Eighth Real-Time Linux Workshop

Organising committee

Dr. Jiming Wang, CNDS Lab, Peking University, CHINA
Prof. Li Lian (Co-Chair), SISE, Lanzhou University, CHINA
Prof. Nicholas Mc Guire (Co-Chair), Distributed and Embedded Systems Lab, Lanzhou University, CHINA
Dr. Peter Wurmsdobler (Co-Chair), Real Time Linux Foundation, USA
Dr. Qingguo Zhou, Distributed and Embedded Systems Lab, Lanzhou University, CHINA
Prof. Xiaoping Zhang, SISE, Lanzhou University, CHINA
Prof. Zhibing Li, CS, East China Normal University, CHINA

Program committee

Prof. Alfons Crespo, Universidad Politecnica de Valencia, SPAIN
Anthony Skjellum, Mississippi State University, USA
Prof. Dr. Bernhard Zagar, Johannes Kepler Universitt Linz, AUSTRIA
Dr. Chen Maoke, Tsinghua University, CHINA
Dr. Chen Yu, Tsinghua University, CHINA
Prof. Douglas Niehaus, University of Kansas, USA
Prof. Dr. Hermann Hrtig, Fakultt Informatik, Technische Universitt Dresden, GERMANY
Dr. Jaesoon Choi, National Cancer Center, KOREA
Prof. Li Xing (Co-Chair), Tsinghua University, CHINA
Martin Terbuc, Universitz of Maribor, SLOVENIA
Dr. Michael Hohmuth, Technische Universitt Dresden, GERMANY
Prof. Nicholas Mc Guire (Co-Chair), Distributed and Embedded Systems Lab, Lanzhou University, CHINA
Prof. Dr. Paolo Mantegazza, Dipartimento di Ingegneria Aerospaziale, ITALY
Ing. Pavel Pisa, Faculty of Electrical Engineering, Czech Technical University, CZECH REPUBLIC
Dr. Qingguo Zhou, Distributed and Embedded Systems Lab, Lanzhou University, CHINA
Prof. Tei-Wei Kuo, National Taiwan University, Department of Computer Science and Information Engineering,TAIWAN
Prof. Thambipillai Srikanthan, Nanyang Technological University, SINGAPORE
Yoshinori Sato, H8/300 project, JAPAN
Yu Guanghui, Dalian University of Techonolgy, CHINA
Yuqing Lan, China Standard SoftwareCo LTD, CHINA
Dr. Zhang Yunquan, Institute of Software, Chinese Academy of Science, CHINA
Zhengting He, University of Texas, USA
Prof. Zhang Yaonan, Cold and Arid Regions Environmental and Engineering Research Institute, Chinese Academy of Sciences, CHINA

Lanzhou 2006

Preface

After several Real-Time Linux Workshops in Europe (Vienna 1999, Milan 2001, Valencia 2003, and Lille 2005), in the USA (Orlando 2000, Boston 2002), and Asia (Singapore 2004), the Eighth Real-Time Linux Workshop comes to China. The event is still driven by the simple goal: bring together developers and users, present new developments, discuss 'real' user demand and get to know those anonymous people that only exist as e-mail folders on your mailing-list archive, and last but not least, encourage the spirit of a community.

Thank you very much for attending the Real-Time Linux Workshop. We hope that your expectations are met during this workshop, as developer, as user or as newcomer to Real-Time Linux.

The proceedings and all informations on the workshop can be found on

`http://www.realtimelinuxfoundation.org/`

The organisation committee

Acknowledgements

No Real-Time Linux community, no Real-Time Linux users, no Real-Time Linux Workshop. Therefore, our thanks go to the Real-Time Linux community for the work done in open source software development as an international cooperation.

All authors and attendees, thanks a lot for your contribution in any respect.

In particular, we want to express our thanks to (in alphabetical order):

School for Information Science and Engineering (SISE),
Lanzhou University, CHINA

IBM China, Xi'an Branch, CHINA

Haag Embedded Systems, AUSTRIA

Cold and Arid Regions Environmental and Engineering Research Institute,
Chinese Academy of Sciences, CHINA

Dr Dobbs Journal, USA

Consumer Electronics Linux Forum, USA

Last but not least, thanks to everybody having contributed to this workshop and not explicitely mentioned above.

