shows that the performance of *MPlayer* is proportional to the bandwidth assignment, until it gets enough bandwidth. As illustrated in Figure 4, *frames per second* increases linearly with the bandwidth assignment. It also verifies our previous assumption that the application would suspend itself when it acquires excess resources.

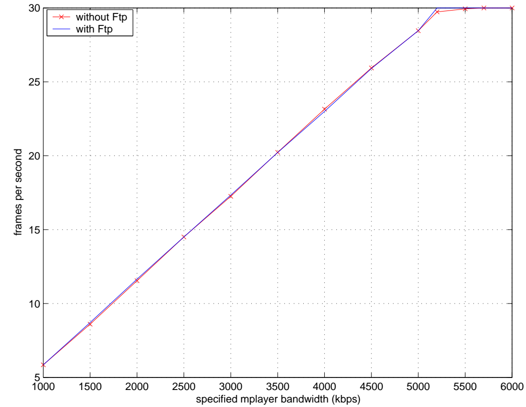


Figure 4. The Performance of MPlayer with/without the influence of Ftp application.

BW of Mplayer(kbps)	BW of FTP(kbps)	FPS
1000	6500	5.85
1500	6000	8.71
2000	5500	11.65
2500	5000	14.50
3000	4500	17.33
3500	4000	20.22
4000	3500	23.00
4500	3000	25.90
5000	2500	28.46
5300	2200	30.00
5500	2000	30.00

Table 2. MPlayer and Ftp application runs on the VRE-NET system at the same time.

4.3 Isolation

This experiment was on isolation of incoming streams. The experimental setup was the same as the previous experiment except that we used 10Mbps ethernet to make the *Ftp* and *MPlayer* contend for network bandwidth simultaneously. To establish an unmodified Linux baseline, we first ran our applications on a unmodified Linux 2.6.0 kernel. The total incoming rate was around 8,000 kbps. *MPlayer* decoded frames at 13.4 to 15.0 fps, which is much lower than the specified rate of 30.0 fps. The limited network bandwidth cannot provide sufficient bandwidth for both applications and thus the file transfer application seriously affected the performance of *MPlayer*.

With the baseline established, we ran the same applications on the modified VRE-NET system. We assigned a total of 7,500 kbps bandwidth to the two applications and the remaining bandwidth to other communications. Note that if we assign a capacity too large for the two steams, the VRE-NET system will schedule packets in proportion to their bandwidth assignments. These results are given in Table 2. *MPlayer* performed better and better when we assigned more and more bandwidth to it. Finally it achieved 30 fps when the file transfer application was running at the same time, but with only 2,200 kbps bandwidth. Figure 4 also illustrates the effectiveness of the isolation mechanism of the VRE-NET system. The curve that represents the performance of *MPlayer* without any influence of other applications and the curve that represents data in Table 2 are almost completely overlapped, which verifies that the VRE-NET system can provide isolation for user applications.

4.4 Rate Controller

The last experiment was on the rate controller. The experimental setup was the same as the second experiment. We assigned an initial bandwidth to *MPlayer* and associated a rate controller with it. The rate controller monitored the current bandwidth and estimated the real bandwidth requirements every 2 seconds.

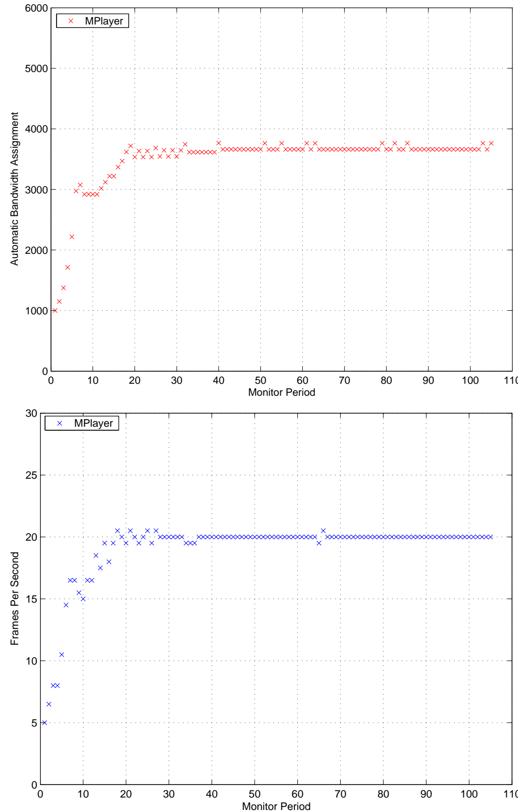


Figure 5. Automatic bandwidth adjustments for *MPlayer* with a low initial bandwidth assignment. The specified playback speed was 20 fps.

First we specified *MPlayer* to decode frames at 20 fps, which required a bandwidth of around 3,550 kbps. We assigned an initial bandwidth of 1,000 kbps to *MPlayer*. As shown in Figure 5(a), the rate controller could automatically find the required bandwidths for *MPlayer* and reserved a bandwidth of 3,662 kbps, which is a little higher than the actual bandwidth requirement of 3,550 kbps. This is a *stable state* for the rate controller as we stated in Section 3.2. We also monitored the performance of *MPlayer* every 2 seconds. As shown in Figure 5(b), *MPlayer* ran at 20 fps after the rate controller achieved the *stable state*.

Next we specified *MPlayer* to decode frames at 30 fps to simulate a higher level of QoS requirement. It follows that *MPlayer* demanded more bandwidth. According to Table 2, the actual bandwidth requirement for *MPlayer* running at 30fps is around 5,300 kbps. As shown in Figure 6, the rate controller automatically adjusted the bandwidth assignment from 1,000 kbps to 5,392 KBPS, which is enough for the bandwidth demand of *MPlayer*.

We also specified *MPLayer* to decode frames at 20 fps

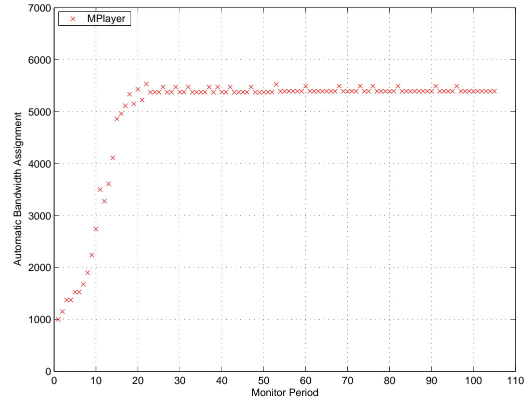


Figure 6. MPlayer with initial bandwidth assignment of 1,000 kbps. The specified playback speed was 30 fps.

and assigned an initial bandwidth of 5,000 kbps to it. The initial bandwidth assignment is much higher than the actual requirement of bandwidth of around 3,550 kbps. The rate controller, however, automatically reserved a bandwidth of 3,650kbps for *MPlayer*. The bandwidth reservation is a little higher than the actual bandwidth requirement and is considered a *stable state* for the rate controller. Due to space limitations, a figure showing these results is not presented.

5 Conclusion

QoS sensitive applications, typically multimedia applications, call for the enhancement of conventional operating systems. In this paper, we investigate the scheduling problem of the conventional interrupt-driven network subsystem and present the VRE-NET system, an adaptive QoS-supported network subsystem model. The VRE-NET system employs three key techniques: the per-process early demultiplexing, the network subsystem scheduler based on the VRE model, and a rate controller. In addition, we use a *virtual release time* in the VRE-NET system to make it suitable for the network subsystem. Together, these mechanisms ensure a fair, predictable allocation of network bandwidth.

The implementation is done on the Linux 2.6.0 kernel. The rate controller is implemented as a loadable module that can be customized by users for different application requirements. We also provide a default rate controller suitable for multimedia applications. Our experiments show the effectiveness of the VRE-NET system. This work provides a foundation for our future work to construct a loosely structured operating system, in which an application could reserve processor, network and disk bandwidths. Moreover, these bandwidth reservations could be automatically adjusted to match each other according to different QoS re-

quirements.

References

[1] Abeni, L., Buttazzo, G., "Integrating Multimedia Applications in Hard Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.*, Madrid, Spain, Dec. 1998.

[2] Banga, G., Druschel, P., and Mogul, J.C., "Resource Containers: A New Facility for Resource Management in Server Systems." *In Proc. of the 3rd USENIX Symposium on Operating Systems Design and Implementation,* pp. 45–58, New Orleans, LA, February 1999.

[3] Bavier, A., Voigt, T., Wawrzoniak, M., Peterson, L., and Gunningberg, P., "SILK: Scout Paths in the Linux Kernel," Technical Report 2002-009, Department of Information Technology, Uppsala University, Uppsala Sweden, 2002

[4] Banachowski, S., and Brandt, S., "The BEST scheduler for integrated processing of best-effort and soft real-time processes," *in Proceedings of Multimedia Computing and Networking 2002 (MMCN 02).* , pp. 46–60, Jan. 2002.

[5] Brandt, S., Banachowski, S., Lin, C., and Bisson, T., "Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes,"*In Proceedings of IEEE International Real-Time Systems Symposium,* Dec. 2003.

[6] Brustoloni, J., Gabber, E., Silberchatz, A., and Singh, A., "Signaled Recerver Processing", *Proceedings of the USENIX 2000 Annual Technical Conference,* June, 2000.

[7] Druschel, P., Banga, G., " Lazy Receiver Processing: A Network Subsystem Architecture for Server systems", *Usenix Symposium On Operating System Design and Implementation*, October 1996, pp. 261-275.

[8] Ghosh,S., Rajkumar,R.," Resource Management of OS Network Subsystem," *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC) 2002*, Washington, D.C.

[9] Goddard, S., Liu, X., "A Variable Rate Execution Model," *in Proceedings of the 16th Euromicro Conference on Real-Time Systems,* Catania, Italy, July 2004.

[10] Goyal, P., Guo H, X., Vin, M., "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proceedings of 2nd Symposium on Operating System Design and Implementation (OSDI'96),* Seattle, WA, pages 107-122, October 1996.

[11] Jeffay, K., Smith, F., Moorthy, A., Anderson, J., "Proportional Share Scheduling of Operating System Services for Real-Time Applications," *In Proceedings of IEEE Real-Time Systems Symposium,* Dec. 1998.

[12] Jeffay, K., Goddard, S., "A Theory of Rate-Based Execution," *Proceedings of the IEEE Real-Time Systems Symposium,* Phoenix, Arizona, December 1999, PP. 304-314.

[13] Liu, X., Goddard, S., "Scheduling Legacy Multimedia Applications," *Journal of Systems and Software,* July 2004.

[14] Liu, X., Goddard, S., "Supporting Dynamic QoS in Linux," *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium,* Toronto, Canada, May 2004.

[15] Nieh, J., Lam, M., "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," *Proc. of the 16th ACM Symposium on Operating Systems Principles,* Saint-Malô, France, Oct. 1997, pp. 184-197.

[16] Oikawa, S., Rajkumar, R., "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior" *In Proceedings of the IEEE Real-Time Technology and Applications Symposium,* Vancouver, June 1999.

[17] Rajkumar, R., Juvva, K., Molano, A., Oikawa, S., "Resource Kernels: A Resource-Centric Approach to Real-Time Systems" *In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking,* January 1998.

[18] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J., Plaxton, C., "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems," *Proc. 17th IEEE Real-Time Systems Symposium,* Dec. 1996, pp. 288-299.

[19] Sundaram, V., Chandra, A., Goyal, P., Shenoy, P., "Application Performance in the QLinux Multimedia Operating System," *Proceedings of the Eighth ACM Conference on Multimedia,* Los Angeles, CA, pages 127-136, November 2000.

Developing a Complete Integrated Real-Time System

Scott A. Brandt, Scott Banachowski, Caixue Lin, and Joel Wu
Computer Science Department
University of California, Santa Cruz
{sbrandt, sbanacho, lcx, jwu}@cs.ucsc.edu

Abstract

Modern systems are frequently called upon to support mixes of applications with different types of timeliness requirements. Current solutions for supporting such mixes are ad hoc *and do not guarantee the requirements of all types of processes. We discuss the need for better systems support for such mixes and present partial solutions toward the development of such systems. These include an integrated real-time scheduler that focuses on best-effort performance, a slack scheduler designed to improve the performance of soft real-time processes, and an integrated soft real-time disk bandwidth manager.*

1. Introduction

Application timeliness requirements vary from hard real-time to best-effort with many flavors in between, including soft real-time, firm real-time, rate-based, *etc*. Many application scenarios ranging from desktop multimedia to large distributed real-time systems require the ability to simultaneously support these different types of requirements, yet existing systems provide little direct support for more than one type. We argue that (1) all systems should support the full range of possible timeliness requirements, and (2) this can only be accomplished with resource management algorithms that are hard real-time at their core. Toward this goal we present a CPU scheduler appropriate for use as the only scheduler in such a system, a slack scheduling algorithm designed to optimally address the needs of *other* tasks, and a disk bandwidth manager capable of handling a variety of timeliness requirements.

Our CPU scheduler, HodgePodge, supports integrated hard real-time, soft real-time, and best-effort scheduling. Unlike many previous real-time schedulers, HodgePodge focuses primarily on the performance of best-effort applications running concurrently with hard or soft real-time applications. Our results demonstrate that it is feasible and practical to combine the processing of these different types of applications without degrading the performance of any of them. We believe this is a necessary first step towards the development of the type of integrated real-time systems that we envision.

In integrated systems with many different types of processes, hard (and occasionally soft) real-time tasks frequently over-reserve resources in order to ensure that they meet deadlines. Whenever these processes use less resources than they have reserved, dynamic slack is generated. The performance of soft and non-real-time processes heavily depends upon the availability and efficient distribution of this slack. Integrated real-time systems therefore demand the development of algorithms that distribute slack so as to best support the goals of the other processes. Our slack scheduling algorithm, SLASH, does this by distributing slack as soon as it is available, to the process with the earliest deadline, and allows jobs of a task to borrow resources from future jobs of the same task. This provides the resources to the most critical job as early as possible while guaranteeing the correctness of the schedule.

Real-time systems research often focuses on CPU resource management. Other resources, and especially I/O devices, are managed minimally, if at all. Solutions that do address these resources tend to focus on a single type of timing requirements: hard, soft, or non-real-time. However, integrated systems that will support a variety of timing requirements must manage resources other than the CPU, and must do so in ways that directly support different timing requirements. Our Hierarchical Disk Sharing (HDS) algorithm begins to address this problem in the domain of disk I/O. Based on a technique developed for networking, HDS partitions the disk bandwidth and allows processes to reserve fixed or relative shares of the available bandwidth. HDS addresses some of the unique issues that arise in managing disk I/O, including the non-uniform (and only partially deterministic) service times associated with disk requests.

So far we have developed CPU and disk allocation algorithms for integrated systems, but are just beginning to combine them into a single system. The following sections discuss HodgePodge, SLASH, and HDS in more detail.

2. HodgePodge

General-purpose operating systems are designed to serve a wide variety of applications. Yet because these systems use best-effort CPU scheduling, the growing body of applications that have time constraints remain unsupported. We envision a merging of real-time scheduling techniques with general-purpose systems. The advantage comes from two perspectives: (1) general-purpose systems need not treat real-time applications (*e.g*. multimedia) in an *ad hoc* fashion, and (2) real-time systems need not treat non-real-time applications in an *ad hoc* fashion.

The definition of "general-purpose" has grown to include the kind of tasks traditionally thought of as real-time. Examples include: games, video players and encoders, home studio software for multi-track audio generation/recording/sequencing, voice recognition, and hardware emulation tasks such as soft-modems. Many other applications benefit from the fine-grained partitioning and isolation of resources real-time schedulers enforce, such as virtual sharing of processors by web servers, or reservation of CPU for highly compute-intensive data consumers such as scientific applications or search engines. In our experience these applications are treated in an *ad hoc* fashion. Typically, because they tolerate some degree of missed deadlines, the tasks are scheduled in the default best-effort manner and mostly meet deadlines because the CPU resource is over-provisioned (or by luck when processor load is high). Other options include playing with the *nice* priority, which is not all that predictable or robust, or overriding the scheduler by choosing a high static priority (a technique sanctioned by a POSIX standard [12]). The latter approach is unsafe when the task is not designed to be cooperative or is buggy [17]. And this approach is not scalable in the case where multiple tasks need real-time support. Our approach is to provide an integrated real-time scheduler for all tasks.

Conversely, many real-time systems share some general-purpose requirements. Although the primary functions of flight, defense, and manufacturing control systems are real-time, they commonly include many non-critical, yet non-trivial, tasks that are best served using traditional time-sharing techniques. In our experience these applications are treated in an *ad hoc* fashion. In a real-time system, non-real-time tasks are often deemed unimportant, and left to background processing, when they instead prefer time-share disciplines. More sophisticated approaches use hierarchies of schedulers [6, 10], allowing co-existence of multiple schedulers for different kinds of tasks. However, an arbitrary scheduling hierarchy may become (needlessly) complex, and even in simple hierarchies the effect of stacking schedulers must be well-understood to ensure meeting constraints [19]. Our approach is to provide a simple mechanism for time-sharing the non-reserved resources of a real-time system.

The goal of the HodgePodge (Holy-Grail, Pipe-Dream) CPU scheduler is to support a veritable hodgepodge of processing or timeliness constraints. HodgePodge uses a real-time scheduler, EDF [15], for all tasks, whether they have time constraints or not. Time-sharing is provided by a resource allocation layer that uses the reservation capability of the real-time scheduler [3]. To make such a system desirable for general-purpose, it should in all appearances mimic a time-share scheduler except when called upon to run a real-time task. Our previous experience showed that, using aperiodic bandwidth servers for non-real-time tasks, we may get behavior similar to time-share algorithms in terms of responsiveness and overhead [2]. The novelty is to provide time-share-like service by adapting each application's aperiodic server parameters during run-time based on behavior.

2.1. The Best-effort Bandwidth Server

General-purpose systems behave unpredictably, because it is not known *a priori* which tasks will run, or when. In contrast, a periodic real-time task is predictable: it is a sequence of *jobs*, where each job begins at the start of a period, and completes at or before the end of the period. Non-real-time tasks may not resemble periodic tasks at all—however, during execution tasks can still be modeled as a sequence of jobs. These jobs may not begin or end in periodic (or even predictable) intervals, so it is therefore natural to treat these tasks as *aperiodic*.

An *aperiodic server* is an algorithm that assigns periodic deadlines to tasks that are not necessarily periodic, or which have no deadlines. Modern processors have the processing headroom for dynamic scheduling, and our previous work shows that the overhead of using an aperiodic server for each task is akin to existing time-share schedulers. It follows that it is practical to use a real-time scheduler as the core of a general-purpose scheduler, with aperiodic servers for non-real-time tasks. The advantage is providing real-time scheduling as a native feature, without resorting to an *ad hoc* addition or combination of schedulers.

The Best-effort Bandwidth Server (BEBS) is an aperiodic server that addresses the two main goals of time-share scheduling: fairness and better responsiveness for interactive tasks. It achieves fairness by adjusting the reservations of tasks equally, and allocating the reclaimed slack fairly. Slack is any CPU that is unreserved, and any reserved CPU that is unused. IRIS [16] is a server designed to reclaim slack fairly, and BEBS is similar to IRIS, with differences noted in our previous report [2]. To meet the interactive goal, the server adjusts its period according to the run-time behavior of the task: interactive servers have shorter periods for better responsiveness, while compute-bound

servers have longer periods (which incur less scheduling overhead). Each server is assigned a utilization equal to a fair-share allowance of CPU bandwidth.

2.2. A Brief Comparison

To illustrate the difference in operation between a traditional time-share scheduler and HodgePodge we describe a simple scenario. Imagine N tasks executing, all using equal amount of CPU. In a time-share system based on multi-level feedback queues, such tasks will reside in the same priority queue and receive service in round-robin quanta of length q. In the worst-case the longest wait for service is $(N-1)q$, and the task will receive at least q amount of service in $q \times N$ amount of time.

In HodgePodge, each task is assigned a reservation by the resource allocation algorithm. In the above workload, a reservation equal to $(p := qN, u := 1/N)$ will, in the worst-case, give q amount of service in $q \times N$ amount of time, the same as the time-share system. In HodgePodge, once set, this reservation will be guaranteed, independent of other activities in the system. This is an advantage over traditional time-share scheduling.

Now consider what happens if tasks differ in interactivity. In the time-share system, they will be served from different queues, with higher priority tasks preempting lower priority tasks. It becomes difficult to predict exactly when a task will receive a full quantum q of service, because the service of its queue may be preempted by other tasks for any unknown number of durations.

In HodgePodge, the performance of tasks is adjusted by controlling server reservations at run-time, based on the past task activity. Interactive tasks do not receive higher priority, but instead receive reservations consistent with their past execution, for example a reduced utilization but increased periodic rate (scaled in accordance with the level of interactivity and other factors such as *nice* setting). Interactive tasks likely preempt less-interactive tasks because their periodic deadlines are likely earlier when active; in the average case they remain responsive. However, each task is still guaranteed a reservation, so all receive a predictable level of service in accordance with the time-share goals. We have found that while running a mix of real and non-real-time applications, the performance of time-sharing in this approach is significantly better than assigning real-time tasks higher priorities, while at the same time there are no violation of real-time constraints [2].

The reservation policies can be tuned to mimic the expected performance from Linux or any other time-share scheduler. An enhancement to the algorithm also tries to auto-detect multimedia and other periodic soft real-time applications while they execute, and make reservations consistent with their inferred requirements (such as by the measured period of frame synchronizations). In this way, the system better supports legacy periodic applications.

2.3. HodgePodge Implementation

In order to build a HodgePodge prototype and test and use BEBS in a general-purpose environment, we implemented EDF in Linux [14]. We replaced the Linux scheduler while leaving as much of the existing infrastructure intact. This is not the approach we'd take if implementing from scratch; the existing structure of Linux definitely impacts our design and performance. For example, some process accounting occurs during a periodic timer interrupt that is not necessary for EDF. However, disabling this interrupt also disables mechanisms for timeouts and synchronization used by many device drivers and applications. Also, removing the interrupt would require significant changes to some of the process accounting. Rather than disable and re-implement portions of the kernel, we decided to leave them intact, and when applicable leverage them for our purpose.