Contents

Ten Years of Research on L4-Based Real-Time Systems

Hermann Härtig and Michael Roitzsch
Department of Computer Science
Technische Universität Dresden
01062 Dresden, Germany
{haertig,mroi}@os.inf.tu-dresden.de

Abstract

Microkernels are an intriguing technology for operating systems research in general and for real-time systems in particular. To gain experience and to explore new ground, the OS research group at Technische Universität Dresden has been developing L4/Fiasco, a real-time implementation of the L4 microkernel specification. Using this kernel, we built an architecture that supports legacy software and provides real-time guarantees. In this paper, we will describe and discuss the design decisions that led us to this architecture. Based on this system, we set out to explore interesting real-time research areas such as networking, disk scheduling and real-time graphics. The results have been published separately, but we will use this article to give a concise overview and present the rationale of our platform strategy as a whole.

1 Introduction

The ideas behind the L4 microkernel were born back in the mid-1990's when Jochen Liedtke reexamined the design of the earlier generation microkernels around Mach. Trying to prove that a minimal kernel can still provide a high system performance, he developed first L3, then L4. The fundamental principle of his microkernels is that a concept will only be allowed inside the kernel, if user-land implementations would be unable to achieve the required functionality. This leads to truly minimalist kernels supporting only address spaces, threads and interprocess communication. These basic services are enough to run isolated user-level processes on top of L4. Any additional functionality must be implemented as a server process. This includes components like file systems, networking and even device drivers, all of which are usually subsumed as an operating system personality.

1.1 Diverse Platforms

Roughly at the same time, multimedia applications were pushing forward into mainstream computing, because the required performance became increasingly available to consumers. The characterising new requirement of those systems was the coexistence of highly dynamic real-time and non-real-time workloads, sharing computer cores, disks, video subsystems and networks. Previously, real-time systems used to be dedicated, having the complete hardware for themselves. Now, both real-time and non-real-time applications are started and stopped side by side at the user's discretion. But although the requirements towards the system changed, the basic architecture of the underlying operating systems stayed the same. Instead, software vendors tried to solve the emerging problems in middleware. We believe this approach is misleading, because no middleware can isolate real-time from non-real-time tasks or reliably enforce resource guarantees for real-time applications without proper core operating system support.

1.2 Resource Throwing

Advancements in computer hardware achieved enormous performance improvements by using caches to exploit the locality of the applications' behavior. However, those techniques are not necessarily useful for real-time systems, because they tend to concentrate on improving the average case, whereas real-time applications must consider the worst case. For the longest time, dedicated real-time systems had been using less powerful, yet expensive specialized hardware to overcome this. On standard computer hardware, these problems were traditionally dealt with either not at all or by spending enormous

amounts of resources. But all those overprovided resources are usually wasted, because the average case behavior is much more benign than the rare but devastating worst case situations, and this gap continues to widen as technology progresses. We intend to solve this problem by dealing with overload situations in ways other than the resource throwing approach. Making the powerful commodity hardware predictable, we can allocate resources much closer to the average case.

1.3 Overview

We do not want to explore our ideas just theoretically, but strive to build a system usable on a daily basis. To this end, we had to consider the maturity and stability of the system. This causes a lot of "unscientific" work, but enforces honesty. Thus, we first present our design decisions for the entire system architecture in Section 2. We include a thorough analysis of the costs caused by this design in Section 2.3. In Section 3, we discuss our solution to handling overload conditions. With those key elements in place, we continue to deal with the management of reservations for resources such as disk, network and graphics bandwidth in Section 4. In Section 5, we conclude by tying all the pieces together into a real-time component architecture that makes the research results readily available for the software development process.

2 Designing the System

A major change in computer systems was the emerging coexistence of real-time and non-real-time applications on the same machine. Today, a large variety of systems has to support a diverse set of such use cases:

- Multimedia applications are used on the average desktop. These applications have immediate real-time requirements, because frames need to be delivered to the display at fixed time intervals. Although deadline misses are not catastrophic, they diminish the user's media experience because they will be visible as motion judder or even frame drops.
  At the same time, non-real-time components may be running next to the player core. For example the media library and subscription management common to today's integrated client applications such as iTunes are clearly non-real-time tasks.
- Off-the-shelf computers are used for sound applications like multitrack editing and live recording of music instruments. Contrasting the media player scenario, enforcing a lower bound on throughput is not the only requirement, but a low latency is needed as well. Music artists can notice delay, if the sound from the speakers is more than 10 ms behind the key hit on the keyboard.
  But arranging the sound in the user interface into multiple tracks, choosing instruments, tweaking the sound and managing media assets has no real-time requirements.
- Mobile phones are being used increasingly for personal information management. Calendar and address book applications are running, as well as games. Sometimes an entire Java virtual machine is used. Alongside, the phone still needs to handle the GSM protocol in a timely manner and once a call is accepted, the speech encoder needs to deliver data from the microphone to the mobile network within certain bandwidth and latency bounds.

From these examples, we can observe that applications with real-time requirements often have small, isolated real-time core functionality surrounded by a large and complex non-real-time part. Current mainstream operating systems do not honor this separation but treat both parts equally, often they are even co-located in the same address space, so proper isolation of resource reservations is impossible. Consequently, current systems have to overprovide resources so that the requirements of the real-time part are satisfied even if it is treated as a best-effort task only.

2.1 Legacy Support

The obvious idea to derive from these findings is that we should reuse an existing operating system personality for the non-real-time parts. This will enable support for legacy software and will ease splitting applications into a real-time and a non-real-time part that can then be treated by the OS differently. Because of its availability in source code and its wide range of application software, the operating system personality of choice is the POSIX personality of the Linux kernel.

One way to implement this is by designing a real-time kernel from the ground up and reimplementing the interface of the legacy kernel in this system. This is the approach taken by the QNX [1] real-time operating system. However, matching the personality of an operating system by reimplementing it is very hard to do, because you are dealing with a moving target. Every change in the semantics of the

legacy kernel must be followed to be fully compatible, which invariably introduces a delay. The state of the reimplementation will always be behind the original, which limits the compatibility.

A better solution is to reuse an existing implementation of the OS personality. This is the approach taken by RTLinux [2], which runs the legacy kernel next to high priority real-time processes on top of a small real-time executive. However, all real-time tasks run along with the real-time executive in kernel mode. The real-time executive is responsible for CPU scheduling and also supports interprocess communication between tasks. But since the legacy kernel is notoriously complex, it is difficult to enforce real-time properties reliably. Furthermore, in the design of RTLinux, there is no isolation between different real-time tasks, which makes it harder to rule out crashes. One real-time task alone can potentially take over the entire system.

Fortunately, with the L4 microkernel technology, we have another option: We can shift the legacy kernel into user space and have it run as an operating system personality server on top of our microkernel. Our implementation of the L4 specification is named Fiasco and it is fully real-time capable, because it guarantees upper bounds for the execution times of all critical operations. Moving Linux to a user space application is possible with only little modifications to the Linux kernel and without disrupting the real-time properties of Fiasco. Consequently, our port of Linux to L4 is called L^{4}Linux [3].

2.2 Real-Time Support

Next to the L^{4}Linux personality for non-real-time tasks, our system provides a real-time personality. Every basic resource such as CPU time and main memory is wrapped by a manager which provides the resource to real-time and non-real-time system components and applications. Using this manager, real-time components like a filesystem can provide an interface with reservations and guarantees for real-time applications and a best-effort interface for L^{4}Linux. This whole architecture on top of our Fiasco microkernel is named the "Dresden Real-time OPerating System", in short DROPS [4].