Linux uses a 1000 Hz periodic timer to drive many operations (a.k.a the *tick* timer). We leverage this interrupt to schedule the release of jobs. All processes in the system are treated as sequences of jobs that begin at periodic intervals. Each job has a processing budget (or quantum) equal to $u \times p$ (determined by the reservation tuple (p, u)). When its budget is used, a job suspends until the start of the next period. By using the tick timer for scheduling these events, we do not need to support arbitrary release times or periods, and reduce the number of interrupts and task preemptions.

Because all task jobs must be released on 1 ms. intervals, the minimum period of a reservation is bounded to an integral number of milliseconds. We call the occurrence of 1000 Hz clock interrupt a *major tick*. Currently there are no sub-millisecond periods. For this system we expect most real-time workloads to involve multimedia rates of at most 50 Hz, so this is currently sufficient for our purpose.

The EDF scheduler enforce reservations by using a timer interrupt to prevent tasks from overrunning their budget (similar to the R-EDF implementation [23]). Periods are scheduled at relatively coarse-grained times, but task budgets may be a fraction of a *tick* interval, requiring a higher resolution timer to trigger a reschedule when a job's budget expires (we use the Pentium-class APIC timer). Thus a one-shot timer only needs to be programmed if an expiration occurs before the next *major tick*. This implementation resembles firm timers [9], because the actual overhead of setting up hardware is avoided when a task is preempted before its budget expires. Also, only a single one-shot timer must be maintained at any time, so we need not maintain lists of timer events. The one-shot timer is programmed to the nearest microsecond, and we call these

clock intervals *minor ticks* (although unlike *major ticks*, there are not periodic interrupts at every *minor tick*).

Every task is assigned a utilization u, which is its allocated fraction of CPU, and dictates the maximum amount of time (budget b) it may execute per period p ($b = up$). The granularity of the one-shot timer limits the budget we may assign to at most 1 μs.[1] To simplify reservations, the system requires the utilization be set in an increment of 0.1% (1 μs per ms period). Since a task must have some utilization, this limits the number of servers we may admit to at most 1000 (however a server may service multiple tasks, so this is not a limitation on task number).

EDF selects the task with the earliest deadline, requiring an $O(n)$ search of n runnable tasks. Previous versions of Linux (< 2.5) also required $O(n)$ selection, but the newer versions bound the search to a fixed number (locating the first non-empty priority queue). Our EDF algorithm is not quite as scalable as Linux's new constant-time algorithm, however it is better than the previous version of Linux.

We shift overhead from the selection code to the queue insertion by keeping the run queue sorted in deadline order. On average, inserting into an already sorted list is better—we found that for random task sets, sorted inserts averaged less than 3 times fewer operations than searching the unsorted list. This gives us less overhead than the previous generation Linux. We are looking into further optimizations in queuing structures to reduce overload for large sets of servers. [2]

2.4. Future Directions

In order to build the HodgePodge prototype and test and use BEBS in a general-purpose environment, we implemented EDF in Linux [14] by replacing the kernel's scheduler. Changing Linux's CPU scheduler alone does not make it real-time, but better equips it to handle workloads with time constraints. An existing problem is that CPU used by the kernel is charged to the currently running task, even if the work is on behalf of another. Using a technique such as Augmented CPU reservations [18] may track the time "stolen" from tasks by OS operations, making our CPU allocations better match reservations.

BEBS could be incorporated into a hard real-time environment by adapting techniques used by DROPS [11], which allow time-share and real-time applications to coexist on a real-time micro-kernel, or RTLinux [22], which runs Linux as a low-priority task on a real-time executive. Both approaches treat the time-share portion of the system as a single user-program; our approach dictates that the time-share portion be treated as a set of servers, with each a corresponding user-program. This is an area for future research.

[1] Budgets this small are impractical for Linux. We measured the average context switch time on a 2.4 Hz P4 to be about 5 μs, so a task with 1 μs budget will consume more time in context switch, not to mention cache warming, than its budget allows.

[2] For handling a larger number of tasks we may consider using a sorted heap with $lg(n)$ insertion and deletion.

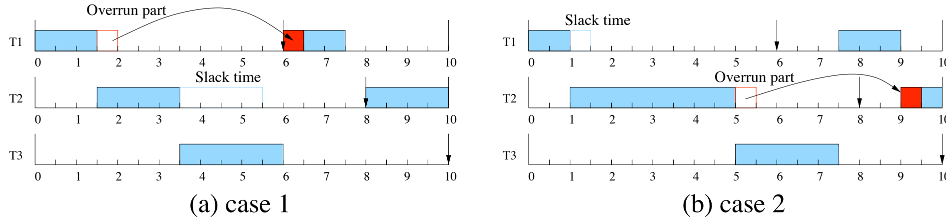
3. SLASH

The increasing demand for more powerful computing platforms and applications requires modern operating systems capable of simultaneously supporting applications with a variety of different time constraints. The hierarchical HLS scheduler [19], and the flat integrated RBED [3] and closely related HodgePodge schedulers are examples. Such systems simultaneously support (1) critical hard real-time applications such as external signal sampling and processing, (2) non-critical soft real-time applications such as desktop multimedia, and (3) best-effort applications such as compilers, word processors, *etc*. Hard real-time applications make worst-case resource reservations to guarantee their constraints; soft real-time applications may reserve less than worst-case to achieve good average-case performance; and best-effort applications generally make no reservations beyond what is necessary to avoid starvation. Any variance in execution times below what has been reserved leads to dynamic slack—reserved but unused resources. Efficient distribution of this slack to processes whose current needs exceed their reservation can significantly improve the performance of both soft real-time and best-effort applications.

SLASH is a slack scheduling mechanism system specifically designed to improve the performance of soft real-time applications while guaranteeing the worst-case reservations of hard real-time processes. Our evaluation shows that SLASH reduces the number of missed deadlines and decreases the average tardiness of late deadline for soft real-time applications when both hard real-time and soft real-time applications coexist. In our experiments, SLASH always misses fewer deadlines than CBS [1], BEBS, and CASH [5], reducing missed deadlines by 70%, 70% and 10% (respectively) in the best scenario we observed.

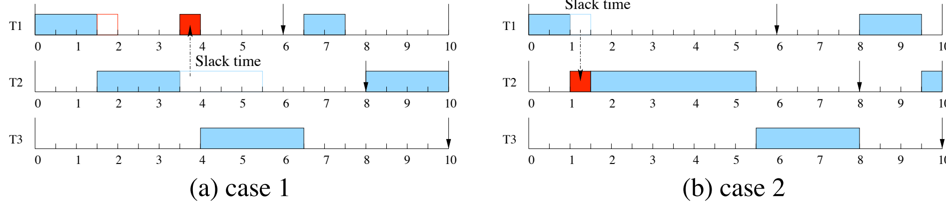
3.1. SLASH Design and Implementation

SLASH is implemented in RBED [3], which uses an integrated scheduler for hard, soft, and non-real-time processes. The low-level scheduler is earliest deadline first (EDF). Processes use the scheduler by associating their tasks with a rate-based server, conceptually similar to CBS and other bandwidth servers. A server is characterized by a reservation tuple (B_s, P_s), where B_s is the execution budget and P_s is the period (both in units of time). The server utilization is $U_s = \frac{B_s}{P_s}$. A deadline occurs at the end of the each period. Each hard or soft real-time task is associated with



(a) case 1 (b) case 2

Figure 1. Drawbacks of idle-time slack management



(a) case 1 (b) case 2

Figure 2. SLASH solves both problems

its own server. Periodic and aperiodic best-effort tasks are scheduled as soft real-time tasks. All other best-effort tasks are served by one server in first come, first serve order.

SLASH combines our **SLA**ck **D**onation (SLAD) algorithm [13] with the CASH slack management algorithm [5]. In SLAD, when a task completes, it *immediately* donates any remaining budget to the task whose server has the earliest deadline. This server is the most critical, the most likely to be near the completion of its current job, and the least likely to benefit from any later donations of slack.

Figure 1 shows two problems that can occur when slack is only made available to tasks when all other tasks are idle, a technique common among other algorithms. The problems are that slack cannot be used to prevent a deadline miss caused by either (1) a past overrun or (2) a future overrun. The examples show three soft real-time tasks, $T1$, $T2$, and $T3$, with the following respective reservation configurations: $(B_1 = 1.5, P_1 = 6)$, $(B_2 = 4, P_2 = 8)$ and $(B_3 = 2.5, P_3 = 10)$; the CPU is 100% utilized. Each task has an actual deadline coinciding with its server deadline, and may overrun its reserved budget. In Figure 1(a), the first job of $T1$ has an actual execution time of 2, exceeding its reservation by 0.5, and the first job of $T2$ has an actual execution time of 2, 2 less than its reserved budget. With idle-time slack management, the overrun portion of $T1$ does not resume execution until time 6, missing its deadline. In Figure 1(b), the first job of $T1$ has an actual execution time of 1, 0.5 less than its reservation; the first job of $T2$ overruns by 0.5. Again with idle-time slack management, the slack is "pushed back" and unused until a later idle point, resulting in $T2$ missing its deadline. By donating slack to other processes as soon as it is available, SLASH solves both of these problems. The resulting schedules are shown in Figure 2. In both cases no task misses its deadline, despite the overruns.

In CASH [5], when a server becomes idle, any remaining budget is recorded in a queue. When a server runs, it first consumes all queued budgets with deadlines $\leq$ its own. When a server consumes its budget, it is recharged and its deadline extended by one period, allowing it to borrow against its future budget to complete the current job. This has some benefits, allowing current jobs that need more CPU to safely borrow from future jobs of the same task, which may need less CPU (or may borrow from still more future jobs). Unfortunately, this borrowing may prevent slack from getting to the tasks that need it most—when a task overruns its budget, its server deadline will be postponed before the task completes its current job, reducing its EDF priority and making it less likely to receive slack.

SLASH addresses this problem by combining the SLAD EDF-based slack donation mechanism with the CASH greedy budget replenishment mechanism, donating slack to the tasks that need it the most and executing overrun tasks as early as possible (so that they improve their chance of meeting deadlines) by not forcing servers to remain inactive when their budget is consumed.

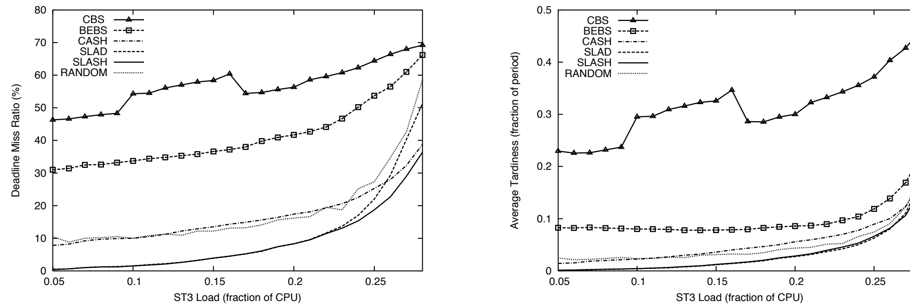
3.2. The SLASH algorithm

SLASH uses an earliest virtual deadline first (EVDF) for slack scheduling decisions. The virtual deadline of a server is calculated as follows:

$$vd_{s,k} = d_{s,k} - \left\lfloor \frac{d_{s,k} - t}{P_s} \right\rfloor P_s$$

where t is current time (note that the condition $d_{s,k} > t$ always holds). Like CASH, SLASH has no *expired* status. When a SLASH server consumes its budget, the budget is recharged, its deadline is advanced by one period, and its state is reset to *waiting*. The algorithm is as follows:

1. At the beginning of each period, the current budget of a server c_s is set to its reservation budget B_s, its



(a) Deadline Miss Ratio as a function of load (b) Average Tardiness as a function of load

Figure 3. Load effect on performance (one soft real-time task, $p = 310$)

dynamic deadline $d_{s,k}$ is set to $d_{s,k-1} + P_s$, and its state is set to *waiting*.

2. The *waiting* server with the earliest deadline becomes *running*.
3. A *running* server executes its pending task on the CPU until it has finished its task or consumed its budget, and decreases its budget c_s by the actual amount of CPU consumed. If it has no pending task, it donates any remaining budget to:
   (a) the task of the *waiting* server with the earliest virtual deadline; or, if none exist, to
   (b) the idle task
4. When c_s of a *running* server equals zero, the server is recharged with full budget $c_s = B_s$, its deadline is incremented $d_{s,k} = d_{s,k} + P_s$, and its state is reset to *waiting* (or remains *running* if it still has the earliest deadline).
5. When a *running* server is preempted, its state is set to *waiting*.

Step 3 implies that if servers always have their own associated tasks to execute, then no slack scheduling occurs. If there is slack available, it is immediately donated to other pending tasks whose servers have the highest priority determined by EDF. It is possible for other slack scheduling choices besides EDF to perform better for certain applications or in certain easily contrived circumstances. Nevertheless, SLASH is simple, straightforward, and effective in practice.

3.3. SLASH Performance

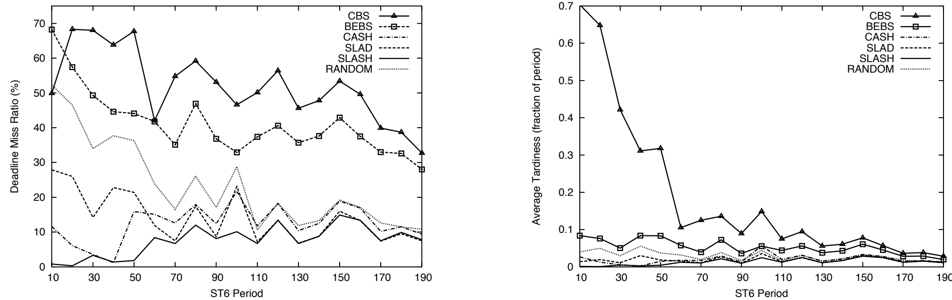
We compare the performance of SLAD and SLASH to CBS, BEBS, CASH, and a RANDOM algorithm (which provides aggressive slack donation like SLAD but, instead of using EDF, assigns slack to a random task). All hard real-time tasks meet their deadlines. Our metrics of soft real-time performance are deadline miss ratio (DMR), and average tardiness (ATD).

The first experiment examines soft real-time performance as a function of system load. The workload consists of two periodic hard real-time tasks and one periodic soft real-time task. In this experiment HRT1 has constant execution time equal to its server budget, HRT2 has normally distributed execution times with its server budget equal to the worst-case, and SRT3 has normally distributed execution times with its server budget set to their average. SRT3 will often overrun its budget but should meet most of its deadlines by taking advantage of the slack from HRT2.

Figures 3 and 4 show SRT3's deadline misses and tardiness, using the different algorithms, as a function of utilization between 5% and 29% (in all cases, the total average sum of server utilization is 100%). RANDOM, SLAD, and SLASH outperform CBS and BEBS, demonstrating the benefit of donating slack at the earliest possible time. SLAD and SLASH outperform RANDOM, demonstrating the additional benefit of giving the slack to the process with the earliest deadline. SLAD and CASH outperform each other in different circumstances. Finally, SLASH outperforms both SLAD and CASH, demonstrating the effectiveness of combining SLAD slack donation with CASH budget replenishment.

The second experiment shows soft real-time performance as a function of server (and task) period. The workload consists of five periodic hard real-time tasks and one periodic soft real-time task. Every hard real-time task has execution times fitting a normal distribution and a server budget set to their worst-case execution time. SRT6 has normally distributed execution times with its server budget set to the average. Each hard real-time task reserves 10% of the CPU and SRT6 reserves the remaining 50%.

Figure 4 shows SRT6's performance as a function of period ranging from 10 to 190. We see results similar to the previous subsection: (1) RANDOM, CASH, SLAD, and SLASH outperform CBS and BEBS, (2) SLAD and CASH outperform each other in different circumstances, and (3) SLASH is always the best. Interestingly, as we increase the number of soft real-time tasks (and decrease their target utilization and reservations accordingly), we find that



(a) Deadline Miss Ratio as a function of period (b) Average Tardiness as a function of period

Figure 4. Period effect on performance (one soft real-time task, $u = 50\%$)

the performance of SLASH improves (not shown).

In our experiments, SLASH always outperforms CBS, BEBS, and CASH in this regard, reducing missed deadlines by 70%, 70% and 10% (respectively) in the best scenario we observed. Although designed for our integrated real-time system, RBED [3], SLASH should work equally well with any deadline-aware scheduler.

4. HDS

Systems that use or serve multimedia data require timely access to data on hard drives. To ensure adequate performance in an integrated real-time environment, users must either prevent overload of disk resources, not generally feasible in a general-purpose environment, or use real-time algorithms that rely on intricate knowledge of disk internals to meet deadline requirements. Hierarchical Disk Sharing (HDS) allows disks to be (nearly) fully utilized while sustaining bandwidth reservations, without requiring detailed knowledge of the drive internals. Derived from hierarchical link sharing for networks [8], HDS uses a hierarchy of token bucket filters to isolate disk access among clients and groups of clients, and to allow for reclaiming of unused bandwidth, capabilities that are absent in current commodity operating systems and which are necessary to support time constraints in an integrated system.

One of our main design goals is that the reservation mechanism be independent of high-level features like filesystems, and low-level features like disk schedulers, so that it can be employed across many systems, including storage network devices. Therefore we chose to implement our prototype of HDS in the block device layer of the Linux kernel. We discuss the design of HDS and present our Linux implementation, demonstrating both the effectiveness (and limitations) of this approach.

4.1. HDS Design and Implementation

Traditional disk access is best-effort, with no timing guarantees. Acceptable performance is achieved when the disk is not overloaded. When demand for disk bandwidth exceeds the supply, all applications may experience performance degradation, including those with time constraints. Our approach to this problem is to provide a mechanism that allows reservations of disk bandwidth, graceful degradation under heavy load, and reclaiming of unused reservations. A hierarchical structure for resource sharing provides a basis for meeting all of these goals.

Specifying a disk reservation by bandwidth is intuitive, but disk bandwidth is not constant; service times vary depending upon the initial position of the read-write head, the position of the requested data on the disk, the low-level disk scheduling algorithm, *etc*. Translating bandwidth requirements into low-level disk operations is a complicated task [7]. Alternatively, specifying a disk reservation by a reserved time-slice (instead of bandwidth) may result in different amounts of data being retrieved per time-slice.

Existing reservation-capable schedulers contend with this issue. The Cello [20] scheduler presents two methods of accounting, either by size or time. Some schedulers allow reservations in terms of number of requests [4, 21]. However, requests may also vary in size and service time. To fulfill QoS goals, the system must provide performance in line with the users' expectations, regardless of the amount of work that the disk is actually doing on behalf of different users. We expect users to perceive the quality of service for disk by the bandwidth (data rate) that it can provide. Therefore HDS accounts for disk usage in terms of bandwidth and does its accounting based on the actual amount of data transferred.

In HDS, a disk's bandwidth is divided between applications in a hierarchical tree structure; an example is pictured in Figure 5. Each leaf node represents a point of control for accessing the disk, and is associated with a Linux process. When a process first attempts to access the disk, a leaf node is created and added to the tree. When it quits, its node is removed. Non-leaf nodes are called *classes*, and represent a group of clients. The children of a class node may be leaf nodes or other class nodes.

Figure 5 demonstrates using classes to isolate best-effort

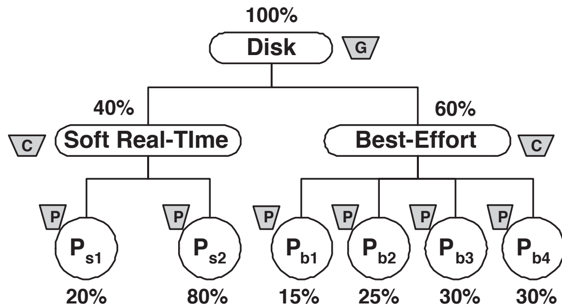


Figure 5. HDS allows arbitrary mix of shares controlled by Global bucket, Class buckets, and Process buckets

and real-time processing, an approach useful for multimedia servers where the requests with time constraints should be isolated from other traffic. Our system has an interface for constructing the desired class hierarchy, including dynamically adding and deleting classes.

Each node x has an associated reservation r_x, determined by the reservations of nodes above it in the tree structure. There are two modes for a node to specify its reservation: either an *absolute* fraction f_x, or a *relative* fraction, based on a weight w_x, of the parent node's reservation. The root node has an absolute fraction of 1. If a node x has an absolute reservation f_x (between 0 and 1), its reservation is this fraction of its parent's reservation. For example, because the root node has $r_0 = 1$, a child of the root with $f_x = 0.4$ will have a reservation of $r_x = f_x r_0 = 0.4$. The sum of absolute fractions among any node's children may not exceed 1. A class's bandwidth that is not used by absolute reservations is shared by its other children in proportion to their relative weights.

By default, clients are added to a parent node with equal weight to promote fair sharing. When a client needs a higher level of service than others, it may do so by either increasing its weight or requesting an absolute fraction of its parent's bandwidth. If the nodes on the path from a client to the root all have absolute reservations, then the client effectively reserves a static fraction of the total disk bandwidth; if any node in this path has a relative reservation, the node's reservation may vary when other nodes join or leave the structure. HDS allows administrators to set permissions for adding classes or nodes, changing reservations, admission control, *etc*.

Disk bandwidth may be controlled at different points in the I/O stack. HDS resides at the block-device layer, between the file system and disk scheduler. The regulation of disk bandwidth in HDS is implemented using token bucket filters. In order to make disk requests, a client must possess tokens. In HDS, each token represents 1 KB of data, meaning a request for 16 KB of data requires 16 tokens. Each node x in the hierarchy has an associated bucket, which may hold up to N_x tokens. When a client request is serviced, tokens are removed from its bucket. Tokens are replenished at a rate corresponding to the client's reservation. If the root token rate is T_0, then its child node x with reservation r_x will replenish tokens at rate $T_x = r_x T_0$. The root token rate represents the entire available bandwidth of a disk.

Although every node has a token bucket, only leaf nodes make requests. The token buckets of non-root nodes facilitate sharing of bandwidth. In addition to its own tokens, a node may use tokens from its parent (which in turn may use those of its parent). The effect is that unused bandwidth is shared first among nodes of the same class, then among parent class, and eventually, globally.