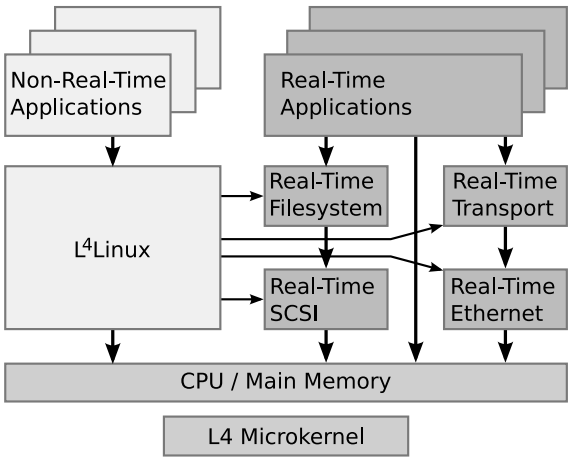


FIGURE 1: *The DROPS Architecture. [4]*

The key advantage of our design over the approach taken by QNX is that we can use the original Linux implementation with only few modifications. Thus, it is a lot easier for us to maintain a current and compatible version of our L^{4}Linux legacy OS personality. And other than RTLinux, our DROPS architecture allows for strong separation of real-time tasks from each other and of real-time tasks from the kernel, because those tasks all run in their own, isolated address space.

2.3 Paying the Price

Unfortunately, the separation of non-real-time tasks, real-time tasks and system servers into their own address spaces comes at a price: Every time an application wants to access a service from another application or server, interprocess communication across address spaces is required and this causes numerous additional context switches. This is one of the major reasons why many people in the OS research community argued that the layer of abstraction provided by a pure microkernel is too low to sustain acceptable performance. However, the non-real-time applications running on top of and therefore heavily communicating with the L^{4}Linux server are a good example to prove otherwise. A performance decrease of a Linux application running on L^{4}Linux instead of native Linux is expected, so we used the AIM multiuser benchmark suite VII to quantify the slowdown [3]. The benchmark tests, how well multiuser systems perform under different application loads. Figure 2 compares monolithic Linux with L^{4}Linux. To compare the performance of Linux on different microkernels, results for an in-kernel and a user-level version of MkLinux, a port of Linux to the Mach microkernel, are also listed. The numbers were obtained in 1997 on a 133 MHz Pentium.

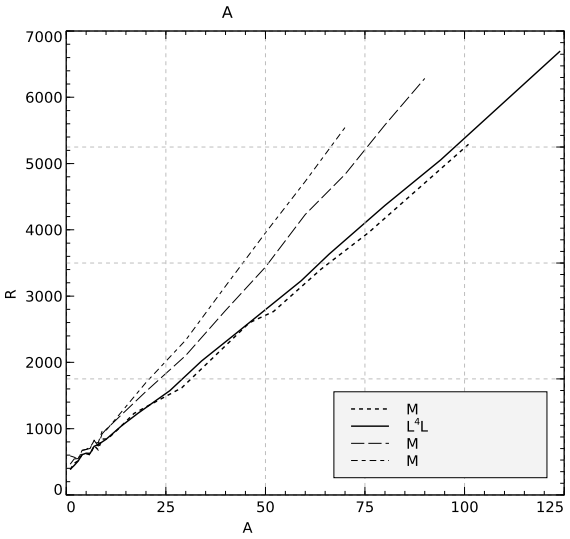


FIGURE 2: *Time per benchmark run depending on AIM load units. [3]*

Averaged over all loads, L⁴Linux is 2.2 % slower than native Linux. User-mode MkLinux is on average 29 % slower than native Linux, the co-located in-kernel version of MkLinux is 21 % slower. This demonstrates that L⁴Linux performs sufficiently close to native Linux, even under high load. Typical penalties range from 2 % to 10 %. The comparison with MkLinux shows, that the performance of the underlying microkernel has a profound influence on the performance of the applications.