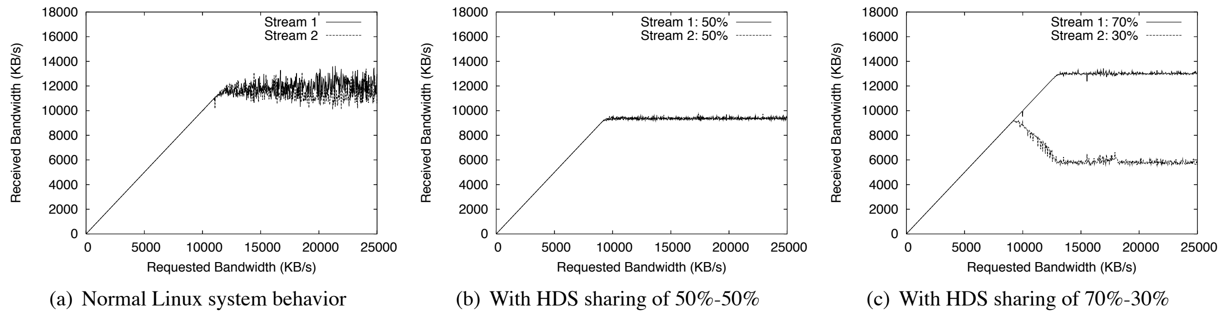
When a node drains tokens from a bucket, it also drains those from buckets in the path up to and including the root. The result is that when a class's children make disk requests, the class's tokens will be drained as well. If some of the children are not fully using their reservation, the parent will have surplus tokens. These tokens are available to other children when their own supply runs out, so that a node that has exceeded its reservation may still be able to proceed. Bandwidth isolation is preserved not only between leaf nodes, but at the class level and, in fact, at every level of the hierarchy.

4.2. HDS Performance

We ran several experiments to demonstrate the ability of HDS to shape disk traffic, using synthetic applications to generate disk workload. We focus mostly on read workloads both because multimedia is typically read-intensive and because write performance is often aided by buffering. Our test system is a 1.5 GHz P4 with 512MB of RAM. The disk is a Seagate ST340810A IDE drive formatted with the ext2 file system.

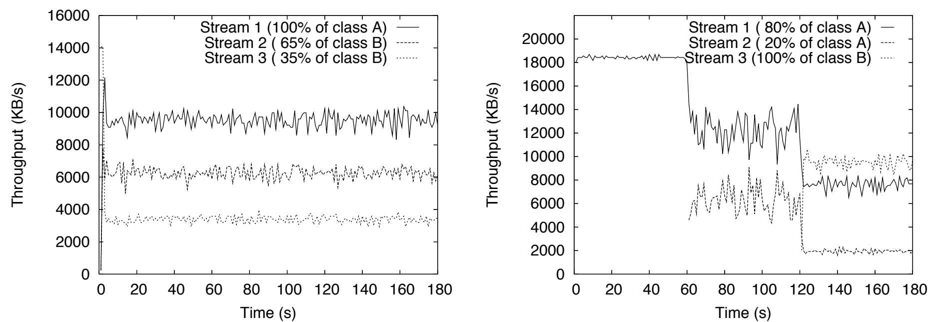
Figure 6 shows a situation where disk bandwidth has become saturated by two processes reading from the disk. The x-axis shows the requested bandwidth and the y-axis shows the measured received bandwidth. Figure 6(a) shows the result on unmodified Linux. Both processes receive their desired bandwidth until the disk becomes saturated with requests. There is no isolation, so at that point actual throughput is unpredictable and varies considerably.

HDS provides reservation and isolation of bandwidth. Figure 6(b) shows the same experiment with HDS, where each task reserves equal relative weight. At saturation the bandwidth divides evenly between the streams and achieved throughput is very stable. This fair-sharing comes at the expense of slightly lower overall disk throughput because we limit the number of requests. Figure 6(c) demon-



(a) Normal Linux system behavior (b) With HDS sharing of 50%-50% (c) With HDS sharing of 70%-30%

Figure 6. The effect of overload on throughput



(a) Isolation of bandwidth (Class A and B reserve 50% each. All streams are greedy) (b) Using unassigned bandwidth (Class A and B reserve 50% each. All streams are greedy)

Figure 7. HDS isolation and slack reclamation

strates the effect of allocating 70% of the disk to stream 1 and 30% to stream 2.

The next experiment shows the ability of HDS to provide hierarchical resource sharing. We created two classes, A and B, each reserving 50% of the disk. Stream 1 belongs to Class A, so it reserves 100% of the class reservation. Streams 2 and 3 belong to Class B, and reserve 65% and 35% of Class B's reservation, respectively. Figure 7(a) shows that all three streams receive bandwidth corresponding to their allocation, with no interference from each other.

Excess bandwidth may be available when a process needs more than its reserved share. Figure 7(b) shows this scenario. In this experiment, there are two classes and three streams. Class A and B each reserve 50%. Stream 1 and 2 belong to Class A and reserve 80% and 20% of its bandwidth. Stream 3 belongs to class B, so receives 100% of its bandwidth. At the beginning, only Stream 1 is active. Although its total share is only a fraction of the total bandwidth (its share is 40%), because no other tasks are active it receives the total disk bandwidth. At time 60 Stream 2 becomes active. There is still excess bandwidth because Class A's share is only 50%. The excess bandwidth is distributed to Streams 1 and 2. From time 60 to time 120, they receive their fair-share plus the excess bandwidth. Excess bandwidth is allocated on first-come first-serve basis, accounting for the observed variation in actual rate (this variation is a topic for future investigation). At time 120 Stream 3 begins and, now fully loaded, the nominal reservations are enforced.

5. Conclusion

We are developing flexible integrated real-time solutions based on real-time scheduling algorithms. This is a first step in providing better real-time support in general-purpose systems, better general-purpose support in real-time systems, and, ultimately, a general framework to fully integrate applications of different types of processing constraints.

The longer-term goals of this project include combining these solutions in a single system, and developing complete solutions for other resources including, network I/O, memory, cache, and others. A key challenge will be the development of a framework that supports the combined management of all of the system resources so that, for example, failure to meet a soft deadline in one resource will not negate the benefit of meeting the same deadline with another resource. Our ultimate goal is the complete merging

of real-time scheduling techniques with general-purpose systems, supporting a range of timing constraints from hard real-time to background best-effort batch processing.

Our research is funded by a DOE HPCS Fellowship and the Intel Corporation.

References

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

[2] S. Banachowski, T. Bisson, and S. A. Brandt. Integrating best-effort scheduling into a real-time system. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004)*, Dec. 2004.

[3] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

[4] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, June 1999.

[5] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the 21th IEEE Real-Time Systems Symposium (RTSS 2000)*, pages 295–304, Dec. 2000.

[6] G. M. Candea and M. B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 157–166, Aug. 1998.

[7] S. Childs. Portable and adaptive specification of disk bandwidth quality of service. In *Proceedings of the 9th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 1999.

[8] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.

[9] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on general-purpose operating systems. In *Proceedings of the 5rd Symposium on Operating Systems Design and Implementation (OSDI'02)*, Dec. 2002.

[10] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, Oct. 1996.

[11] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the Fifth Parallel and Real-time Systems (PART98)*, 1999.

[12] The Institute of Electrical and Electronics Engineers. *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application Programming Interface (API)-Amendment 1: Realtime Extension [C Language]*, Std1003.1b-1993 edition, 1994.

[13] C. Lin and S. A. Brandt. Efficient soft real-time processing in an integrated system. In *Work in Progress Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS WIP 2004)*, Lisbon, Portugal, Dec. 2004.

[14] The Linux kernel archives. http://www.kernel.org, Jan. 2004. A web site with the latest Linux kernel and information.

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

[16] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A new reclaiming algorithm for server-based real-time systems. In *10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS04)*, May 2004.

[17] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4UNIX scheduler unacceptable for multimedia applications. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, 1993.

[18] J. Regehr and J. A. Stankovic. Augmented CPU reservations: Towards predictable execution on general-purpose operating systems. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS01)*, pages 141–148, May 2001.

[19] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, Dec. 2001. IEEE.

[20] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55. ACM Press, 1998.

[21] R. Wijayaratne and A. L. Reddy. Integrated QOS management for disk I/O. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 487–492, June 1999.

[22] V. Yodaiken and M. Barabanov. Real-time Linux. In *Proceedings of Linux Applications Development and Deployment Conference (USELINUX)*, Jan. 1997.

[23] W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS01)*, May 2001.

A unified framework for managing different resources with QoS guarantees

Luigi Palopoli, Tommaso Cucinotta,
Antonio Mancina, Luca Marzario,
Paolo Valente
Scuola Superiore S.Anna, Pisa, Italy
{palopoli, cucinotta, mancina, marzario, valente}@sssup.it

Abstract

This paper describes ongoing research activities aimed at developing a unified framework by which a general purpose operative system is able to support applications with Quality of Service requirements. The work is focused on a feedback based dynamic management of the resources required by an application, where each resource is handled through the use of a Resource Reservation paradigm. This allows applications to share the access to a resource by specifying the fraction of usage, where such fractions are dynamically adapted by the system on the basis of observations made on the hosted activities.

Research in this area comprises development of both a theoretical framework for modelling applications and analysing the impact of control theoretic feedback strategies to the QoS experienced by applications, and a prototype implementation of an architecture with the ability of providing the needed functionality on a Linux Operative System.

1. Introduction

In the past few years, the use of computer based technologies has become pervasive in a wide class of applications ranging from communication (e.g., Voice-over-IP) to entertainment (e.g., video games). For many of these applications, the Quality of Service perceived by users is tightly linked with the temporal behaviour. For instance, for a video-on-demand streaming application, the frame-rate plays a role of paramount importance, whereas for a video-conference application the delay and the jitter are also crucial parameters. Recent research activities in this eld have covered the following topics: 1) scheduling mechanisms inside the operating system such that it is possible to allocate speci ed fractions of resource to the competing tasks [11, 9, 7], 2) feedback based management strategies to cope with scarcely known or time-varying execution requirements of the tasks [5, 3, 10], 3) architectural solutions for the Operative Systems and middleware to support the technologies described above.

The results cited above do not offer a conclusive response to the problem of QoS management. Indeed, they miss a fundamental point: soft real-time tasks typically use resources of different types at the same time. Therefore we need a holistic approach, where scheduling requests of heterogeneous types, such as CPU, disk and/or network requests, are streamlined and managed in a holistic and coordinated way. As an example of such an application, consider a MPEG decoder. Its main routine could be the following:

```
for(;;) {
    read_frame();
    decode_frame();
}
```

From this sketch, it is possible to see that it makes little sense to provide the correct amount of CPU for the decoding part, if the application is delayed in its access to the disk or network. In general, the need for different types of resources may generate undesired effects of unexpected harshness, unless the interaction of different allocation mechanisms is not adequately accounted for. The problem becomes even more complex if adaptivity is required (i.e., feedback based allocation of resource in response to temporal variations of the workload).

Moreover, an evident shortcoming of reservation based scheduling mechanisms proposed so far is that they are strongly biased toward a particular class of applications, i.e., soft real-time systems and particularly continuous media (CM). From a wider perspective it is possible to identify three different types of applications that populate general purpose systems:

Interactive: applications like console shells, text editors and GUI-based applications;

Soft Real Time: o ws of data like audio and video streams;

Batch: applications which ask for an intensive use of a resource but which do not have strict timing requirements (they are much less sensitive to delays than soft real-time ones), i.e. le-transfer applications.

Notably, the above is not a clear-cut division: indeed, applications may even change their nature, according to the inputs provided by the user and by the environment. As an example, consider a graphic application waiting for a user choice, and then starting a particular operation such as a spell-checker or a compiler. In this case an application switches from an interactive to a batch behaviour.

In our view, the operating system should be able to guarantee a speci ed QoS level - albeit of a different type - to all of these applications. For instance, interactive applications are sporadic in the true meaning of the adjective (i.e., they do not occur with speci ed frequencies) and have typically little requirements; still when they do occur the QoS experienced by the user is tightly connected with the shortness of the response time (i.e., the user expects to see a character on the screen right away, when he/she presses a key). Standard heuristics to solve this problem are commonplace in modern operating systems, but they are not able to offer any guarantee whatsoever when the system is heavily loaded. We will discuss in this paper a possible strategy to combine temporal guarantees with responsiveness for interactive applications, which is also sensitive to applications changing their behaviour as in the example above.

As far as soft real-time (e.g., Continuous Media) applications are concerned, in our prior work [5] we have shown a QoS management system built on the top of a resource reservation mechanism. In that case, we used a uid model of the system's evolution (i.e., of the dynamic equations of the "plant" to be controlled). This model can be approximated to a desired precision by a resource reservation scheduler, paying the price of a potentially considerable overhead. In this paper, we take a more pragmatic view by building a model of the system that is closer to the implementation. The granularity used by the QoS manager is necessarily coarser but most of the attractive properties of the control schemes shown in [5] can be recovered to a satisfactory extent. Another contribution of this paper is the presentation of a middleware where these techniques are implemented. Although based on the Linux kernel, the architecture of the middleware relies on a kernel abstraction layer, which allows for easy porting to other operating systems (e.g., MAC-OSx and FreeBSD). Finally, we will show the lines of an ongoing research aimed at extending the approach to the case of applications modelled as pipelines of stages using different resources. This extension can be done both for the theoretical aspects (i.e., control design) and for the middleware.

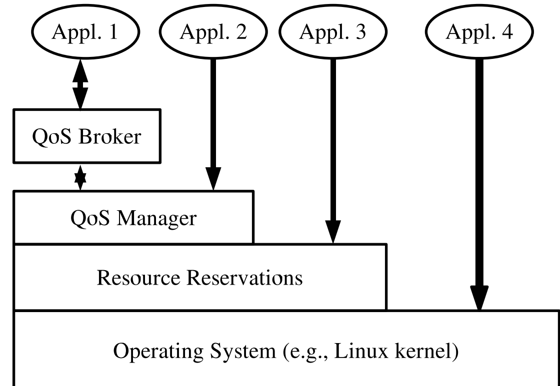


Figure 1. Main conceptual blocks

2. Overview

Our framework is described in Figure 1 and it comprised of the following elements:

1. Resource Reservation (RR) mechanisms
2. QoS manager
3. QoS Contract Broker

The Resource Reservation layer is the fundamental building block enabling us to control the allocation of CPU and other resources to the applications. In our idea, it should not require any kernel-speci c functionality but, rather, be conceived as a sort of "plug-in" installable into any kernel by means of minimally invasive modi cations. Indeed, referring to the Linux implementation, the RR is implemented in a small patch and in a pluggable module. Applications scheduled according to the resource reservation mechanism can coexist with other applications scheduled according to the standard policy used in the "host" kernel, although the latter do not receive any type of temporal guarantee. With respect to the standard view of resource reservations, our approach allows us to dynamically change the resource bandwidth alloted to each task, thus acting as a sort of "actuation" mechanism inside the kernel. Moreover, we explicitly address the problem of guaranteeing responsiveness to the interactive applications.

The QoS manager purpose is twofold. On the one hand, the QoS Manager performs an admission control: when an application requires admission, a negotiation phase with the broker determines the QoS level, if any, which can be guaranteed. The second purpose of the QoS manager is to dy-

namically tune the bandwidth reserved to each task for every resource it uses (in the following sections we will analyse this aspect thoroughly). The purpose of this activity is to maintain as much as possible the previously negotiated QoS levels. In case an overload occurs, the manager can decide to degrade the QoS level of any task by a call-back noti cation, urging the application to reduce its bandwidth utilisation.

At the QoS management level the way for measuring QoS is quite rough: the QoS is simply related to the delay with which a task accomplishes its function related to the expected termination. Moreover, the "currency" used for negotiating the admission of the application into the system is the resource utilisation required on the average by the application. This information is hardly available at the application level (a programmer typically does not make any assumption on the platform that will be used for the application), unless prior executions on the same platform have been pro led and logged. Indeed, the QoS level required by the application is associated to a set of parameters (e.g., resolution, latency, delays), whose mapping to the resource utilisation is not obvious. For this reason, we envision the use of a QoS contract broker, which is able to translate the application level requirements into kernel level parameters governing resource allocations. For example, a video player could ask for a 25 fps frame rate with a given resolution (level 1), or for the same frame rate with a lower resolution (level 2). The QoS contract broker is in charge of translating these application level parameters into resource demands (understood by the QoS Manager) and, likewise, to supervise the callback mechanism so that a bandwidth reduction required by the QoS manager is translated into the appropriate application level working mode. To the current state this component is still in the early design phase and, for this reason, it will not be referred to in this paper.

3. Resource Reservations

The problem coped with in this paper is how to effectively schedule time-shared resource in soft real-time systems. This task is to be regarded as a challenging one. Indeed, a dif cult match has to be found between two different requirements: 1) ensuring a correct timing behaviour to the applications, 2) making an ef cient use of the shared resource. Our strategy is based on the concept of *adaptive reservations*, which combine state-of-the-art scheduling solutions for soft real-time systems (resource reservations) and a feedback-based adaptation strategy.

In this section, we will formally introduce the problem of scheduling time-shared resources in a real-time system and present our proposed scheduling solution. Then, we will state the feedback control problem.

3.1. The real-time tasking model

A real-time task $\tau^{(i)}$ is a stream of jobs, or task instances. Each job J^i_j is characterised by an arrival time $r^{(i)}_j$, a resource utilisation time $c^{(i)}_j$, and a deadline $d^{(i)}_j$.

When a job arrives at time $r^{(i)}_j$, the task is eligible for the allotment of temporal units of the resource (e.g. CPU) from the scheduler. After the task receives $c^{(i)}_j$ time units the job nishes at a *finishing time* labelled as $f^{(i)}_j$. We will consider *preemptive* scheduling algorithms. In our model job $J^{(i)}_j$ cannot receive resource units before $f^{(i)}_{j-1}$, i.e. the activation of a job is deferred until the previous ones from the same task have been completed.

Furthermore, we will restrict to *periodic* tasks: task $\tau^{(i)}$ generates a job at integer multiples of a xed period $T^{(i)}$ and the deadline of a job is equal to the next periodic activation: $r^{(i)}_j = d^{(i)}_{j-1} = kT^{(i)}$.

For example, a typical CPU-intensive task code structure is the following:

```
task () {
    initialisation;
    while(condition) {
        computation
        wait_for_next_instance();
    }
}
```

After initialisation, the task enters a loop where it performs the computation and then waits for the next activation. We assume that the task is activated by a periodic timer that, upon expiration, send a signal or a message that wakes up the task. Each instance of the loop is a job; the nishing time of the job is the time it invokes the `wait_for_next_instance()`.

As said, we assume that the relative deadlines of the tasks are equal to the tasks' periods. Therefore, if the task is soft real-time, a job is allowed to complete after the next activation. In this case, activations are buffered.

3.2. Temporal protection and Resource Reservations

When dealing with a multiprogrammed environment a major emphasis is usually put on the issue of *protection*. Roughly speaking, protection means that the effects of a problem in the execution of a task are con ned to the task itself. The most important feature of real-time applications is that their correctness is related to their ability of meeting real-time constraints. Therefore, to cope with execution time variations and unpredictability in this type of systems, the notion of *temporal protection* plays a role of

paramount importance alongside of memory protection. Restricting for the sake of simplicity to the case of independent tasks $\tau_1, \ldots, \tau_n$, a formal de nition of this concept can be as follows: *a scheduling algorithm is said to guarantee temporal protection if the ability for each task $\tau^{(i)}$ to meet its timing constraints only depends on the evolution of the resource utilisation time and inter-arrival times, and not on the other tasks' workload.*

There are many scheduling algorithms known in literature that exhibit this property; examples are Proportional Share schedulers, and Resource Reservations.

In this paper, we focus on Resource Reservations, originally proposed in [9]. Reservations have been successfully applied to a variety of different resources (including CPU, disk bandwidth, network bandwidth and so forth) [9]. Therefore, even though we focus on CPU scheduling in the following, the presented approach may be applied on different kind of resources as well.

A *resource reservation* RSV_i for a task $\tau^{(i)}$ is described by a pair $(Q^{(i)}, P^{(i)})$, with the meaning that the task is *reserved* the resource for a maximum time $Q^{(i)}$ every $P^{(i)}$ units of time. $Q^{(i)}$ is the reservation *maximum budget* and $P^{(i)}$ is the reservation *period*. In general, the task $\tau^{(i)}$ needs not to be periodic; also, in case of periodic tasks, $P^{(i)}$ can be different from the task's period $T^{(i)}$.

The basic idea behind resource reservations is that every task has a limitation $Q^{(i)}$ on the amount of computation it can perform every period $P^{(i)}$. For example, if we want to keep under control a task $\tau^{(i)}$ with variable computation time, we can assign it a reservation $(Q^{(i)}, P^{(i)})$ with $P^{(i)} = T^{(i)}$ and $Q^{(i)}$ equal to some value above the average computation time of the task. The resource reservation *guarantees* that the task will never request more than $Q^{(i)}$ units of time every $P^{(i)}$. If a job $J_j^{(i)}$ needs to execute for more than $Q^{(i)}$ time units, some action must be taken to *enforce* the limitation on the execution time. There are many different algorithms that implement the resource reservation approach. Our framework is based on a derivation of the well known Constant Bandwidth Server (CBS) [1]. The interested reader is referred to the cited paper for details. For our purposes, it is suf cient to say that the CBS manipulates the deadlines to be used in an Earliest Deadline First Scheduler. The idea is very simple: at the beginning of a server period, a task is given a scheduling deadline equal to the end of the period. When a task becomes eligible for execution it starts to execute consuming its budget. This algorithm can be proved to be a possible implementation scheme for resource reservations.

As an important remark, with this scheme a reservation $RSV_i = (Q_i, P_i)$ behaves like a periodic task with worst-case computation time equal to Q_i and period equal to P_i. Therefore, to ensure the schedulability according to the selected scheduling algorithm, the following inequality has to be respected:

$$\sum^{(i)} \frac{Q^{(i)}}{P^{(i)}} \leq U^{lub} \qquad (1)$$

with $U^{lub} = 1$ for the CPU. If (1) holds, every task $\tau^{(i)}$ attached to a reservation $(Q^{(i)}, P^{(i)})$ is guaranteed to receive its reserved amount of time (i.e., $Q^{(i)}$ time units over $P^{(i)}$). The ratio $B^{(i)} = \frac{Q^{(i)}}{P^{(i)}}$ is said *reserved bandwidth* and it can intuitively be thought of as the fraction of the CPU time reserved to the task.

3.3. Dealing with CM: Adaptive reservations

Resource reservations cannot be regarded as a conclusive solution to the problem of guaranteeing a correct temporal behaviour to a set of soft real-time tasks. In particular, a non-trivial problem still holds: how should the $(Q^{(i)}, P^{(i)})$ pair be dimensioned? In presence of scarcely known or and/or time-varying execution times a static partitioning of the CPU time (based on x ed choice for $(Q^{(i)}, P^{(i)})$) may lead to infeasible or in e xible choices. To address this problem, a feedback based mechanism can be used to self-tune the scheduling parameters and to dynamically recon gure them in presence of changes in the workload.

More speci cally , such a feedback mechanism can be constructed based on:

- an *actuator*, which permits to apply the feedback action to change the system behaviour;
- an *observed value*, used as input to the feedback mechanism;
- a *feedback function*, used to compute the new actuator's value, based on the observed value.

Since the feedback strategy is applied to a reservation based scheduler, the actuator is the amount of reserved time $Q^{(i)}$. In this case $Q^{(i)}$ and $B^{(i)}$ are not constant, but can change for each job: therefore, they will be indicated as $Q_j^{(i)}$ and $B_j^{(i)}$. We call the resulting abstraction an *Adaptive Reservation* [2]. To respect the consistency of the resource reservation algorithm, such changes cannot be done abruptly and attention must be paid to the current state of the servers that change their bandwidth. However important, we will not deal with this issue in this paper for evident reasons of space.

Since we aim at applying feedback control to a CPU scheduler, the observed value can be some QoS metric related to the tasks. The ideal goal of an adaptive reservation could be to schedule $\tau^{(i)}$ so that $\forall_j, f_j^{(i)} = d_j^{(i)}$. Indeed, in this way, not only is it guaranteed that the task progresses respecting its timing constraint but also the quantity of CPU alloted to the task is exactly the one the task needs. Therefore, the deviation, $f_j^{(i)} - d_j^{(i)}$ seems to be a

reasonable choice as an observed value, to be used as a QoS metric. When this quantity is positive (and this is allowed to occur in a soft real-time system), we need to increase the amount of CPU reserved to the task. When it is negative, then it means that the task received CPU time in excess and we may want to decrease it. Unfortunately, due to the de nition of a resource reservation given above, *it is impossible to control the exact time instant when a job finishes*. However, it is possible to control the reservation period inside which the job will nish. Assuming that $(h^i_j - 1)P_i \leq f^{(i)}_j \leq h^i_j P_i$ (i.e., the job terminates within the $h_{i,j} - th$ execution of reservation RSV_i and that the task period $T_i = n_i P_i$, we can de ne the scheduling error as the difference $(h^i_j - jn_i)P_i$

The *feedback function* $f()$ that is used to compute the new actuator value based on the observed value determines the closed-loop system behaviour, and must be carefully designed to attain the desired design goals. We will talk in some detail of this issue in the next section.

The System Dynamic Model A considerable advantage of the choice of the observed value as provided above is that it is possible to construct an accurate mathematical model of the system dynamic evolution. This model can be leveraged in the design of a feedback function $f()$.

Thanks to the temporal isolation property it is possible to describe the evolution of the scheduling error by the following equation:

$$\varepsilon_{k+1} = \max\{0, \varepsilon_k\} + \left\lceil \frac{c_k}{Q_k} \right\rceil P - T \qquad (2)$$

where ε_k denotes the scheduling error experienced by the $k - th$ job, c_k denotes the computation time experienced by the $k - th$ job and Q_k is the budget allocated to the job. In the equation above dropped the (i) superscript referring to the task for notational convenience. The above is an approximate and simpli ed model that describes the evolution of the scheduling error. For practical purposes, the approximation can be regarded as a satisfactory one.

The case of multiple resources The model described above can easily be generalised to the case of *flows* managed by a pipeline of tasks. Each stage performs a partial processing and stores the results into an intermediate buffer (for instance, a stage could do the fetching of a packet, a stage could decode it and a stage could visualise the movie on the screen). Clearly, at each stage the task is periodically activated but its computation might be delayed if the data processed in the previous stage is not yet available. As far as the actuation variable is concerned, in this case we have a vector of budgets to work with (one for each stage of the pipeline). The observed variables can still be the scheduling errors experienced on each stage (assuming for each stage a deadline equal to the period). The extension of the model in Equation 2 is the following:

$$\varepsilon^r_{k+1} = \max\{0, \varepsilon^r_k, \varepsilon^{r-1}_{k+1}\} + \left\lceil \frac{c^r_k}{Q^r_k} \right\rceil P^r - T, \qquad (3)$$

where the superscript r denotes the stage of the pipeline. This is a nonlinear model with a diagonal structure. This model is developed assuming that at each stage of the pipeline the task makes a prevalent utilisation of one type of resource, although it is not required that the resource used at each stage be of the same type (for instance Q^1_k could refer to the disk and Q^2_k to the CPU). Moreover we assume the presence of in nite buffers between the different stages.

3.4. Dealing with interactive applications

This mechanism has been shown to perform very well on continuous media application. However, as we said above, this is not necessarily the case for interactive applications. Consider as an example a text editor waiting for the user input; assume also that it is running in a lightly loaded system. If the application then starts a spell checker, a classic CBS strategy would grant the whole bandwidth to the reclaiming task and this would eventually end up with a deadline postponed far away, thus strongly reducing the application's priority. If the application becomes interactive once again (waiting for some other inputs), the user may experience a very low responsiveness due to low priority of the task.

The inde nite postponement of the deadline in response to a set of events like the one suggested earlier breaches the correctness of the implementation of a resource reservation scheme by the CBS and it has been solved in a ne tuning of the algorithm known as IRIS [6]. Unfortunately, the problem of lack of responsiveness still holds, depending on the choice of period lengths. Indeed,:

1. if we choose a long period P_i tasks' interference is reduced, but responsiveness can be very low (the Q_i budget can be potentially given only at the end of the period still respecting the resource reservation idea);
2. if we choose a short P_i, we get a better responsiveness paying the cost of a greater overhead

To cope with this problem, we propose and adaptive mechanism that operates on P_i. When the bandwidth required by the application grows, P_i is lowered thus allowing a fast response if the task really behaves in an "interactive" fashion. However, in the subsequent server activations P_i is increased in order to reduce the overhead if the task mutates into a batch application. The possibility of using also P_i to trade-off responsiveness against overhead discloses interesting opportunities, which are largely unexplored.

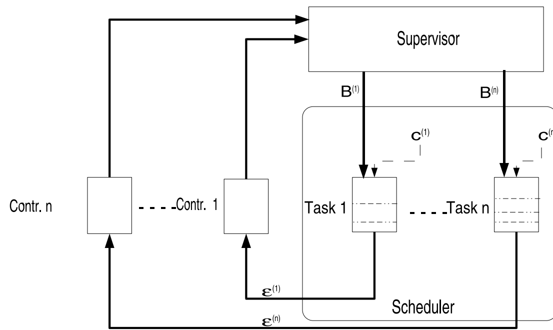


Figure 2. **Decentralised control scheme**

4. QoS Manager

In this section, we show a conceptual architecture for a feedback based management of the QoS experienced by the different tasks. In our model, the information on the QoS can be measured using the scheduling error $\varepsilon^{(i)}$. As shown earlier, when task $\tau^{(i)}$ is served by a reservation pair $(Q_k^{(i)}, P^{(i)})$, it generates a discrete event subsystem. The dynamic evolution of the subsystem can be succinctly described by Equation 2. In this model, $\varepsilon_k^{(i)}$ plays the role of state variable, $c^{(i)}$ is an exogenous disturbance and $q_k^{(i)}$ can be used as a command variable to steer the evolution of the system.

The subsystems related to the different tasks are very loosely coupled. Indeed, owing to the temporal isolation property, the evolution of task $\tau^{(i)}$ depends only on its own parameters and it is asynchronous from the one of the other tasks. Moreover, since we take measurements upon the termination of each job, a system-wide notion of “sampling” is entirely missing. The only global constraint is the one expressed by the consistency relation in Equation (1), which the system must never violate lest the temporal properties of the RSV mechanism be severely jeopardised.

The considerations above almost naturally dictate the control scheme in Figure 2, which is referred to as “decentralised” in the control literature. Each task is attached a dedicated controller, called “task controller”, that uses only the information collected from the task. Roughly speaking, a task controller is responsible for maintaining the QoS provided by the task in speci ed bounds with the minimum impact on CPU utilisation. Still, the total bandwidth request from the different task controllers is allowed to exceed the bound in Equation (1). This situation is de ned *overload* and it is the indirect outcome of concurrently large requests of computation from different tasks. In presence of an overload, a component called supervisor is used to reset the bandwidth allocated to the different tasks within appropriate levels respecting Equation (1). While an overload occurs, the task controllers are not allowed to work properly, since there may be alterations and/or delays in the application of the budget that they dynamically decide. For this reason, we will assume that overloads are episodic occurrences. This working condition can be attained if: 1) the number of tasks admitted into the system is kept within reasonable bounds, 2) the algorithms used for task controllers are designed to avoid excessive bandwidth requests.

4.1. Task controllers

A task controller is comprised of two components: a feedback controller and a predictor (as shown in Figure 3). An appropriate sensor located inside the RSV scheduler returns at the termination of each job the computation time of the job just concluded and the experienced scheduling error. The former information is used by the predictor to produce an estimation regarding the next computation time c_k.

We have developed both deterministic [8] and stochastic [4] approaches for control design. In the case of deterministic design, the control goal is to keep the evolution of the scheduling error con ned in a small set. To attain this goal, the controller “plays” a game against the uncertainty introduced by the variability in computation time. In other words, the controller tries and counteract the variability of the computation times based on an assessment of its worst case effects. For this reason, the predictor produces an interval in which the next computation time is likely to fall, while the feedback controller decides the new budget based on the measurement of the current scheduling error and on the interval provided by the predictor. Intuitively speaking, the larger the size of the predicted range, the larger the set where it is possible to control the scheduling error.

The same conceptual structure is applicable also to stochastic approaches. In this case, the controller considers c_k as a stochastic process and tries to choose the control action to optimise some probabilistic performance metrics (conditioned to the current measurements of the scheduling error and to its past history). For instance, one possibility is to choose the control value so as to have an expected value of the scheduling error in the next step equal to 0. We called this approach stochastic dead beat (SDB) since it tries to reduce the scheduling error to 0 (in the average sense) in one step. The predictor in its turn works in a stochastic framework. For instance in the case of the SDB controller, the predictor produces a guess of the expected value of the next value of c_k conditioned to the past story of the process [4].

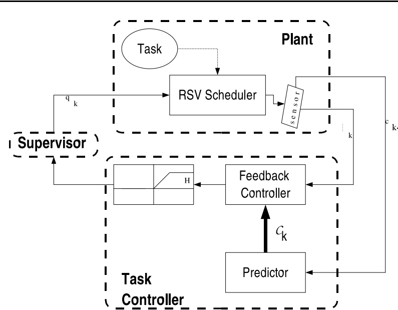


Figure 3. Block diagram of a task controller

4.2. Supervisor

The supervisor component has the following design goals:

1. if all task controllers have bandwidth requests that do not exceed the bound in Equation 1, and are compatible with security limits imposed by the system administrator, then the supervisor has to be “transparent” (i.e. it simply forwards to the scheduler the bandwidth requests coming from the task controllers)
2. in case of overload, the supervisor has to ensure to every task that the deviation from its required resource utilisation will not be permanent. For instance, consider the case of a deterministic control policy aiming at con ning the system evolution into a set $\mathcal{I}$. In presence of overload the scheduling error may be taken afar form $\mathcal{I}$. This requirement amounts to saying that *eventually* it will return into $\mathcal{I}$.

The second design goal can be translated into guaranteeing a minimum bandwidth $b^{(i)}_{min}$ for each task, which is at least equal to its average computation requirements. Taking a pragmatic approach, we will require the enforcement of this requirement, only to the tasks for which such an estimation is available (choosing $b^{(i)}_{min} = 0$ for the remaining ones). The selection of a policy for managing in presence of overloads the bandwidth exceeding the minimum guaranteed to each task is a free design parameter for the supervisor. We have studied different alternatives (based on a prioritisation of tasks, on a weighted compression of the required bandwidths or on both), but we will not detail them here for reasons of space. The reader is referred to [5] for further details.

4.3. Possible extensions to multiple resources

The scheme proposed above can be extended to multimedia streams managed by a pipeline of tasks. Indeed, as discussed in Section (3.3), we can introduce a scheduling error ε^r for the $r-th$ stage of the pipeline and the evolution of the pipeline is describe by Equation (3). An important feature of this model is that the scheduling error at one stage only depends on the scheduling error experienced by the stage immediately before. Thereby, it is possible to operate with a dedicated controller at each stage of the pipeline. This controller takes its decision based on: 1) the scheduling error experienced by the previous job at the current stage of the pipeline ε^r_k, 2) the scheduling error ε^{r-1}_k experienced at the current job by the pipeline stage immediately before the one in consideration, 3) the prediction of the resource workload c^r_k required for the current job. In this framework, the extension of the deterministic control approaches cited earlier is straightforward. But, we are currently evaluating the possibility of different control strategies, which, for each stage, take their decision based on a global information.

5. Implementation Architecture

In this section we describe a software architecture aimed at proving the technological feasibility of the theoretical approaches described so far. The architecture is built on the top of the GNU/Linux OS (Kernel Version 2.4.27). The choice of Linux was motivated by its growing popularity and by the availability of the source code and of a large documentation on the kernel. A considerable advantage of Linux, is also its modular structure that allows one to extend the kernel by inserting a module. However, this choice is also to be considered as incidental: most of the architecture proposed here can be ported with a limited effort to other Operative Systems. More generally, our purpose was to prove that the theoretical machinery described in the previous sections can be introduced into a general purpose operating system, with a limited impact on the kernel structure and on the introduced overheads.

5.1. Design Goals and Architecture overview

The design of the system was carried out pursuing the following goals:

Portability: the link between the proposed architecture and the adoption of a speci c kernel platform (Linux kernel 2.4.27) is shallow. To achieve this goal, we designed a layered structure where kernel dependent code is con ned inside the lowermost level. Moreover, the changes made on the kernel are minimal and the communication between the different components of the architecture (which run partly at user and partly at kernel level) uses virtual devices, which are commonplace in operating systems of Posix class.

Backward compatibility: we did not change the API of the Linux kernel. Therefore, preexisting applications can run without modi cations. Moreover, tasks that do not

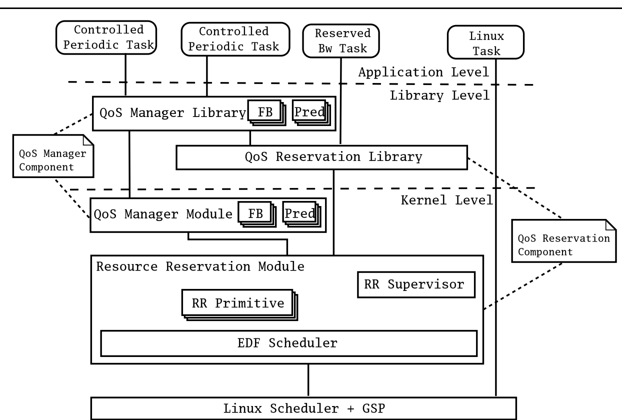


Figure 4. System Architecture

require QoS services are taken care of by the Linux scheduler.

Flexibility: our architecture allows one to easily introduce new control and prediction algorithms. These algorithms can be run either in user or in kernel space. In the former case, it is possible to use oating point mathematics and third parties math libraries (which is very useful during the prototyping phase). Another possibility potentially offered by user space implementation is to provide speci c predictors along with the different applications (whose introduction into the kernel might be untrusted). On the other hand, kernel space implementation is certainly more ef cient and preferable when the algorithms are well-tested and reliable (as is the case of the algorithms proposed in this paper).

Ef cienc y: the overhead introduced by QoS management mechanisms is acceptable. Moreover, applications that do not use QoS management functionalities experiment a negligible overhead.

Security: the possibility of changing the bandwidth reserved to the different applications makes for denial of service attacks; therefore the bandwidth allocation mechanism has to be compatible with a "maximum" CPU bandwidth de ned on a per-user basis (in the same way as disk quotas are).

The proposed architecture is depicted in Figure 4, and it is composed of the following main components:

- the Generic Scheduler Patch (GSP), a small patch to the kernel (276 lines) which allows to extend the Linux scheduler functionality by intercepting scheduling events and executing external code;
- the Resource Reservation Component, composed of a kernel module and of an application library communicating through a Linux virtual device:
  - the Resource Reservation module implements the resource reservation mechanism and the RR supervisor; a set of compile-time con guration options allows one to use different Resource Reservation (RR) primitives, and to customise their exact semantics (e.g. soft or hard reservations);
  - the Resource Reservation library provides an API allowing an application to use resource reservation functions;
- the QoS Manager Component, composed of a kernel module, an application library, and a set of predictor and feedback subcomponents which may be con gured to be compiled either within the library or within the kernel module:
  - the QoS Manager module offers kernel space implementations of some feedback control and prediction algorithms (including the ones shown in this paper);
  - the QoS Manager library provides an API allowing an application to use QoS management functionalities; as far as the control computation is concerned, the library either implements the control loop (if the controller and predictor algorithms are in user-space) or redirects all requests to the QoS Manager kernel module module (in case a kernel-space implementation is required). In the former case, the library communicates with the resource reservation module to take measurements of the scheduling error or to require bandwidth changes (such requests are " ltered" by the RR supervisor).

6. Conclusions and future work

In this paper we stated the general problem of adaptive management of resources for soft real-time applications. We described extensions to our prior model, which focused only on CPU allocation, in the context of applications using multiple resources. We also introduced the architecture that is being developed for supporting these mechanisms on a general purpose OS like Linux, based on an extension of our prior architecture allowing adaptation of the CPU bandwidth for QoS control.

We plan to investigate on the impact of the introduced decentralised control technique on the QoS experienced by soft real-time applications. We also plan to develop centralised control strategies, where decisions are based on the state of the entire pipeline of activities by which a multi-resource application is composed. We expect this approach to be particularly effective whenever applications possess

the ability to adapt their fraction of use of a resource depending on quality parameters, as is commonly the case for multimedia applications.

Finally, even though development is primarily focused on the Linux OS, we plan to maintain portability across multiple systems (e.g. FreeBSD, on which we already have a resource reservation based management of the disk) by means of the kernel abstraction layer.

References

[1] L. Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[2] L. Abeni, L. Palopoli, and G. Buttazzo. On adaptive control techniques in real-time resource allocation. In *Proceedings of the Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[3] G. T. C. Lu, J. Stankovic and S. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Ppecial issue of RT Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, 23(1/2), September 2002.

[4] T. Cucinotta, L. Palopoli, and L. Marzario. Stochastic feedback-based control of qos in soft real-time systems. In *Proc. of the IEEE 2004 conference on decision and control (CDC04)*, Paradise Island,Bahamas, December 2004.

[5] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L.Abeni. Adptive reservations in a linux environment. In *Proc. of the 10t10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTSA04)*, Toronto, Canada, May 2004.

[6] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: a new reclaiming algorithm for server-based real-time systems. In *Proc. of Real-Time Application Symposium (RTAS 04)*, May 2004.

[7] C. W. Mercer, R. Rajkumar, and H. Tokuda. Applying hard real-time technology to multimedia systems. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.

[8] L. Palopoli, T. Cucinotta, and A. Bicchi. Quality of service control in soft real-time applications. In *Proc. of the IEEE 2003 conference on decision and control (CDC02)*, Maui, Hawai, USA, December 2003.

[9] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[10] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third usenix-osdi*. pub-usenix, feb 1999.

[11] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

Adding new features to the Open Ravenscar Kernel *

Santiago Urueña José A. Pulido Juan Zamorano Juan A. de la Puente
Universidad Politécnica de Madrid, Spain

Abstract

ORK is a specialized real-time kernel for high-integrity embedded systems based on the Ada Ravenscar profile. The paper is focused on the evolving requirements for a new generation of Ravenscar kernels, coming both from the evolution of the Ada language and the needs of future aerospace systems. An assessment of the changes is done, and a set of new features to be included in the next ORK version is selected. The new features are organized as an upward-compatible set of kernel configurations, which can be used in different kinds of systems.

1. Introduction

The Open Ravenscar real-time Kernel (ORK) [11, 13] is a small, reliable kernel for high-integrity embedded real-time systems which supports a simple computational model which can be analysed for temporal correctness using rate-monotonic and response-time analysis techniques [15, 6]. The ORK computational model is defined by the Ada Ravenscar profile [7, 8], and supports systems consisting of a static set of periodic and sporadic tasks communicating by means of protected shared objects. Embedded real-time applications can be built on top of ORK in Ravenscar Ada or in C, using the GNAT[1] compilation system.

The current version of ORK[2] has shown its value for high-integrity embedded real-time systems, after having been used in some pilot applications with very positive results (see e.g. [25]). It has been recently adopted as a basis for a professional software development system for mission-critical spacecraft embedded systems [21]. However, new requirements for supporting a wider class of embedded systems have arisen, and a proposal to update the Ada language standard, including the Ravenscar profile itself and a number of interesting new features for real-time systems, is expected to be approved soon. These changes in the application domain and in the Ada technology make it necessary for the kernel to be adapted to the challenges that the development of high-integrity real-time systems will raise in the near future.

The aim of this paper is to make assessment of the required modifications to ORK, and a selection of the new features that the upcoming versions of the kernel must have, in order to be used in future high-integrity embedded systems.

The paper is organized as follows: a short summary of the ORK functions and architecture is first made in section 2. New requirements coming from the upcoming Ada 2006 standard are then analysed in section 3. Other changes and possible alternative computation models coming from the evolution of high-integrity embedded systems are discussed in section 4, taking the aerospace domain as a basis. Section 5 contains a proposal for a configurable enhanced version of ORK addressed at future aerospace embedded systems. Finally, some conclusions and proposals for future work are presented.