One of the goals of our architecture was to ensure guarantees for the real-time tasks even when they are running next to L⁴Linux. Some additional modifications to L⁴Linux are necessary to achieve this [5]: Special measures must be taken to tame the locking mechanism of Linux so it cannot disable interrupts any more. We confirmed the effectiveness of our solution by measuring the actual periodicity of an L4 real-time tasks that requests a 100 ms period from the system. Running standalone on L4, the period length deviates by about 1 to 7 μs. With the untamed L⁴Linux, entire periods are lost because response times increase beyond the 100 ms bound. This is because untamed L⁴Linux can disable interrupts and thus prevent any scheduler activity for arbitrarily long intervals. Our tamed version of L⁴Linux however shows deviations of 24 ms, so the response times do increase compared to a real-time task running standalone on L4, but the periodicity of 100 ms can be supported. These results were obtained in 1998 on the original L4 implementation by Jochen Liedtke and motivated the development of Fiasco as a real-time time microkernel to improve the response times.

To further evaluate the real-time performance of our system, we wanted to compare it against RTLinux. With our DROPS system using a separate address space for each real-time task to increase fault-tolerance, a degradation of response times is expected compared to RTLinux, which runs all real-time tasks as kernel-level threads. To compare both systems, we developed the L4RTL library, which implements the RTLinux API on DROPS and measured interrupt response times on 1.6 GHz Pentium 4 [6]. To actually get worst case behavior, we ensured that caches and TLBs were always cold when an interrupt occurred. The measurements yield a worst case latency of 24 μs on RTLinux and 33 μs on DROPS. This shows that the cost of using address spaces for real-time tasks is not significantly larger than uncertainties introduced by dirty caches or blocked interrupts, which designers of real-time systems seem to accept readily.

We summarize that modifying Linux to run on top of our L4 microkernel Fiasco allows a performance close to native Linux without compromising the real-time properties of the system. Using separate address spaces for real-time tasks increases the fault-tolerance of the system without a significant impact on response times. This allows running real-time and non-real-time applications side by side, which we believe to be a key feature for today's computing requirements. With the presented system architecture, we can also support tasks that want to use both real-time and non-real-time services. Those hybrid tasks can communicate with the L⁴Linux part and with the real-time resource managers.

2.4 IPC Scheduling

Because a lot of system services are separated into dedicated servers, clients are required to communicate often. This drives the need for component interaction that is both fast and predictable. Such interaction is performed on L4 via synchronous interprocess communication (IPC). A successful message transfer requires a rendezvous between the sending and the receiving thread. However, synchronous message passing also introduces dependencies among components. Combined with fixed-priority scheduling of threads, this can lead to the infamous priority-inversion problem:

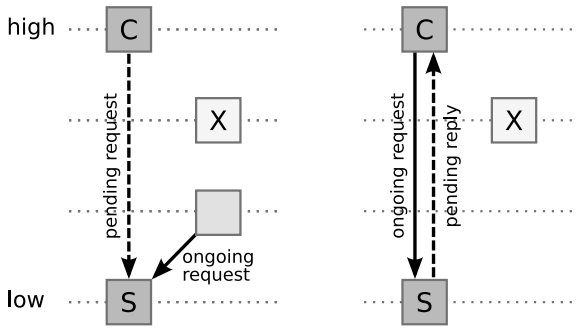


FIGURE 3: *Priority inversion during message passing. [7]*

The example shows a high-priority client C requesting a service from a low-priority server S. Because it is currently engaged in communication with another client, S cannot respond immediately. Before S can answer C, it is preempted by a medium-priority thread X, which leads to the high-priority client C waiting for X although X's priority is lower and although C has no communication relationship to X. Well known methods to avoid such a priority inversion are priority inheritance and stack-based priority ceiling. The basic message passing mechanism of our system must support those methods without sacrificing performance.

Our approach is to divide the thread context that is held in the kernel into two separate contexts:

- an execution context that keeps track of saved CPU registers and the thread state and
- a scheduling context which represents a time quantum coupled with a priority.