2. ORK functionality and structure

ORK supports restricted Ada tasking as defined by the Ada Ravenscar profile. This results in the following set of functionalities:

- Thread creation only at system start time. Threads cannot be terminated.
- Preemptive priority thread scheduling, with FIFO dispatching for threads with the same priority. Priorities are fixed, and can only be changed as part of the implementation of the ceiling locking access protocol.
- Thread synchronization is restricted to mutexes and a simple form of condition variables. In order to properly support Ravenscar protected objects, mutexes are static and are locked according to the ceiling locking protocol [2, 22]. At most one condition variable can be used with a mutex, and at most one thread can be waiting on a condition variable.

*Work supported by MEC, project TRECOM (TIC2002-04123), and the EC 6FP, project ASSERT (IST 4033).

[1] `http://www.gnu.org/software/gnat/gnat.html`

[2] Available at `http://www.dit.upm.es/ork`.

- Time keeping and absolute delays implemented with the highest possible accuracy [28].
- Storage management, restricted to linear allocation of stack space for threads at system start time.
- Interrupt handling.

The kernel is organized as a set of Ada packages which implement the above functionality (figure 1). This architecture has been designed in order to ensure portability and configurability, thus easing its maintenance and evolution.

3. Ada 2006 real-time features

3.1. The Ada 2006 revision process

The first motivation for the evolution of ORK comes from the changes in the Ada language itself. After a good amount of work by the responsible standardization bodies and the Ada community, an amendment to the language has been developed which will expectedly result in a new standard to be issued at the beginning of the next year.[3] The amendment is aimed at improving the language in several areas, one of which is real-time systems.

The most visible enhancement is the addition of the Ravenscar profile itself to the standard [1, D13.1], thus acknowledging its relevance for the high-integrity systems domain. The main implication of this change is that all Ada compiler builders who want to offer the Ravenscar profile have to do it in a standard way, so that Ravenscar programs are portable at the compiler level. However, existing real-time kernels that support the Ravenscar profile need not be modified, as the profile definition has been stable for some years now.

In addition to making the profile part of the standard, there are other changes in the real-time area that have a potential impact on real-time kernels. Some of the changes are outside the Ravenscar definition, but they still have to be analysed in order to find possible conflicts or side effects, and to assess the possible benefits of extending the computation model so that the new features can be used in high-integrity systems. The following paragraphs include a discussion of the most important Ada 2006 changes and their impact on Ravenscar kernels.

3.2. Dynamic ceiling priorities

Dynamic task priorities is an important feature for real-time systems with multiple operating modes, which was already supported in Ada 95. However, there was some inconsistency in that the ceiling priorities of protected objects could not be dynamically changed. Although there are workarounds that enable mode changes to be effected with constant ceilings, they are error-prone and may introduce additional blocking. The proposed Ada amendment solves this inconsistency by enabling ceiling priorities to be changed at run time [1, D5.2].

Dynamic task priorities were excluded from the Ravenscar profile because they may compromise temporal predictability if used improperly [8]. The 2006 amendment sticks by this policy by keeping dynamic ceiling priorities out of the profile. This implies that no modification to the kernel is required by this change. However, if ORK would be extended to support some kind of “extended Ravenscar profile”, which has some supporters in the real-time Ada community, [14], it would be a simple matter to add support for dynamic priorities and ceilings. All that is needed is to provide kernel services for modifying the priority entries in the kernel data structures associated with tasks and protected objects, respectively.

3.3. Execution-time clocks and timers

Execution-time clocks provide a mechanism for measuring the CPU time that a task has spent since it was started [1, D14]. This is indeed a useful feature for high-integrity systems, as it enables execution-time budgets to be monitored and overloads to be detected. It has been shown to be compatible with the Ravenscar restrictions [12], and has been included in the new standard definition of the profile.

The Ada 2006 proposal also includes execution-time timers, which can be used to make a handler procedure to be executed whenever a specified amount of CPU time is consumed by a task [1, D14.1]. This feature enables execution-time overruns to be detected and, when applicable, error recovery to be performed. However, effective use of execution-time timers requires asynchronous transfer of control, a construct which introduces what is usually considered a unacceptable degree of indeterminacy for high-integrity systems [8]. For this reason it has been excluded from the Ravenscar profile.

A related feature is execution-time budgets for groups of tasks [1, D14.2]. The main motivation for this mechanism is to provide support for aperiodic servers [23, 24]. Group budgets have been kept out of the Ravenscar profile for similar reasons as execution-time timers.

The impact of the above changes on ORK is not the same for each of them. Execution-time clocks have to be supported, as they are part of the profile, and can be implemented in a rather straightforward way by keeping track of each task consumed CPU time on every context switch [27].

Execution-time timers and group budgets are not part of the profile, but are clearly of interest for a wider class

[3] See `http://www.ada-auth.org/amendment.html`.

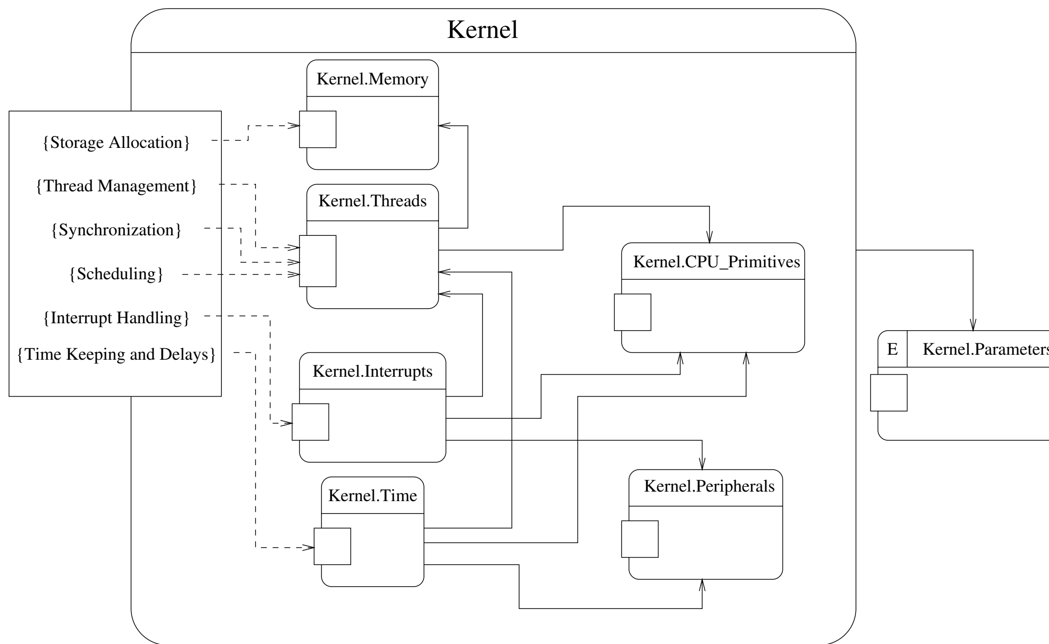


Figure 1. ORK architecture

of real-time systems requiring a strict control on overruns. Their implementation is more complex, as it involves using a hardware timer, which in some architectures has to be shared with the implementations of delay services. A pilot implementation has also shown that these features introduce a significant overhead on context switches and interrupt handling [27].

3.4. Scheduling policies

The current Ada standard [2] defines a preemptive priority scheduling policy, with FIFO dispatching order for tasks at the same priority, and an immediate ceiling priority access protocol for shared data encapsulated in protected objects. The Ravenscar profile mandates this scheduling policy, and forbids dynamic priority changes other than those required by the locking policy, as well as dynamic tasks and protected objects. All these restrictions define a static, analyzable task model [8].

The proposed new standard enlarges the Ada scheduling model by including additional scheduling methods in an upward-compatible way. The primary scheduling mechanism is still priorities, but different dispatching policies may be specified for different priority levels. The new dispatching policies include non-preemptive FIFO, round-robin, and earliest deadline first (EDF). Dispatching policies apply to a band of priority levels, so that different policies, e.g. EDF and preemptive FIFO may be used on the same system [1, D2].

The new scheduling policies are not part of the Ravenscar profile, but non-preemptive and EDF scheduling are clearly of interest for an “extended Ravenscar” class of applications. The impact of adding alternative scheduling policies to the kernel is comparatively high, as it involves a major modification of the thread management package (figure 1). Different dispatching procedures have to be implemented, and the one to be used at scheduling point depends on the scheduling policy which is used at the current priority level. This mechanism has a cost on context switch times which is still to be evaluated.

3.5. Timing events

Timing events enable a low-level mechanism for executing procedures at specified points of time, without using tasks or delay statements [1, D15]. This feature enables efficient implementation of short time-triggered actions, and is thus of interest for real-time embedded systems.

The updated Ravenscar profile definition for the new standard allows timing events, but only at the library level, i.e. the set of timing events must be static. Therefore, the kernel has to be updated in order to support static timing events.

The impact of this change on ORK is high. The approach is to use a hardware timer and an ordered queue of timing events. Delay expirations are a particular case of timing events, and thus share the same timer and event queue. The main problem is with event cancellations, which may

compromise the static, predictable nature of Ravenscar programs.

4. Requirements of future embedded systems

4.1. The ASSERT project

Technological changes in hardware and software are expected to keep the trend towards increasing flexibility and complexity in high integrity embedded systems in the forthcoming years. This trend is exemplified by the ASSERT[4] project, which is aimed at improving the system-and-software development process for critical embedded real-time systems in the Aerospace and Transportation domains. Preliminary requirements capture for two pilot projects in the aerospace domain which has been performed in the framework of ASSERT shows that future real-time embedded systems in this domain will have significant differences in size and complexity within a single product family of related applications. This means among other things that the real-time kernels for these systems will have to be configurable and flexible enough to support the variability of future families of embedded systems.

The most important characteristics that have been identified in this analysis are:

- *Distribution.* Future embedded systems will require distributed execution on a set of possibly heterogeneous computers. Communication must be transparent with respect to the location and architecture of the communicating entities, and at least in some cases it must also be predictable in the temporal domain.

- *Criticality*. Some applications are highly critical, in the sense that their failure may lead to loss of life or mission failure. These applications are usually classified as level A or B according to DOD-178B [20] or a similar standard, and are usually required to undergo a certification process which is also defined in the relevant standard.

- *Partitioning*. The increasing power of microprocessor hardware has led to putting together in the same computer different applications, possibly with different criticality and timing requirements. Partitions are logical spaces for the protected execution of such applications, so that storage space and processor time allocated to one application are not invaded by other applications.

- *Dependability*. In order to ensure the integrity of high-criticality applications, dependability-oriented techniques such as replication and fault containment regions may have to be used in future embedded systems.

The implications and impact of these properties on real-time kernels are analysed in the next paragraphs.

4.2. Distributed execution

Transparent communication and other functionality are appropriately handled by a middleware layer [4]. Indeed, middleware to be used in real-time systems must exhibit a predictable temporal behaviour [5], which can be analysed using appropriate response-time analysis methods [18]. PolyORB [26] is a middleware which can be configured for different so-called *personalities*, including some real-time standards as RT-CORBA and Ada DSA. It has been adopted as one of the building blocks of a generic distributed systems architecture which is to be prototyped in the framework of the ASSERT project (see 5 below).

The impact of distribution features on the real-time kernel is mainly on the lower-level communication layers. Predictable communication requires bounded, analysable message transmission times as a basis [18]. A number of network and communication protocols are available that have the required properties [5], but not all of them are appropriate for the ASSERT application domains. RT-EP [16] and SOIS MTS [9, 19] have been selected for the prototype architecture. The only modification which is required from the kernel in order to support these protocols is the development of a device handler for a LAN chip and the associated RT-EP driver, which are included in the board-support package.

4.3. Partitions

In a partitioned system, computer resources are allocated to a number of partitions, in which different applications run. Each application may in turn have a number of concurrent threads (figure 2). In order to ensure space and time isolation among partitions, appropriate mechanisms have to be used that prevent one application to run into other application memory space or to use processor time beyond a given budget. The same applies to other resources, such as input-output devices and communication links.

There are a number of methods that can be used to enforce inter-partition isolation. Hardware mechanisms (MMU) are customarily used to divide a physical storage space into a number of virtual memory spaces, which can be allocated to different partitions. Time partitioning can be achieved by hierarchical scheduling. A global scheduler distributes processor time among partitions, and a local scheduler determines which thread within the running partition executes at a given time. Timers of different kinds

[4] Automated proof based System and Software Engineering for Real-Time systems, FP6 IST 4033.

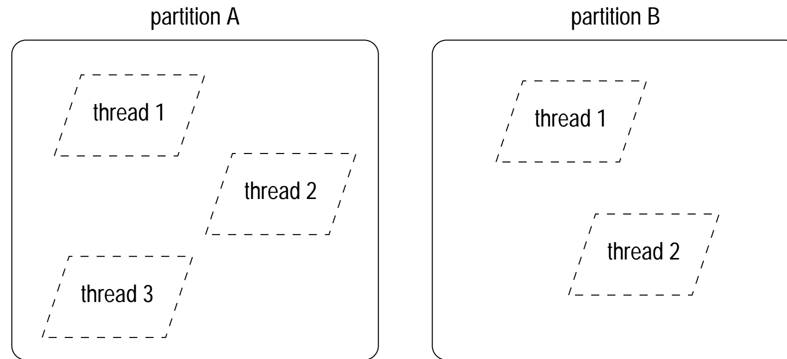


Figure 2. Partitioned system

can be used to detect overruns both at the local and global levels.

The ARINC 653 standard [3] for integrated modular avionics (IMA) is based on such principles. It defines an operating system interface (APEX), with two-level scheduling. Static scheduling is used for partition scheduling, in a similar way as a cyclic executive works, while thread scheduling within partitions is based on fixed-priority preemptive scheduling (FPPS). This approach leads to highly predictable, analysable timing behaviour, at the cost of an inherent lack of flexibility at the partition scheduling level.

A possible alternative is to use FPPS for scheduling partitions. This approach has the advantage of a greater flexibility and better processor utilization, although temporal analysis becomes more complex. Recent work in the framework of the FIRST project [10] provides new techniques for response time analysis of systems with hierarchical scheduling that make FPPS a real alternative to static scheduling for partitioned systems.

Another relevant issue in partitioned systems is inter-partition communication, which in the real-time case must be predictable. Communication mechanisms should be designed in such a way the integrity of the communicating partitions is not compromised.

The impact of partitioning on the kernel is deep. Although hierarchical scheduling can be implemented using priority bands and group budgets as proposed in the Ada amendment, this may not be enough for systems with a high level of criticality. A hierarchical architecture of the kernel reflecting the hierarchy of partitions and threads seems a better approach for building partitioned systems, as proposed in section 5.

4.4. Criticality and dependability

Applications running on a partitioned system may have different criticality levels. Moreover, all the software which is involved in the execution of a high-criticality application must be certified at that criticality level. This means that, in order not to have to certify all the applications to the highest level, applications must be isolated so that a failure in a low-criticality application —which may be acceptable at its level— does not compromise the execution of high-criticality applications. Indeed, partition mechanisms can be used to this purpose, including MMU hardware for spatial isolation and appropriate scheduling methods, together with temporal analysis and overrun detection mechanisms, for temporal isolation. Of course, this approach requires all the software implementing the partition mechanisms to be certified at the highest level, but in turn applications only have to be certified at their respective criticality levels. Under this approach, partitions act as failure confinement regions for the applications running within them.

The main impact that this approach has on the kernel is that it requires all the partition support software to be certified at the highest level. This includes the real-time kernel and other components which are described in section 5 below. An implication of this requirement is that the kernel and related software must be kept simple enough so that is behaviour can be shown to be predictable at all times.

5. A tailorable real-time architecture

5.1. Introduction

Not all the embedded systems of the future will require support for all the above described properties. The need for small, highly-reliable systems based on a single processor board will still exist, and the Ravenscar profile will undoubtedly be a reasonable computation model for this kind of systems. On the other hand, complex multi-partition distributed systems will also be needed, and appropriate computation models providing the required levels of predictability and dependability for such complex systems will be an issue with growing importance. As we have seen, partitions with space and time separation, hierarchical FPPS and

predictable communications and middleware offer solutions based on available technology for these systems.

Our proposal is to develop a tailorable family of real-time kernels that can give support to different kinds of systems. The baseline configuration is a Ravenscar-compliant kernel, similar to the current ORK version plus the compatible Ada 2006 extensions and communication drivers. Some possible extensions include an "extended Ravenscar profile" with dynamic priorities and ceilings, execution-time timers, and task group budgets. Support for alternative scheduling methods, such as EDF and non pre-emptive priorities, as well as priority-band scheduling, is another clear extension to the basic profile.

Supporting partitions and distributed systems is a more complex issue. A hierarchical architecture has been designed in the framework of the ASSERT project as a first step to investigate these issues, and a first prototype has been built with the aim of showing the capabilities of present day technology and learning more about the problems of complex, distributed embedded systems.

5.2. ASSERT middleware prototype architecture

The first prototype of the ASSERT middleware architecture is based on a set of components that provide support for predictable distributed execution of hard real-time systems (figure 3.) The main software components are:

- A real-time kernel that provides support to thread scheduling and other functions. In the current prototype this component is instantiated by ORK.
- A message transfer service (MTS), which is instantiated in the prototype by a SOIS-MTS package.
- A middleware layer providing support for transparent communication between distributed application components. In the prototype this layer is instantiated by PolyORB, tailored to work on SOIS-MTS and providing both RT-CORBA and Ada DSA services to applications.

The prototype works on bare PC boards linked by a dedicated Ethernet link. A sample application is built on top of it, showing a minimal but representative distributed real-time configuration.

5.3. Partitioned architecture

The above prototype architecture does not support partitions. In order to enable multi-partition systems to be built, a proposal for a partitioned architecture has been developed (figure 4), as an extension of the distributed architecture. In order to make the architecture upward compatible, the extended architecture puts all the partition management functions in a separate nano-kernel layer, leaving the upper layers unchanged or even removing it in the computer nodes that have only one partition. This approach has already been explored in other real-time domains with promising results (see e.g. [17]).

The main nano-kernel functions are:

- Memory management and spatial separation among partitions.
- Partition scheduling and temporal separation. Partitions can be scheduled according to a static execution plan, or with fixed priorities following the scheme proposed in FIRST [10]. Temporal separation can be enforced by using execution-time timers at the partition level.
- Inter-partition communication. Message-based communication provides upward compatibility with the middleware layer and location transparency at this layer.
- Virtual device handling for partition input-output.

Detailed design of the partitioned architecture is expected to be developed before the end of this year.

6. Conclusions

The need for evolution of the ORK kernel comes mainly from two sources: The Ada 2006 revision process and the increasing complexity of high-integrity embedded real-time systems. We have analysed the changes that are required, and a basic set of features have been selected to be added to the ones defined in the Ravenscar profile.

Future versions of the kernel will be tailorable, ranging from a basic Ravenscar configuration to a full-fledged kernel that is included in a partitioned, distributed architecture including also other components. A first prototype supporting predictable distribution has already been developed, and an extended prototype also supporting partitions on the same computer will soon be designed in the framework of the ASSERT project.

7 Acknowledgments

The ASSERT middleware prototype has been developed jointly with Laurent Pautet, Jerome Hugues and Khaled Barbaria from ENST, Stuart Fowler and Marek Prochazka from SciSys, and Tullio Vardanega from the University of Padua. The authors have also had many fruitful discussions with these and other members of the ASSERT project about the architectural concepts presented in this paper.

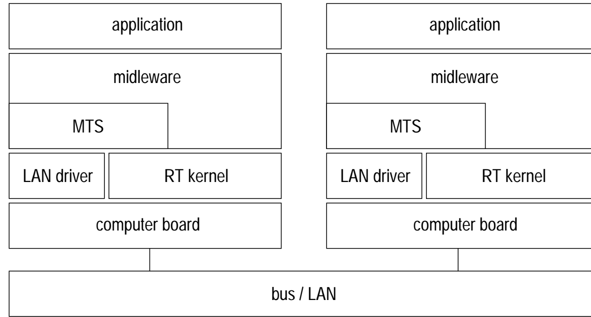


Figure 3. Distributed system architecture

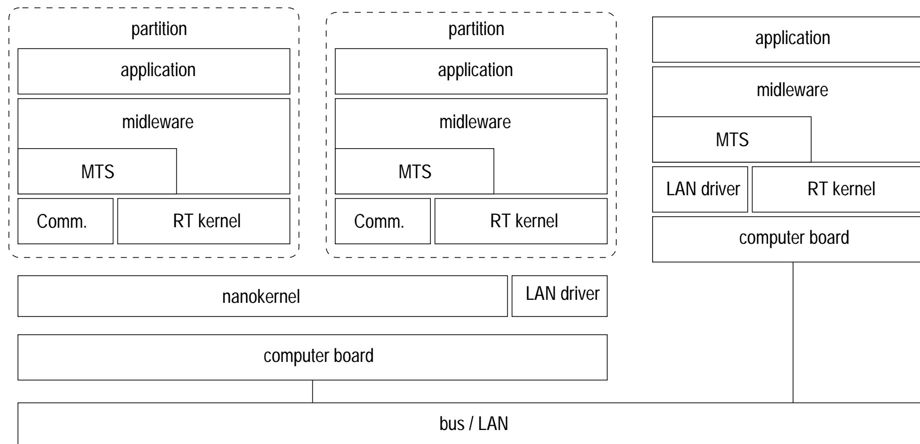


Figure 4. Partitioned system architecture

References

[1] *Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1 (Draft 11)*, 2005. Available on `http://www.adaic.com/standards/rm-amend/html/RM-TTL.html`.

[2] *Consolidated Ada Reference Manual. Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995(E) with Technical Corrigendum 1*, 2000. Available from Springer-Verlag, LNCS no. 2219.

[3] *Avionics Application Software Standard Interface — ARINC Specification 653-1*, October 2003.

[4] P. A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.

[5] B. Bouyssounouse and J. Sifakis, editors. *The ARTIST Roadmap for Research and Development*, volume 3436 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2005. ISBN: 3-540-25107-3.

[6] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S. Son, editor, *Advances in Real-Time Systems*. Prentice-Hall, 1994.

[7] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar profile for high integrity real-time programs. In L. Asplund, editor, *Reliable Software Technologies — Ada-Europe'98*, number 1411 in LNCS. Springer-Verlag, 1998.

[8] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems. Technical Report YCS-2003-348, University of York, 2003.

[9] Consultative Committee for Space Data Standards (CCSDS). *CCSDS Spacecraft On-board Interface Services Green Book – CCSDS 830.0-G-0.4*, December 2004. Draft.

[10] R. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. Technical Report YCS-2005-385, University of York, 2005.

[11] J. A. de la Puente, J. F. Ruiz, and J. Zamorano. An open Ravenscar real-time kernel for GNAT. In H. B. Keller and E. Ploedereder, editors, *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.

[12] J. A. de la Puente and J. Zamorano. Execution-time clocks and Ravenscar kernels. *Ada Letters*, XXIII(4):82–86, December 2003.

[13] J. A. de la Puente, J. Zamorano, J. F. Ruiz, R. Fernández, and R. García. The design and implementation of the Open Ravenscar Kernel. *Ada Letters*, XXI(1), 2001.

[14] B. Dobbing and J. A. de la Puente. Session: Status and future of the Ravenscar profile. *Ada Letters*, XXIII(4):55–57, December 2003. Proceedings of the 12th International Real-Time Ada Workshop (IRTAW 12).

[15] M. H. Klein, T. Ralya, B. Pollack, R. Obenza, and M. González-Harbour. *A Practitioner's Handbook for Real-Time Analysis. Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Boston, 1993.

[16] J. M. Martínez and M. González Harbour. RT-EP: A fixed-priority real time communication protocol over standard ethernet. In T. Vardanega and A. Wellings, editors, *Reliable Software Technologies - Ada-Europe 2005*, volume 3555 of *LNCS*. Springer-Verlag, 2005.

[17] M. Masmano, I. Ripoll, and A. Crespo. An overview of the XtratuM nanokernel. In *OSPERT 2005 — Workshop on Operating System Platforms for Embedded Real-Time Applications*, Palma de Mallorca, July 2005.

[18] J. C. Palencia, J. J. Gutiérrez, and M. González-Harbour. On the schedulability analysis for distributed hard real-time systems. In *Proc 9th Euromicro Workshop on Real-Time Systems*, pages 136–143. IEEE CS Press, June 1997.

[19] C. Plummer and P. Plancke. The spacecraft onboard interfaces, sois, standardisation activity. In *DASIA 2002 - Data Systems in Aerospace*, 2002.

[20] RTCA Inc. *Software Considerations in Airborne Systems and Equipment Certification — RTCA/DO-178B*, 2002.

[21] J. F. Ruiz. GNAT Pro for on-board mission-critical space applications. In T. Vardanega and A. Wellings, editors, *Reliable Software Technologies - Ada-Europe 2005*, volume 3555 of *LNCS*. Springer-Verlag, 2005.

[22] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Tr. on Computers*, 39(9), 1990.

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

[24] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Tr. on Computers*, 44(1), January 1995.

[25] T. Vardanega and G. Caspersen. Using the Ravenscar Profile for space applications: The OBOSS case. In M. González-Harbour, editor, *Proceedings of the 10th International Workshop on Real-Time Ada Issues*, volume XXI, pages 96–104. Ada Letters, 2001.

[26] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063, pages 106 – 119, Palma de Mallorca, Spain, June 2004. Springer Verlag.

[27] J. Zamorano, A. Alonso, J. A. Pulido, and J. A. de la Puente. Implementing execution-time clocks for the Ada Ravenscar profile. In A. Llamosí and A. Strohmeier, editors, *Reliable Software Technologies - Ada-Europe 2004*, volume 3063 of *LNCS*. Springer-Verlag, 2004. ISBN 3-540-22011-9.

[28] J. Zamorano, J. F. Ruiz, and J. A. de la Puente. Implementing Ada.Real_Time.Clock and absolute delays in real-time kernels. In A. Strohmeier and D. Craeynest, editors, *Reliable Software Technologies — Ada-Europe 2001*, number 2043 in LNCS, pages 317–327. Springer-Verlag, 2001.

OCERA: A Framework based on Components for Real-time Embedded Applications

(Invited talk)

Alfons Crespo
DISCA Universidad Politécnica de Valencia
e-mail: alfons@disca.upv.es

Abstract

The recent introduction of Linux into the embedded sector has been one of the most exciting changes in the last few years. Based on the open-source model, it offers new possibilities to embedded engineers traditionally used to commercial operating systems. The main benefits address important areas for embedded software developers as cost and availability issue based on no runtime royalties and available source code at no charge.

The Linux kernel was not originally designed for real-time applications. Although kernel developers are actively working in improving the responsiveness of the kernel, Linux is still not suited to support hard real-time applications. However, the use of RTLinux implemented a real-time executive permits the execution of real-time tasks in a Linux system.

In this paper we present the OCERA architecture based on Linux kernel and RTLinux executive which provides support for critical real-time applications that need a very low response time and can be considered as critical, less critical and more complex real-time applications, that may not be trusted and both kind of applications incorporating new techniques in scheduling resource reservation, communications and fault tolerance. A list of components improving the functionalities and implementation of real-time applications is described.

*This work has been funded by the Commission of the European Communities under contract IST 35102 (OCERA project)

Lightweight RTAI for DSPs

Jens Kretzschmar, Robert Baumgartl
Department of Computer Science
Chemnitz University of Technology, Germany
{*jens.kretzschmar, robert.baumgartl*}@*informatik.tu-chemnitz.de*

Abstract

Digital Signal Processors (DSP) are an interesting alternative to conventional microcontrollers for embedded real-time systems. Up to now, no open source operating system has been widely accepted for DSPs. The special architecture of these chips prevents a simple port of Linux and its real-time modifications.

This paper presents a novel approach to implement real-time APIs for DSP systems with small memory resources: remove Linux from the Real-Time Application Interface (RTAI). We realized this concept for the VLIW DSP platform TMS320C62x. The interdependencies between Linux and RTAI are identified and eliminated. We document necessary modifications to RTAI and present a methodology to systematically port RTAI modules to our system. The resulting operating system is compared and evaluated to origninal RTAI. A careful optimization results in an interrupt latency below 100 clock cycles. The system is small enough to fit into DSP internal memory.

1. Introduction

During the last couple of years, Linux has attracted much interest within the embedded system market for its obvious advantages: freedom, flexibility and performance. The advent of RTLinux [13] and RTAI [4] has provided the foundation for the construction of Linux-based hard real-time systems.

Embedded systems based on digital signal processors (DSP) have been omitted from this trend for two main reasons:

- scarce memory and
- non-existent competitive compilers.

Application and even more operating system programming has been a task for only a few very skilled specialists producing few lines of code per day. Operating systems for DSPs have been very efficient, small and expensive. Almost always they are closed source. Porting software across different DSP platforms is usually difficult due to non-standardized OS APIs.

This situation has changed somewhat with the introduction of Texas Instruments' C6x DSP family. This processor is able to address several megabytes of memory, it has a fairly orthogonal instruction set and the compiler produces reasonably optimized code. Not surprisingly, ports of uCLinux to the C6x DSP have been published [5, 6, 7].

With regard to the main application area of DSPs, it would be favorable to have not only the plain Linux kernel but its real-time extensions available. RTLinux applications could be executed on DSP platforms and it seems in general very attractive to have a well-established and open source real-time OS API available for DSPs. Up to now, neither RTAI nor RTLinux have been ported to a DSP architecture. Therefore, we decided trying to migrate RTAI to a DSP. For historical reasons, we solely concentrated on RTAI.

The port has been done for the TMS320C62x VLIW fixed-point DSP architecture [10]. The chip has a Harvard architecture and provides separate internal memory banks for program and data of 256 kBytes each. Depending on the actual hardware platform external RAM of different sizes are possible. Clock rates range from 150 to 300 MHz. Due to its downward compatibility, our solution is also usable for the more recent type TMS320C64x, which is also used in application-specific DSPs, e. g. the TMS320DM642 and DM643. Typically, these processors incorporate between 1 and 8 MBytes of internal RAM and are clocked between 400 and 1000 MHz.

There is also a floating-point variant of the architecture, the C67x, which has been omitted from our port so far due to certain architectural differences to the C62x. We feel it would be not too difficultto include that version in the port, should the need arise.

The target hardware platform is a so-called Imaging Evaluation Kit (IEK C64x) by the french company Ateme [1].

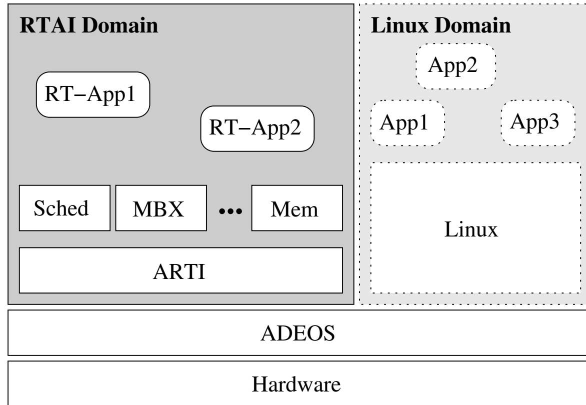


Figure 1. Basic architecture of RTAI

The rest of the paper is structured as follows. In section 2, we give a short overview of RTAI and the used development tools, motivate the elimination of Linux and give an overview of the porting process. Section 3 details some of the technical aspects and difficulties of the port. Efforts to optimize the resulting system are also described. The following section 4 documents the achieved performance and compares our solution to a conventional RTAI/x86 system. The final section 5 summarizes the main contributions of this work and gives a short outlook on further research.

2. Basic design

2.1. RTAI overview

Figure 1 depicts the basic structure of an RTAI system. The so-called ADEOS nanokernel realizes the domain concept and the virtual interrupt pipeline [12]. ADEOS is a separate entity and could also be used without RTAI, for instance to construct abstract machines. The ARTI (ADEOS Real-Time Interface) layer implements timers, generates `/proc` entries in Linux and contains some legacy functions. On top of it reside RTAI components which together constitute the real-time API and RTAI applications. From Linux' point-of-view both RTAI components and applications are specialized kernel modules. RTAI schedules the Linux domain (kernel and applications) with lowest priority, i. e. when no real-time application is ready to run. ADEOS prevents seizure of interrupts by Linux. More details on RTAI can be found elsewhere [4].

2.2. Compiler considerations

RTAI has been designed and implemented using the GNU Compiler Collection (GCC). Therefore, it would be natural to employ GCC for the port. Unfortunately, although GCC has indeed been ported (by our group) to the C6x DSP architecture, this port is not mature enough yet for serious development projects. Especially its optimizing features are inferior to Texas Instruments' C6x compiler [9]. Because our solution aims at low interrupt latency and low operating system overhead we decided against GCC.

For many years, Digital Signal Processors (DSPs) have been very special for programmers. To obtain optimum performance, it is quite common to implement even large software projects solely by means of hand-optimized assembler code. We did not pursue this approach for reasons of complexity and code maintainability. As we will demonstrate in section 4 this decision causes no intolerable performance penalty.

The only remaining option was to use the C compiler by Texas Instruments which represents the state-of-the-art in code optimization for the C6x architecture. Unfortunately, this tool is not open source. Additionally it is not fully compatible to GCC. For instance, the interface between C code and assembler statements differs in several ways from GCC.

By constantly improving our GCC port, we hope to migrate RTAI/C6x to that compiler in the future.

2.3. Eliminating Linux

To port RTAI to the DSP architecture, two principal design approaches can be distinguished. The first one is the 'classical': one of the Linux versions ported to the DSP must be extended with RTAI functionality.

Obviously, porting a monolithic kernel like Linux to a DSP causes serious overhead. Many kernel functions are obsolete within a DSP context (e. g. , drivers, memory management). Adding RTAI functionality (interrupt virtualization, real-time scheduler, . . .) to Linux is likely to increase that overhead. The resulting system will have a large memory footprint, large interrupt latency and large processing overhead. We believe this to result in an inacceptable performance penalty and therefore propose a radically different approach: Port RTAI to the DSP *without* the Linux kernel! Hopefully, the resulting system would be small and fast.

The feasibility of this approach strongly depends on the interdependencies between RTAI and Linux. If RTAI is using many Linux kernel functions, the port will be difficult, because these functions have to be reimplemented. As we will demonstrate, this is not the case.

In a typical RTAI system, Linux is used for

- booting the system,

- real-time system maintenance (adding tasks using insmod, modifying system behavior etc.),
- non-real-time communication to the RT subsystem (e. g. logging).

Additionally, it offers some advantages to let the development environment and the target platform reside on the same machine.

If these responsibilities are dispensable or can be emulated, the resulting system does not need Linux.

The Linux boot mechanism is not of any value for a DSP, because these devices usually boot from EEPROM or some interconnection network. System maintenance solely depends on the execution environment and therefore must be reimplemented anyway. Logging functionality is not immediately necessary and will be implemented if the need arises. Clearly it is impossible to use the DSP as development machine. Hence, we can conclude that Linux may be safely removed from an RTAI port to a DSP (the dotted parts in figure 1 vanish).

2.4. Challenges

Migrating software between such radically different architectures is obviously a demanding task. The peculiarities of the DSP and its C compiler rendered the port difficult but not impossible.

In contrast to conventional ports, two different types of functionality must be localized and reimplemented:

1. hardware-dependent code, and
2. functions provided by Linux.

The former comprises interrupt handling, context switch and timer manipulation. The latter is, fortunately, only a small set of functions: printk(), kmalloc(), request_irq(), some data type definitions and functions for bit manipulations (bitops.h). In addition, the Linux functions to transfer data between kernel and user address spaces (copy_to/from_user() have to be removed from certain RTAI modules (for instance, the mailboxes).

Hardware-dependent code in RTAI is well separated. The amount of porting work differs from module to module. The scheduler is—with the notable exception of the context switch—almost completely hardware- and Linux-independent. Memory allocation is a different story: the MALLOC module uses the Linux kernel function kmalloc().

For rapid prototyping we used the following technique. Whenever a Linux kernel function was encountered, it was substituted by a reimplementation based on Texas Instruments' library without paying attention to memory overhead. For instance, a call to kmalloc() would simply be substituted by a call to the TI library function MEM_alloc(). After basic RTAI/C6x was up and running the according function was reevaluated, whether it could be safely removed, must be implemented from scratch or could be resubstituted by an RTAI module function which was not available initially. In the case of kmalloc(), the last option was chosen, that is, it was substituted by a call to rt_malloc() provided by the MALLOC module.

3. Porting RTAI

Porting RTAI to the C6x DSP by eliminating Linux can be structured as follows:

- adapting ADEOS,
- implementing ARTI functions (timers),
- migrating individual RTAI modules: scheduler, memory management, MBX, ...
- implementing a new module loader and
- optimizing the system.

The individual stages are briefly described in this section.

3.1. Adapting ADEOS

Porting the ADEOS abstraction layer was surprisingly easy. All functions and data structures that are important for registering and initializing a new domain were adopted. Simplification is possible, because ADEOS/C6x does not modify interrupt handling of operating systems executed on top. Functions that manipulate the interrupt hardware like adeos_hw_cli() were implemented from scratch. Many ADEOS functions are obsolete in the DSP context and have been subsequently omitted, among them

- ADEOS' semaphore implementation,
- SMP code,
- domain manipulation (suspending, deregistering).

3.2. Timer emulation

For obvious reasons, a very precise emulation of the RTAI timing functionality is crucial. Original RTAI time keeping uses three different facilities: the 8254 timer for generating interrupts, the time stamp counter (TSC) register for accurate execution time measurement and scheduling decisions and the Linux global kernel variable jiffies. The C6x provides three timers

which offer a reasonable resolution and are able to generate interrupts. Table 1 compares the timing facilities of the x86 and DSP architectures.

One of the DSP timers is used exclusively to provide an 8254 emulation. Another timer approximates the behavior of the TSC by using its shortest possible period. Due to the 40 bits precision of arithmetic operands, the maximum representable time value is 14660 seconds (four hours) for a 600 MHz DSP which might not suffice for all application scenarios. We hope to eliminate this limitation in the future.

The Linux variable `jiffies` is used very seldomly within RTAI. It would have been easy to replace it. Due to its usage at innumerable locations within the Linux kernel we felt it would be favorable to preserve it because very probably kernel (and device driver) programmers are used to its existence. To ease porting of RTAI/x86 applications, we decided to implement a `jiffies` emulation.

3.3. Porting RTAI modules

As mentioned in section 2.4 the scheduler is easy to port. Only the context switch is hardware-dependent because registers have to be saved and restored. Its implementation is straightforward though: all registers and the return address must be saved on the stack, certain task management structures must be manipulated (e. g. the pointer to the current task `rt_smp_current[0]` is updated), the stack pointer of the next task is obtained and finally the next task's registers have to be restored. Because the C6x is a uniprocessor the various multiprocessor schedulers were not regarded.

In original RTAI another context switch exists: the so-called domain switch is performed when control is transferred between different operating systems. In RTAI/C6x, domain switches never occur because it consists of only one domain. Therefore, domain switches have been eliminated.

The MALLOC module manages a memory heap available to applications by means of a buddy system. The size of this heap is defined at compile time. When RTAI is booted, the heap is allocated with a call to `kmalloc()` from Linux kernel space. When the heap is fully allocated by RTAI applications and more heap is still required it is extended by calling the internal function `alloc_extend()` which essentially allocates another chunk of Linux kernel memory via `kmalloc()`.

The following aspects of the module were modified. First, instead of calling `kmalloc()` at boot time, a memory block of the configured size is simply reserved by the development tools. Second, there is no way of increasing the size of the heap at run time (`kmalloc()` will always fail if the heap has been consumed). Hence, the programmer must carefully predict the needed heap size before. Finally, because the DSP offers two different memory spaces, MALLOC manages *two* heaps. The macros `rt_malloc()`, `rt_text_malloc()`, `rt_free()` and `rt_text_free()` constitute the programming interface.

Currently, only the most needed modules have been ported to RTAI/C6x: the scheduler, the memory allocator MALLOC, the semaphore SEM and two communication modules, MQ and MBX. A significant effort in the future will be to bring more modules into RTAI/C6x. The following methodology proved successful:

Redefine Data Types. RTAI extensively uses data types that are defined in Linux header files. Therefore, data types like `size_t` or `time_t` have to be redefined. Redefining data types was the main effort in porting both communication modules.

Including Linux Header Files. Every RTAI source file includes some Linux headers. In most cases, only very few pieces are really needed from these files. Either the required lines are brought into RTAI/C6x or the complete header file is taken over, as it has been done for `errno.h`.

Adapting RTAI Header Files. Every RTAI module defines its API in a separate header file which must be made known to RTAI by including it in `rtai_schedcore.h`.

Substituting Macros. All macros with variable parameter list must be substituted by an inline function due to limitations of TI's compiler. Other macros like `MODULE_LICENCE` etc. can be eliminated.

Reimplementing Linux functions. In RTAI/x86 modules may access all Linux kernel functions and data. Because Linux does not exist in RTAI/C6x, all these functions must be reimplemented or eliminated. Fortunately, in the modules ported so far only a small set of Linux functions and data was referenced. An example is `printk()` which is used extensively throughout RTAI for logging purposes. During debugging, this function is substituted by the TI library function `LOG_printf()` which passes the logging messages to the simulator. Another example is the `jiffies` variable for which we provide a suitable reimplementation as mentioned in section 3.2.

Handling 64-bits data. Operations on `long long` data (64 bits) are not supported by the compiler. The C6x DSPs are only able to operate on 40 bits wide operands. Therefore, it must be analyzed whether this reduced precision can be tolerated. If not, software emulation must be employed. The MSG module uses several 64 bits constants, therefore other communication

Table 1. Comparison of timing hardware in x86 and C6x architectures

	C6x, 600 MHz	x86 8254	x86 TSC, 600 MHz
register size	32 bits	16 bits	64 bits
frequency	75 MHz	1.19 MHz	600 MHz
shortest period	13 ns	840 ns	–
longest period	57 s	0.055 s	$3 \cdot 10^{10}$ s

modules were ported before.

3.4. Module loader

As a consequence of the elimination of Linux, the tools `insmod` and `rmmod` which are used in conventional RTAI systems to insert or remove individual modules and applications are not available anymore. A simple alternative is the transferral of one large binary file containing all RTAI modules and needed applications to the DSP at boot time. Because this prevents task and system modifications at runtime, we decided to implement a more flexible mechanism. Figure 2 illustrates its principle.

Providing a universal communication interface for all kinds of DSP hardware is beyond the scope of this paper, because DSP hardware is in general very heterogeneous (different communication interfaces, chip sets, external periphery ...). Therefore, we followed a pragmatic approach which solely concentrated onto our current hardware platform. Some implications for the implementation of a generic communication infrastructure between a Linux host and DSP hardware are described in [2].

In our current system configuration, the DSP communicates to the host processor via a JTAG interface. Due to lack of documentation, direct access to the host's JTAG interface is impossible. Instead, the Code Composer Studio IDE is used as a gateway. Our `insmod` and `rmmod` tools connect via COM to CCS which interacts with the JTAG port. The so-called RTDX libraries by Texas Instruments [11] encapsulate JTAG communication on both the DSP and the host side. We developed a very simple communication protocol between Host and DSP which is thoroughly described in [8]. Basically it allows to install and remove RTAI modules and applications.

Furthermore, the DSP needs a loader which analyzes the received executable files, relocates the contained sections and finally starts the program. Unfortunately, Linux' object loader could not be adapted for that purpose because it uses the ELF binary format, whereas the DSP compiler generates COFF binaries. Therefore, a new DSP module loader has been implemented from scratch. On arrival of a new COFF binary it does the following:

1. read section headers from received COFF file, allocate memory, copy sections into it,
2. relocate pointer destinations,
3. copy initial data values to the object's data memory,
4. register the module, save its exit handler,
5. jump to the entry symbol `_app_init`.

Module removal is straightforward. Several implementation problems and unclear technical documentation made the module loader one of the most difficult parts of our project. More details can be found in [8].