Thanks to this separation, the kernel can switch both contexts independently, which enables the implementation of a time donation scheme named capacity-reserve donation from one thread to another [7]: The kernel tracks references to the current execution context and the current scheduling context. On regular thread switches, the kernel will switch both contexts. However, when a request is sent from a client to a server thread, the kernel switches only the execution context, but keeps the client's scheduling context. In doing so, the client effectively donates its time quantum and priority to the server for the time the server executes the request on behalf of the client. When the server replies, the kernel switches back to the client's execution context and the client reobtains its donated scheduling context. If a server needs to contact another server to fulfill a request, the scheduling context donation is transitive. This scheme allows for fast message passing, because scheduling is eliminated from the critical path by directly switching from the sender to the receiver of a message with no priority changes. The consumed CPU time is always correctly accounted to the client that originated the request.

With capacity-reserve donation, the priority inheritance and stack-based priority ceiling protocols can be implemented efficiently. This elegantly solves the priority inversion problem and makes component interaction more predictable, while preserving the high performance of the interprocess communication primitive of the underlying microkernel.

3 Probabilistic Scheduling

As the basic architecture of the system is now in place, it is time to consider resource scheduling in real-time systems. The primary resource real-time system designers focus on is CPU time, because it provides the basis for all subsequent resource accesses. Scheduling the CPU is all about timeliness: Real-time applications provide a deadline and require sufficient CPU time to finish their job before the deadline is reached. The system scheduler must guarantee this property to all real-time tasks it admitted to run. However, the key problem here is how to determine, what "sufficient CPU time" means. Today's hardware makes a lot of effort to speedup applications in the average case by using caches to exploit locality in the application's behavior. Unfortunately, this widens the gap between the average case and the worst case time consumption. In addition, a common real-time application in desktop computing is video playback, which per se does not have a fixed execution time per job.

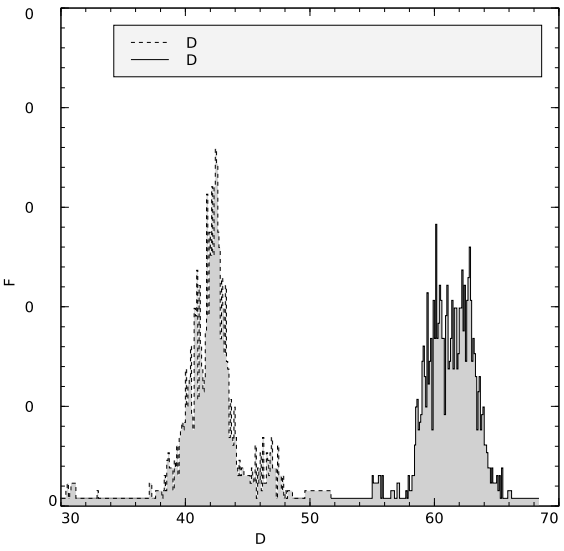


FIGURE 4: *Measured distribution of total decoding times per group of pictures. [8]*

These two factors cause execution time distributions to have a long tail. If the CPU resource was always allocated for the worst case, the system's utilization would be very low, because only few jobs can be admitted and large amounts of resources would be wasted. But media applications are an example for a class of real-time tasks that can tolerate occasional deadline misses, if this does not happen too frequently. With this observation, we devised a system that can handle overload predictably without dedicating enormous amounts of resources [8]. Our idea is to allow a percentage of deadlines to be missed; applications can configure this percentage as a quality level. Resource reservation is based on the distribution of the execution time instead of just the worst case value. Competing approaches in this area such as Imprecise Computation [9] and Statistic Rate Monotonic Scheduling [10] are either based on deterministic duration of resource usage or cannot guarantee a desired quality.

The task model of our Quality-Assuring Scheduling (QAS) allows each real-time task to be split into mandatory and optional parts. The mandatory parts are always guaranteed to be executed before their respective deadline, so worst case reservation is performed. Of the optional parts only the percentage requested with the quality parameter is guaranteed to be completely executed before the deadline. For these parts, admission and reservation is performed using the distribution of the execution time. Caused by the typical long tail of these distributions, even requested qualities only slightly below 100 % cause considerably less resources to be reserved, which greatly increases the overall utilization of the system. Although the scheduling is probabilistic, the requested quality levels are matched quite accurately, as the following table proves. The results were obtained for MPEG decoding with the I- and P-frames as mandatory parts and the B-frames as optional parts with the given quality.