The loader is not needed if the task set is constant at runtime.

3.5. Optimization

After a timer interrupt event all software layers (IRQ handler, ADEOS, ARTI, scheduler) have to be traversed until finally a ready task is activated. Therefore, the time to process a timer interrupt was chosen as quantitative measure of the initial port's efficiency. The measurement was performed using a DSP simulator. Due to the C6x architecture such measurements are possible with single cycle precision.

Note that this time is not the interrupt latency (which is lower, cf. section 4.1).

The first RTAI/C6x version needed 1448 cycles for the activation of a task ready for execution. Almost half of the time was spent within ADEOS (cf. table 2). Therefore, ADEOS was analyzed and optimized as follows:

ADEOS is designed to manage several operating system instances in parallel. In a classical RTAI system two such domains are realized: RTAI itself and Linux. It is possible, to install and execute more operating system instances but to our knowledge this has not been

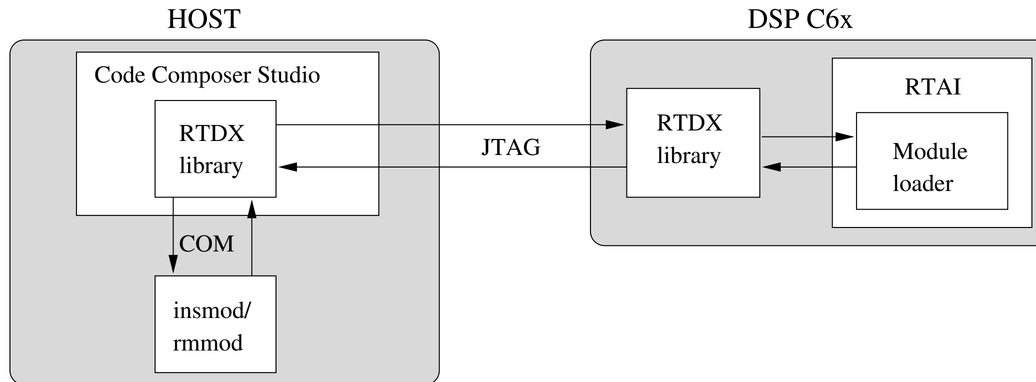


Figure 2. Communication between host and DSP

explored yet. Because we removed Linux from RTAI and we do not intend to execute another operating system it seemed reasonable to us to bypass the interrupt pipeline completely. As a consequence, several unnecessary functions were removed. The principle of queueing incoming interrupts in software when interrupts are disabled was not changed though in order to preserve standard ADEOS behavior. This optimization saved approximately 230 clock cycles.

Additional execution time improvements were achieved by exploiting certain architectural features of the DSP: inlining bit operations, optimizing interrupt enable and disable functions and parallel register saving. By this, we were able to save another 216 clock cycles.

Still, ADEOS needed some 320 cycles. Because low interrupt latency is crucial in many DSP applications we decided to introduce a change in interrupt semantics: only one incoming interrupt is queued up now if interrupts are disabled (interrupts may get lost when they are disabled for too long, now). Because critical sections are rare and short we feel that the limitation can be justified. This last optimization effort reduced ADEOS' execution time drastically to only 32 clock cycles!

4. Achieved performance

4.1. Interrupt latency

Two interrupt scenarios must be distinguished. An incoming timer interrupt causes all four RTAI components (IRQ handler, ADEOS, ARTI and the scheduler) to run, because it usually means that a task has been activated (cf. section 3.5).

When an other interrupt arrives, the IRQ handler, ADEOS and ARTI are executed whereas the scheduler is omitted, because the association of tasks with interrupts is performed in ARTI (no scheduling decision is necessary). Therefore, interrupt latency is the sum of the execution times of the IRQ handler, ADEOS and ARTI.

Table 2 shows the contribution of every RTAI component involved and reflects both interrupt latency and the time to process a timer interrupt. We compare our initial port, the current status obtained by the optimizations described in section 3.5 and a typical x86-based RTAI system (AMD Duron 650MHz, Linux 2.4.26, RTAI 3.1). All values are in clock cycles. The results for the x86-based system were obtained by reading out the Time Stamp Counter (TSC) register at selected points. We did not pay attention to serialization, therefore the results may be not as accurate as for the DSP.

The time to save registers has not been measured for the x86 version. Due to the low number of registers (nine), it can be neglected.

It it obvious that RTAI/x86 needs much more clock cycles due to the different chip architecture and the fact that original RTAI provides richer functionality.

The obtained results indicate that ADEOS/C6x is competitive. The optimization process proved successful. Especially the low interrupt latency renders RTAI/C6x suitable for reactive systems.

The optimization potential of the scheduler seems comparatively small as long as no change in its semantics is intended. A subject of further study could be the design of a specialized DSP scheduler which offers a similar performance boost as we obtained by optimizing ADEOS.

4.2. Task switch duration

Another important aspect is task switch efficiency (cf. Table 3). The duration of a task switch depends on whether the task to be activated has previously released the processor voluntarily (i. e. by

Table 2. Comparison of time to process a timer interrupt (in clock cycles)

	RTAI/C6x (initial)	RTAI/C6x (optimized)	RTAI/x86
IRQ Handler+saveregs	64	40	?
ADEOS	704	32	≈ 1182
ARTI	24	24	≈ 47
Scheduler	656	552	≈ 6804
interrupt latency	792	96	≈ 1229
timer interrupt processing	1448	648	≈ 8033

rt_task_wait_period()) or preemptively (has been interrupted by the timer). Again, we can report a significant performance gain by optimization.

Table 3. Task switch time (in clock cycles)

	RTAI/C6x (initial)	RTAI/C6x (optimized)	RTAI/x86
voluntarily	704	512	≈ 16800
preemptively	1448	784	≈ 11400

It is interesting to note that a x86 task switch needs 14 to 32 times more cycles than its C6x counterpart. This result is somewhat mitigated by the higher clock rates of x86-based processors but even considering the highest available clock rates for both processors the DSP version is in the lead.

More measurement results can be found in [8].

4.3. Memory footprint

One of the motivating factors for this project was to adapt RTAI for systems with very limited memory resources. Because the actual memory consumption is influenced by the number of needed RTAI modules and the number of application tasks, we consider two different systems.

A minimum system consisting of ADEOS, ARTI, SCHED and MALLOC requires only 39808 bytes of code and 6151 bytes of data memory. If the DSP BIOS by Texas Instruments (a kind of low-level library) is used, another 1216 bytes of code and 6816 bytes of data are needed. Every application task requires memory for its code, data, its stack and used application libraries and a certain amount of space for management structures (e. g. an RT_TASK structure).

The full set of ported functionality so far comprises the abstraction layers ADEOS and ARTI, the modules SCHED, MALLOC, SEM, MBX and MQ as well as the loader to dynamically modify the task set at run-time. This system requires 87104 bytes of code and 24474 bytes of data memory (plus the space for DSP BIOS, otherwise run-time communication to the host is impossible).

The numbers indicate that a real-time system based on Lightweight RTAI is very likely to fit into internal RAM of even the smallest C6x variant.

5. Conclusions and outlook

We demonstrate the feasibility of separating RTAI from Linux. Additionally, we show that it is possible to provide the RTAI API for a DSP without the overhead caused by the Linux kernel. Modules for scheduling, memory allocation, inter-process communication and semaphores have been ported to the DSP. Task switch time and scheduler performance are competitive in comparison to x86-based systems. Interrupt latency is especially low and outperforms x86-based systems by more than a factor of 10 in clock cycles. As expected, the memory footprint is small enough to locate RTAI and application tasks fully in DSP internal memory.

The main motivation for this work was to establish an open-source real-time API for the C6x DSP which allows manipulations of the task set at runtime. We feel this has been achieved.

Currently, the project is under active development. Some of the necessary improvements were mentioned throughout this paper. Among other, the following work must be done:

- porting remaining RTAI modules,
- implementation of DSP-specific RTAI modules,
- ease some of the module loader's limitations.

We will provide CVS snapshots at our website

http://rtg.informatik.tu-chemnitz.de

soon. Interested developers are welcome to join the project. Hopefully, ports to other DSP architectures will follow.

As soon as a competitive GCC for the C6x DSP is available, we will migrate our solution from Windows to the Linux host operating system. This project is currently under active development [3]. Several Linux drivers and tools for communication with DSP hardware have been developed. The GCC suite is improved continuously.

Another interesting question is whether Lightweight RTAI can be established for systems using conventional processors. This would require migrating drivers from Linux to RTAI which seems to be an interesting project itself!

References

[1] Ateme SA: *IEK C64x Imaging Evaluation Kit*, Data sheet, http://www.ateme.com/products/iekc64.php, 2005

[2] Robert Baumgartl, Ingo Oeser, Daniel Schreiber, Michael Schwindt: *DSP Accelerator Support for Linux*, Proceedings of the International Conference on Signal Processing Applications & Technology (ICSPAT), Dallas, October, 2000

[3] Robert Baumgartl, Ingo Oeser, Mirko Parthey, Adrian Strätling: *Bridging the Gap between Linux and DSPs*, Proceedings of European DSP Education and Research Workshop (EDERS), Birmingham, 2004

[4] Pierre Cloutier et al. *DIAPM-RTAI Position Paper*, Proceedings of the 21st IEEE Real-Time Systems Symposium and Real-Time Linux Workshop, Orlando, 2000

[5] Eatamar Drory, Or Sagi. *Introducing MediaLinux – A new real-time Linux Approach* http://www.linuxdevices.com/articles/AT554348551.html, 2003

[6] Eatamar Drory. *Linux-programmable Audio-Video device based on TI DM64x DSP*, Proceedings of the Texas Instruments Developer Conference (TIDC), Birmingham, 2004

[7] Jaluna SA: *Jaluna OSware Linux Edition for TI C6000 DSP*, Product Datasheet, http://www.jaluna.com/ fileadmin/pdf/TIDSP2305JalunaUS.pdf, 2005

[8] Jens Kretzschmar. *Implementing RTAI on a DSP Processor without Linux*, Diploma Thesis, Chemnitz University of Technology, May, 2005

[9] Jan Parthey, Robert Baumgartl: *Porting GCC to the TMS320-C6000 DSP Architecture*, Proceedings of the Global Signal Processing Conference and Expo (GSPx), Santa Clara, September, 2004

[10] Texas Instruments, Inc.: *TMS320C6000 CPU and Instruction Set Reference Guide*, October 2000

[11] Texas Instruments, Inc.: *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide*, November, 2001

[12] Karim Yaghmour. *Adaptive Domain Environment for Operating Systems*, Whitepaper, http://www.opersys.com/ftp/pub/Adeos/adeos.pdf, 2001

[13] Victor Yodaiken, Michael Barabanov. *Real-Time Linux*, Linux Journal, March, 1996

Power Measurement as the Basis for Power Management

David C. Snowdon, Stefan M. Petters and Gernot Heiser

National ICT Australia
and
School of Computer Science and Engineering
University of NSW, Sydney 2052, Australia
{daves,smp,gernot}@cse.unsw.edu.au

Abstract

Energy has become a critical component of computer system design, particularly in the embedded space where the source of energy is often finite. While hardware design has the more significant effect on the system's power consumption, well designed system and application software make an important contribution to controlling the energy consumed.

In order to optimise software systems to reduce energy consumption, feedback is required. Traditional techniques rely heavily on models of the system which have various disadvantages. We examine the benefits of using live power measurements using statistical sampling for both off-line and on-line feedback on application power consumption. A hardware platform is manufactured, operating system modifications made, and extensive validation carried out. We conclude that the idea shows promise and justifies further investigation.

1 Introduction

Computer power usage has become an important area of research for a number of reasons. High-performance systems are limited by problems with thermal dissipation, and portable and embedded systems are often supplied power from a limited source (batteries, solar panels). In both cases, energy efficiency is a key quality determining computer system utility. Energy awareness and management is critical in improving the energy efficiency of a computer system. This is loosely termed *power management*.

There are three key problems which power management research has attempted to solve. Generally, these problems are:

- determining how, and why, power is used in a computer system;
- configuring hardware to match power and performance requirements (dynamic voltage scaling and low power idle states have become standard on modern hardware);
- adapting software to use available resources efficiently.

We observe that it is useful to know, to some level of detail, how and why a computer uses power. Such information is used to evaluate and analyse the operation of power management algorithms, can help to optimise application and operating system code with respect to power consumption, and supports the generation of off-line schedules. We further hypothesise that, should the information be available at run-time, it would be useful as the basis for some classes of power management algorithms, particularly for intelligent throttling of system components (CPU via frequency scaling, hard disk via spin-down, etc).

This paper presents the details of an investigation into obtaining power usage information through direct measurement of the current supplied to the computer's processor core, memory and IO subsystems. Hardware and software infrastructure for making these measurements is developed, the overheads examined, and the accuracy of the system assessed.

The initial version of the system was used as an off-line analysis tool. The same methodology can be used to perform on-line measurements, giving feedback to the operating system and user-level processes, and allowing the system to block processes which exceed an energy quota. The effectiveness of these techniques is examined, leading to ways in which the information can be used, both for off-line analysis and for on-line accounting/power-management.

2 Previous work

A variety of energy-estimation techniques have been developed with a view to programmer feedback, power management research evaluation, and on-line accounting. Many of these techniques rely on indirect measurements coupled with a model of the system in order to estimate the power used. There are several disadvantages to a model-based approach with regard to power estimation:

- A sufficiently detailed model is required in order to obtain a given accuracy. Computational complexity must be traded with the detail and accuracy of the model, and sufficient information must be available in order to construct it. Circuit-level simulations require the actual circuit design, etc.
- Modifications and additions to the system require changes to the model — a significant engineering effort.
- The model must be verified against real-world measurements.
- Most established simulators concentrate on the CPU rather than the entire system.
- Models will inevitably miss details: the model of a hard disk is unlikely to take into account the physical condition and situation of the disk, which might affect power consumption.
- Similar to execution times the manufacturers are reluctant to provide the details required, since it may give advantageous information to competitors.

One advantage of model-based estimation techniques is that the parameters fed to the model are often useful in their own right.

Simulators are often proposed as an off-line analysis tool [5, 7, 10–12]. Typically, trace output from an architecture-level simulator (such as SimpleScalar [**?**]) is obtained, and an energy associated with each instruction in the trace. The energy used is usually pre-calibrated via measurement of the actual hardware, since it is rare that the detailed design information necessary to accurately determine this via circuit-level simulation is available.

Event-counter based techniques, typified by Bellosa and Weissel's work [14], use live data generated by CPU performance counters as the input to a model. The counters are configured to measure events which are significant to the energy consumption (cache misses, instructions retired, etc), and a model interprets these results to estimate the total CPU power consumption. The accuracy of the system is therefore determined by the amount of information available (the number of event counters and measured properties). The model used is typically simple and the simplifications can lead to inaccuracies in the estimation. These systems have the advantage that they can efficiently be used on-line, allowing the information obtained to be used by power management algorithms. This technique has only applied to CPUs, since performance counters are generally not available in peripherals. The only exception to this are memories, which could be observed, albeit indirectly, by counting cache misses and write back operations.

State-based accounting techniques such as those employed in ECOSystem [16] instrument operating system software to track the state of the CPU and its peripherals (e.g. for a disk, whether it is spun up or down and whether it is active or idle). The power in each of these states is pre-calibrated, and the time spent in each used to determine the energy consumed. In ECOSystem the energy is then accounted to the processes causing the device to transition out of its lowest-power state. These techniques have the advantage of being an all-software solution which can simulate the entire system, however they fail to capture any variation of the power within a given state. The accuracy of the technique therefore depends on the number of states (detail of the model).

Jejurikar and Gupta introduce a system which uses off-line and on-line analysis to reduce the energy usage of their applications [8]. The off-line part produces the slowdown factor in such a way that under the assumption that every task runs for its worst-case execution time (WCET) all deadlines are just met. In the real system deployment this will hardly be the case as most applications almost never run for their WCET. The on-line part takes advantage of this "gain time" to increase the slowdown in order to keep CPU utilisation high. Such an approach would complement the proposed measurement technique we are presenting here, which may be used to produce the input values for their optimisation. Similarly, AlEnawy and Aydin look at on-line and off-line methods [2]. Their results are produced by simulations rather than using direct measurements our system would enable.

An alternative to using model-based power estimation techniques is proposed by Flinn [6]. He uses statistical sampling techniques (as are widely used in sampling profilers such as Shark [3]). Measurements of the power consumed by a computer are taken periodically, along with the program counter, and process ID. This information is stored and later analysed by attributing each power sample to a process and to a symbol within the process' code. Although the sampling rate is slow in comparison to the CPU's clock rate, over time enough samples are attributed

to each piece of code (process/symbol) that a statistically significant average is obtained. This information is more detailed than either state-based or event-counter-based techniques can provide, and comparable with (and potentially more accurate than) results obtained via simulation. Flinn called his energy-profiler Powerscope.

Using these direct measurements counters the disadvantages of model-based approaches, however there are several problems which were not addressed in the original and subsequent Powerscope work.

One of particular importance is the inability of the system to account for background activity (activity which is not associated with the process which is running on the CPU) such as blocking disk reads and writes). In the original Powerscope system this activity is accounted to the running process rather than the blocked process which is actually responsible for the activity. It is impossible to distinguish between the power consumed by the disk (which should be attributed to the blocked process) and the power consumed by the processor (which should be attributed to the running process).

A related issue is not being able to understand how and where energy is being used in the system. For example, it is not possible to discern between energy consumed by the CPU and memory subsystems (without estimating via a system model).

Other problems include the cumbersome hardware setups (the original Powerscope work involved a second computer connected to an external multimeter in order to perform the power measurements. The low sampling rate which was used means very short functions are not measured accurately (owing to an effective low-pass filter at the measurement input). Lastly, the information can not be used at run-time, since the measurements are taken using a different computer.

3 Measurement system

Many of the problems identified in Powerscope can be avoided by building a computer with hardware support for taking the measurements. one of these problems is, for example, the attribution of background activities, such as network traffic or hard-disk data transfer, to the wrong process. By splitting up the measurements into separate entities for these devices and associating those with modules instead of processes, a deferred attribution to processes is possible.

For example, a system with a CPU, memory, network card, and hard disk, would measure the power consumed by each of these independently. The system can then associate the CPU measurement with the running process, the network card with processes which have submitted or received packets, the disk with processes using file systems, and the memory system with processes causing memory bus accesses (which may be associated with devices, owing to DMA).

3.1 The PLEB 2 Platform

PLEB 2 is a single-board computer based on the Intel XScale PXA255 [1]. It was custom-designed primarily as a reference to be used in embedded systems research, but secondarily as a platform for applications implementation.

The PXA255 was chosen as being representative of high-performance CPUs designed for embedded systems. It consists of a 400MHz ARMv5TE compatible core combined with a set of on-chip peripheral units including memory, interrupt, DMA and LCD controllers.

The main processing core consists of the CPU, SRAM and flash memory. Three switching power supplies generate core, memory and IO power from lithium-ion battery voltage. A minimal set of peripherals (infra-red, USB, and serial port) are provided on-board, and supplied from IO power. An 8-bit microcontroller (an Atmel AVR) resides on-board in a supervisory role. This models a typical embedded system. Un-used pins on the XScale are connected to two connectors which allow for other peripheral electronics to be added.

Linux 2.4.19, Linux 2.6.8 and L4ka::Pistachio [9] have been modified to run on PLEB 2 hardware.

Of particular interest in this context are the device's features designed to support power management. The CPU core, and memory clock frequencies can be changed in (very roughly) 30MHz and 10MHz intervals respectively (the intervals are smaller for the lower frequencies), although not all combinations of clock frequency, bus frequency, memory frequency, etc. are possible.

One problem encountered with frequency scaling (changing the frequency on-the-fly to adapt to performance requirements, thereby saving energy) is that there is an overhead associated with changing frequencies. The XScale attempts to solve this by offering a *turbo mode* which is a second frequency mode. Changing between the run (normal) mode, and turbo mode is much faster than changing between arbitrary frequencies because the system can perform a synchronous switch between the modes, without having to disturb the memory controller, LCD controller, and other peripherals.

As well as being capable of setting its core clock frequency, the CPU can enter a number of low-power states. These states disable circuitry within the CPU: the more circuitry disabled, the longer it takes to re-activate. Therefore it is necessary to ensure that the energy saved by being in

the sleep for a period of time is enough to offset the energy used to sleep and wake up.

Techniques for voltage scaling have also seen a lot of attention in the literature (most notably by Weiser [13], but also others too numerous to cite) - at a lowered frequency, the CPU's core voltage can be reduced, allowing quadratic energy savings (at half the frequency, the system will use a quarter of the power). The power supply used on PLEB 2 supports setting the voltage between 0.8 and 1.5V in 0.1V increments. The chip communicates with the PXA255 via I2C (a bidirectional serial bus). Similarly, the memory voltage can be set to either 3.3V or 2.5V, depending on speed and peripheral requirements.

Lastly, the Micron SDRAM used can place itself in power-down and self-refresh modes. These low-power states can save significant amounts of power. The Intel flash memory used for non-volatile storage has power-saving features, but they are not controllable, and therefore are of little interest in terms of power management.

3.2 Power measurement hardware

In support of embedded systems research, PLEB 2 was designed with power-measurement hardware on-board. Each of the three power supplies (nominally for the CPU core, memory and IO)[1] are instrumented with current sensors. Each power supply is well regulated to its designated voltage, therefore the voltage is assumed to be constant and the current is proportional to the power (P=IV).

The microcontroller on-board has an integrated analogue-to-digital converter and can read the sensors at up to 15kHz. Since it can only measure one of the sensors at a time, this equates to a maximum of 5kHz on the individual sensors when all are measured at equal rates.

Samples are transferred from the microcontroller to the PXA255 as they are taken (as described further in Section 4). Communication between the microcontroller and the PXA255 is via I2C. This is a significant limitation since I2C transfers data slowly (400kbps). Thus, in order to avoid excessive overhead, the transfer of each measurement requires several interrupts (one per byte — each measurement requires two data bytes to be transferred, along with the I2C bus' addressing byte). Furthermore, the maximum sampling rate is limited by the rate at which data can be transferred between the processors.

Figure 1 details the protocol for taking measurements: once enabled, the microcontroller interrupts the PXA255

[1] Note that, should peripherals (such as a network interface) be connected to the system, they will be connected to one of the three power supplies. This breaks the power-supply-per-device concept.

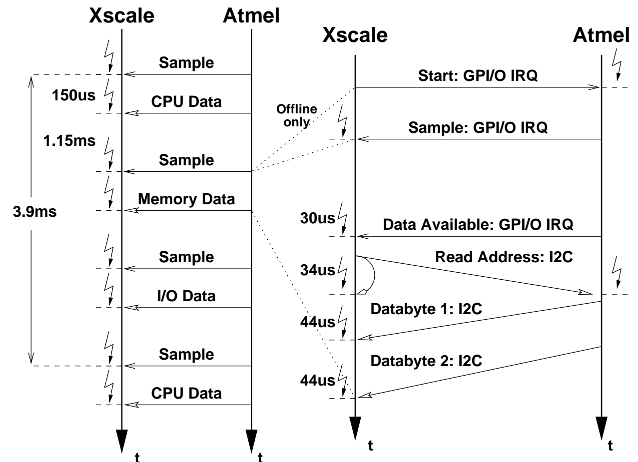


Figure 1. On-board communications timing of on-line analysis

periodically. The PXA255 initiates a measurement by asserting an interrupt line on the ATMEL chip. After a fixed delay the measurement of one of the three sensors is started. At the same moment the interrupt at the PXA255 is asserted. The interrupt handler records the state of the system (PC, PID, etc.) when the interrupt occurred. Once the analogue-to-digital converter has completed, the microcontroller interrupts the XScale a second time, triggering the XScale to start a transfer using I2C by sending a read command over the bus. This generates three further interrupts. After a short pause (which controls the sampling frequency) the microcontroller moves on to the next sensor to be sampled and the process repeats. The data transferred is stored by the XScale along with the PID and PC information previously recorded.

4 Off-line analysis

The experiments presented in this paper were conducted using Linux 2.4.19 because of its immediate availability. Kernel modules, as well as minor changes to the kernel itself, were used to implement communications with the microcontroller, the off-line analysis and energy accounting and budgeting (Section **??**).

Off-line analysis in this context implies that the system collects data at run-time, and stores it for later examination. Power and time used by each process running on the system, as well as each function within the process and its shared libraries can be obtained from the data collected [6].

The off-line analysis facility is based on a port of the Powerscope code to the PLEB 2 platform which has been

Benchmark	No profiling	Off-line	% overhead	On-line	% overhead
gzip	10.025	10.83	8.03	10.784	7.57
mpg123	30.256	31.213	3.16	31.071	2.69
vision	54.664	55.902	2.26	55.803	2.08
celp	85.397	87.34	2.28	87.17	2.08

Table 1. Time Overhead Introduced by the Measurement System in seconds

extended in order to take advantage of the extra hardware features available.

The Powerscope framework obtains samples via the protocol outlined in Section 3.2. It stores the samples for later analysis via a user-level daemon which reads from a kernel buffer (un-modified from the original implementation). The tool designed to analyse the data was modified to accommodate the three current sensors which do not sample concurrently.

This arrangement has a number of advantages over the original Powerscope implementation:

- Three current sensors are sampled, giving a user insight into how power is used in the system (it is possible to distinguish between memory and CPU power, for example). Further sensors can be added easily within the same software infrastructure.
- Because each of the three current sensors are connected to measure the major functional units, asynchronous activity (e.g. IO) can be accounted to the correct process and code. (ie. we can distinguish between background activity and activity directly correlated with the program counter and present process ID).
- The device is an integrated unit with no external apparatus required (Powerscope used a second computer and multimeter to reduce overhead on the system being profiled). This makes using the tool as easy as any other profiling tool. This also means there is only one data file which needs to be analysed, saving the need to move copy samples and data files to the same location.

The information gathered can be used in various ways. One way would be to guide the trade off between memory hierarchy and performance. Applications depending heavily on the CPU might benefit from the increased reuse of previous computation results stored in main memory, while applications with a large memory-bandwidth requirements can be optimised by recomputing values instead of relying on results stored in memory. In such a way the performance and energy usage could be optimised.

5 On-line analysis

5.1 Energy accounting

Because the XScale is set up to receive its own power measurement data, the information can be used on-line at run-time. The method of receiving data is very similar to the off-line system. For each sample, the value obtained is accumulated in Linux's process control (task) structure, and a field indicating the number of samples is incremented. Using this, the information is made available at user-level via the Linux `/proc` interface.

The method of taking direct measurements of the power consumed has numerous advantages over other methods of estimating the power consumed on-line:

- It does not employ a model, and therefore is not hindered by inaccuracies in that model. Furthermore, the extensive development time required to build an accurate model is avoided.
- Computation associated with accurate model-based simulations effectively prohibits their use for on-line power estimations.
- When comparing with state-based power estimators [15], which are often used for on-line power management in the literature, the measurement-based system can capture variation within a single state (for example, network interface power will vary greatly depending on whether it is sending, receiving, or both. The likelihood of these states can not be predicted by the operating system. Furthermore, the energy expended per packet will depend on the availability of the network).
- The approach does not only cover the CPU, but all the components within a system which are usually not covered in indirect measurement or simulator-based approaches.
- It is also possible, with little effort, to extend the approach to charge background IO activity to the process

	Powerscope CPU	LEA CPU	Powerscope Mem	LEA Mem
copymem	0.306	0.308	0.347	0.352
fillmem	0.310	0.314	0.405	0.412
fp-exercise	0.315	0.318	0.211	0.212
add_bench	0.268		0.211	

Table 2. Comparison of some typical results for on-line (LEA) and off-line (Powerscope) measurements. All measurements in Watts.

initiating this activity, rather than to whichever process is running during this background activity. This gives similar capabilities (with better accuracy) to the state-based currentcy system [15].

5.2 Energy budgeting

The on-line accounting technique has been used to implement an energy budgeting system. The information available allows the operating system to make scheduling decisions based on energy related criteria. The implementation is similar to the currentcy approach (cf. [15]) described in Section 2.

The OS process control structure is augmented with an energy remaining and energy budget field. The energy accounting system is used to decrease the energy remaining. Periodically (in the present implementation, once per second), the energy remaining field is reset to the budget value. The scheduler was modified to ignore processes with a negative energy remaining field. This halts processes whose budget has been exhausted until it is replenished.

The system is able to control the processes' power using direct measurements, rather than state-based accounting, as was deployed in the currentcy work. This allows the processes to be throttled more accurately. A Linux `/proc` interface allows to set the budget for each process.

The energy budgeting system allows control over how much energy processes use. This is a mechanism by which power management algorithms can throttle processes (ignoring quality of service constraints). Desired goals achievable include obtaining a desired battery lifetime, or maintaining a maximum CPU temperature. Further work will revolve around validating the energy budgeting system and leveraging the infrastructure to implement power management algorithms.

Throttling the processes via this energy budgeting technique is a mechanism rather than a power management algorithm. In order to meet deadlines and other quality of service objectives, the processes would have to degrade gracefully. This degradation is likely to be application specific, although it could potentially be built into middleware.

Techniques such as voltage scaling and the use of low-power processor modes are complementary and can be used to eliminate idle time and increase the "work" done by the system per joule.

6 Results

Of major concern is the perceived overhead of taking measurements, both in terms of power and time. We have chosen four benchmark programs to discuss the impact of the measurement system on the application:

- *gzip* represents a compression algorithm, which may be used to reduce memory footprint of data — in this case, a 1.4MB MP3 — the output is discarded;
- *mpg123* is an MP3 player operating on the same 1.4MB MP3 file and is representative of a typical multimedia application — the output is discarded;
- *vision* is computer vision software, which uses a low resolution (128x128) greyscale image and identifies the type, location and orientation of an object within the image;
- *celp* is a codebook excited linear predictor and has been adapted from version 3.2 of the US DoD's Federal-Standard-1016 implementation for a lossy speech compression algorithm.

The benchmark applications cover a wide range of embedded applications. Compression algorithms are common to reduce the amount of data to be transmitted over low bandwidth interconnect and field buses. Multimedia applications like the MPEG decoding example mpg123 are common in most 3G phones and PDAs. Similarly the celp example stresses audio compression technique for mobile communication over a low bandwidth carrier. The vision example is typical for software used in industrial manufacturing automation involving simple, low resolution and robust image processing software.

The data presented in Table 1 shows the time overhead introduced by the measurement system with the main processor running at 199MHz, the system bus at 99MHz, and the memory bus at 99MHz. It suggests that the impact of the off-line measurements is not unreasonable. The overhead comes from three sources: the Linux interrupt handling code, the measurement system interrupt handling code, and the associated cache-related costs of running these two. The comparably large impact on `gzip` can be explained by its heavy memory and cache dependency. The Linux interrupt overhead code is much larger than the actual measurement system code. One possibility is that Linux pollutes the caches, badly affecting the working set of `gzip`. Another possibility is that the interrupt code and page mapping are evicted from the cache and TLB when running memory-intensive applications, leading to overhead when the interrupt is triggered.

In the case of the on-line system, the overheads presented in Table 1 are not unreasonable for a prototype system. `gzip` is adversely affected in the same way as in the off-line measurements. The on-line measurement system shows slightly lower overhead than that of the off-line system as a result of its not having to store large amounts of data (a running total, rather than a complete history, is maintained), and smaller interrupt handler code size.

Future versions of the system will make use of the XScale's *fast interrupt queue* (FIQ) vector, avoiding overheads associated with the Linux interrupt code, while at the same time reducing interrupt latency and improving the measurement accuracy because the sampled program counter will be better synchronised with the actual measurement. Furthermore, the interrupt handler could be pinned in the cache and TLB, avoiding the particular effect on cache-intensive applications.

The measurement based monitoring technique may be used for real-time systems, especially when the interrupt is moved to a separate interrupt queue, the TLB entry pinned and handler is locked into cache. In the context of real-time systems the off-line measurements may be obtained at the same time as the traces for a measurement based worst-cases execution-time analysis as in [4]. This would make effective use of the test scenarios created.

It is necessary to take into account the potential latency added by other interrupt service routines. We believe that it is safe to assume that enabling the measurement system does not extend the WCET of any task being analysed. The testing undertaken may well be used in a measurement based WCET approach, as described by Bernat et al [4].

The accuracy of the measurements has been validated by checking:

- **sanity**: for a variety of benchmarks and combinations of benchmarks, measuring the total average current consumed at the input and comparing with the total power given by the measurements;
- **proportionality**: running benchmarks with a consistent power consumption and comparing the ratio between the CPU and memory power for both the output of the measurement system and the voltage presented to the microcontroller by the current sensors;
- **consistency**: running different combinations of benchmarks and checking consistency of the measured results (i.e. in the absence of cache effects or other cross-coupling of the process' power, the measured power should be the same independent of what programs are running concurrently).

In order to measure the system's instantaneous power consumption, it was necessary to use artificial benchmarks which hold the power consumption at a constant level (while that benchmark is running).

- `fillmem` repeatedly fills a 1MB block of memory with meaningless numbers.
- `copymem` repeatedly copies a 1MB block of memory.
- `fp_exercise` performs some CPU intensive operations.
- `mul_bench` repeatedly executes the `mul` instruction.
- `add_bench` repeatedly executes the `add` instruction.

In order to obtain a larger variety of powers to be measured, the frequency of the main processor core was adjusted to 99MHz, 199MHz, and 398MHz. The processor's internal bus was also adjusted according to half the core frequency.

There is some advantage given to the measurement system by keeping the benchmarks at a constant power: bandwidth limitations and mis-alignment of the analogue and digital samples will have less impact. This could be further examined by comparing identical functions run in different programs, or comparing the total energy consumed rather than the power. However, in these cases, the measurement system accuracy will be reduced because of a smaller number of samples. Furthermore, as the size of the entity being measured is reduced, the assumption that the power of the entity is not affected by the surrounding program (or programs) becomes less valid. Lastly, since it is unlikely that the entity would be passed the same input each time, its power would further vary. The latter two are not inaccuracy

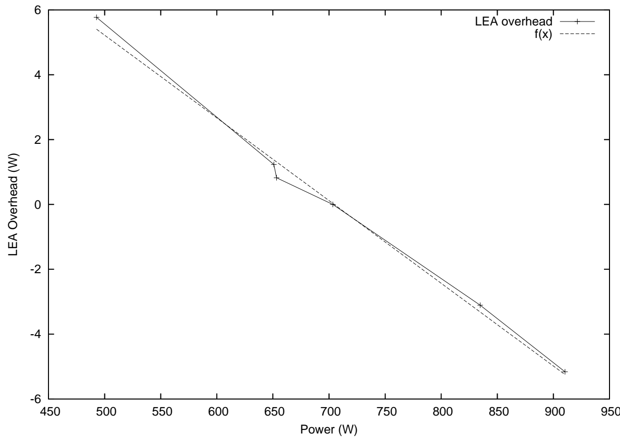


Figure 2. Overhead vs. actual power

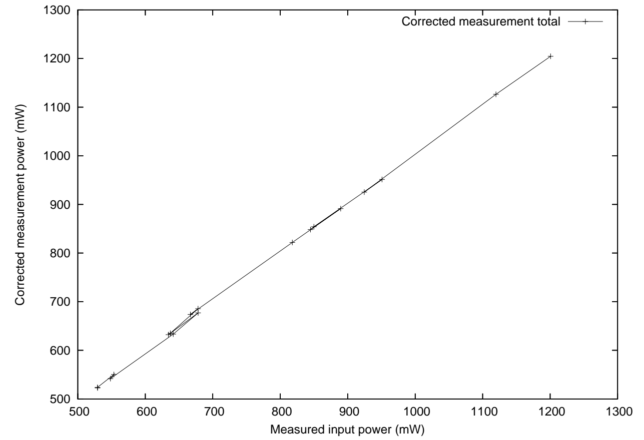


Figure 3. Measured vs. input power

in the measurement system, but a natural variation in the power. Further validation forms part of the future work in this area.

Figure 2 compares the power overhead with the actual system power without the measurement system running. The extra power used by the measurement system is found to be approximately constant, with a similarly constant percentage of the time executing. This is consistent with data, since there will be no change in input power if the measured software is using the same power as the measurement system. (i.e. the power will be constant). Mathematically:

$$P_{meas} = (1 - r)P_{in} + rP_{sys} \quad (1)$$

where P_{meas} is the power when the measurement system is running, P_{in} is the power when its not running, and r and P_{sys} are constants describing the proportion of time running the measurement system and the measurement system power respectively. Fitting this model to the data using least squares, we find that the measurement system uses approximately $C = 700mW$ for $r = 2.7\%$ of the time. This is approximately the same as thees time overheads shown in Table 1.

A sanity check was performed by comparing the input power (obtained by measuring the input current and voltage with two multimeters) with the sum of the (CPU, memory and IO) measurements for a given constant-power benchmark as given by the on-line measurement system. However due to the nature of the circuit the sum of the power consumed by the sum of the CPU, memory and IO power will not equate with the input power. There are several sources of power dissipation which must be considered in order to compare the two sets of measurements:

- The system uses several DC-DC converters to convert from the main supply voltage to the CPU, memory and IO voltages required. These converters have an efficiency which varies with the current and difference in voltage. For simplicity we assume a constant efficiency for each converter.

- A small amount of current consumed on the IO line. While PLEB2 has been designed to allow this line to be monitored, the components to do it were not available. The supply was physically measured using a current meter, and shown to be 20mA.

- There is a linear regulator to supply a clean voltage to the XScale's phase-locked loop, and another to supply the DC-DC converter logic.

- Other power drains are not accounted by the measurement system. Their power dissipation varies with the main supply voltage. We suspect these are resistive loads within the power-supply circuits.

For the purposes of checking the measurements, a simplified model was developed and fit to the measured values.

$$P_{in} = \frac{1}{\eta_{cpu}} P_{cpu} + \frac{1}{\eta_{mem}} P_{mem} + \frac{1}{\eta_{io}} P_{io} + \frac{V_{in}^2}{R_{DCDC}} \quad (2)$$

where η represents efficiency, P represents power, V represents voltage, and R_{DCDC} represents an equivalent resistive load in the DC-DC converter chip. DC-DC converter quiescent power, the power consumed by the linear regulators, and non-linearities in η are ignored for simplicity.

The data was fit to the above model numerically using least squares. The efficiency of the DC-DC converters was found to be approximately $\eta_{cpu} = 86\%$ and $\eta_{mem} = 95\%$.

The rated efficiency in the data sheet for the DC-DC converter circuit is 90% (the figures could be distorted by sources of power loss not considered in the model).

Figure 3 compares the corrected total power against the input power (measured with two multimeters) — i.e. each side of Equation 2. The maximum error between the two for these measured cases is 8mW (1.29%), showing that the measurement system is indeed making sane measurements (much of the error is likely to stem from the simplified model used to compare the input and measured power).

The defining feature of the measurement system is the ability to distinguish between the power used by different pieces of software running concurrently. This ability is proven by running several benchmark processes concurrently (i.e. with the Linux scheduler switching between them). Each benchmark should cause a near constant power draw during its period of execution. Varying the benchmarks which are run concurrently changes the system's average power consumption. The measurement system should be capable of distinguishing the different processes' constant power in each case.

Tables 3 and 4 show consistency between the measurements. Each of six tests was formed via a combination of one or more of the benchmarks and the power measured using the off-line system. It can be seen in the CPU results that the system varies by 2mW(0.6%) for the CPU power, and 9mW(2.6%) for the memory power (for this small sample size). The small variation in memory power can be explained by the increased cache misses caused by running concurrent processes. This effect varies due to cache placement, concurrent cache refills and write-back.

While these measurements presented were performed using the on-line system for convenience, the conclusions should apply to off-line measurements (i.e. the off-line and on-line results should be equal). Table 2 shows typical measurements for the constant-power benchmarks measured using both the off-line and on-line systems for comparison.

In summary, it was shown that the measurement system measurements can be equated with the observed input power. Then, that the measurements are consistent when run with a variety of benchmarks. Lastly we present some samples from both the on-line and off-line systems for comparison. We conclude that the system is useful as a tool for making power measurements.

7 Conclusions and future work

Direct power measurements, correlated with the in-system activity, provide a good way to obtain information to be used in analysing power use in computer systems. The implementation presented has low overheads and provides accurate results. It is easier and neater to use than previous implementations. The system can be used for off-line static analysis, and adds support for on-line accounting and budgeting.

A major advantage of the proposed approach over previous work is the ability to measure from more than one current sensor, allowing accounting for background activity, as well as more detailed information about how power is used in the system. Further advantages include not having a requirement for a detailed system model.

Future work will investigate several ideas:

PLEB 2 was designed as a general purpose research platform, and so was designed with the basic power monitoring features described in the previous sections. Given experience with PLEB 2 and a greater knowledge of the requirements placed on the measurement system, more appropriate hardware could be designed. An FPGA, rather than a microcontroller, could be used to coordinate current measurements. This would allow significantly reduced overheads, since the link via I2C could be replaced by a connection directly to the PXA255's memory bus, speeding and simplifying the data transfer. The FPGA could perform any necessary integrations or scaling.

Instead of measuring the current supplied by each power supply, a current sensor could be installed per device, allowing the system to measure the current consumed. This would mean each IO device would be individually monitored allowing users of the off-line analysis tools to understand how and why each device consumes power as well as allowing the on-line tool to accurately account for background activity.

Energy accounting could be done in the FPGA in order to improve the accuracy and and reduce the measurement system overheads. An FPGA with memory could be informed of a context switch, allowing it to track the power consumed by running processes without interrupting the system.

Applying the techniques discussed would both validate the ideas, and provide useful feedback about the behaviour of a typical system. Two possibilities are: compare a number of proposed dynamic voltage scaling techniques to determine how they perform at a system level (rather than using the CPU-specific power estimations), analysing the power consumption of a range of benchmarks, and integrating with timing analyses.

A more detailed investigation of the sensitivity to sampling frequency and process power variation would also be desirable.

Test	1	2	3	4	5	6
copymem	0.306		0.307	0.308		0.307
fillmem		0.310	0.311	0.311		0.311
fp_exercise				0.315	0.315	0.315
add_bench					0.268	

Table 3. Comparison of CPU power measurements by Powerscope for four benchmarks in six combinations. All measurements in Watts.

Test	1	2	3	4	5	6
copymem	0.347		0.349	0.340		0.340
fillmem		0.405	0.405	0.405		0.397
fp_exercise				0.211, 0.211	0.211, 0.211	0.211
add_bench					0.211, 0.211	

Table 4. Comparison of memory power measurements by Powerscope for four benchmarks in six combinations. All measurements in Watts.

References

[1] Intel PXA250 and PXA210 applications processors developer's manual. http://www.intel.com/design/pca/products/pxa255/techdocs.htm, 2005.

[2] T. A. AlEnawy and H. Aydin. On energy-constrained real-time scheduling. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Catania, Italy, July 2004.

[3] I. Apple Computer. Tools - performance and debugging. http://developer.apple.com/tools/performance/, 2005.

[4] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real–time systems. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, Austin, Texas, USA, Dec. 3–5 2002.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *the proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 83–94, 2000.

[6] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE, Workshop on Mobile Computing Systems and Applications*, 1999.

[7] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. T. Kandemir, T. Li, and L. K. John. Using complete machine simulation for software power estimation: The softwatt approach. In *Proceedings for the 8th International Symposium on High Performance Computer Architecture*, pages 141–150, 2002.

[8] R. Jejurikar and R. Gupta. Optimized slowdown in real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Catania, Italy, July 2004.

[9] L4Ka Team. L4Ka::Pistachio kernel. http://l4ka.org/projects/pistachio/.

[10] T. Simunic, L. Benini, and G. D. Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Design Automation Conference*, pages 867–872, 1999.

[11] A. Sinha and A. Chandrakasan. Jouletrack —a web based tool for software energy profiling. In *Design Automation Conference*, pages 220–225, 2001.

[12] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 1994.

[13] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[14] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems CASES'02*, 2002.

[15] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Currentcy: Unifying policies for resource management. In *Proceedings of the USENIX 2003 Annual Technical Conference*, 2003.

[16] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.