Requested Quality	Reservation Time for Optional Parts	Achieved Quality
0.95	55 ms	0.9506
0.90	53 ms	0.8588
0.80	47 ms	0.7875
0.70	39 ms	0.6740
0.60	32 ms	0.5804
0.40	23 ms	0.4063
0.20	9 ms	0.2451

TABLE 1: *Requested quality, derived reservation time and measured quality of the optional parts. [8]*

Unfortunately QAS' applicability is limited because it only handles periodic tasks with uniform and harmonic periods and the admission is expensive, especially when the distributions are to be calculated with a high resolution. The model can handle arbitrary periods as well, but then the admission cost is increased beyond practical applicability. Therefore, the designated successor of QAS is QRMS, the Quality-Rate-Monotonic Scheduling. It simplifies QAS by assigning the mandatory and optional parts a unified reservation time, which is regarded as constant in the admission control. Thus, QRMS ignores situations where jobs do not completely consume their reservation. QRMS is therefore more pessimistic than QAS, but has a tremendously simpler admission even for arbitrary periods. The results convince of the accuracy and feasibility of the model:

Requested Quality	Quality Achieved with QAS	Quality Achieved with QRMS
0.70	0.7001	0.7024
0.50	0.5019	0.6742
0.7323	0.7326	0.7324

TABLE 2: *Requested and achieved quality of QAS and QRMS for a task system with three concurrent tasks.*

Another interesting property of both QAS and QRMS is that applications can be notified when the optional parts overrun their deadline. This way, an application can react, for example by reducing quality:

```
set_period(period);
reserve_time(mand_time, mand_priority);
reserve_time(opt_time, opt_priority);
do {
  begin_period();
  try {
    do_something();
  } catch {
  exceeded:
    adjust_quality();
  }
  next_reservation();
  try {
    do_something_else();
  } catch {
  exceeded:
    discard_result();
  }
} while (!end);
end_period();
```

With these unified admission and scheduling schemes, we can give probabilistic guarantees for periodic real-time tasks. Considering the variation of execution times allows us to admit far more applications and thus achieve better resource utilization than in systems based on worst-case admission.

4 Resource Management

The previous chapter dealt with scheduling the CPU, but this is not the only resource a real-time application might need. In our DROPS system architecture, all resources used concurrently by multiple tasks must be encapsulated and scheduled by a resource manager running as a server in user-land. In the following, the design of such managers for resources like disk, network and graphics bandwidth is presented.

4.1 Disk Requests

Disk usage in modern systems combines traditional best-effort file access with storage and retrieval of real-time streams, such as audio and video data. The former has relatively weak requirements while the latter must meet deadlines for individual disk requests. For good overall performance, the disk-request scheduler has to optimize the disk utilization as well. This is challenging, because the construction of disk drives causes a poor ratio of average and worst-case execution times. However, the same idea that has been successfully applied to CPU scheduling as discussed in the previous section also helps here: If an application can tolerate occasional deadline misses, probabilistic service guarantees can substantially improve the disk utilization compared to guarantees based on the worst case [11].

Thus, the basic idea is again to split real-time disk requests into mandatory and optional requests and to assign a quality parameter to the optional requests, which denotes the percentage of requests that must be completed. To optimize utilization, requests should be scheduled with the SATF (shortest access time first) algorithm, which is aware of the position of the drive's head on the disk. However, this scheduler does not know anything about deadlines. But instead of implementing a new scheduler, we devised a method to decouple the scheduling of the disk requests from the deadline and reservation enforcement [11]: The Dynamic Active Subset (DAS) always includes all pending requests that can be executed in any order without violating any deadline or reservation. This subset of disk requests is recalculated after every request completion and if enough time is available, the set even includes non-real-time requests to increase utilization. The SATF scheduler or any other scheduler can be run on this set to pick the request to execute next without having to know about deadlines.

With this technology, the disk request scheduler matches the desired quality levels of the tasks.

Bandwidth	Requested Quality	Achieved Quality	Achieved Bandwidth
640 KB/s	0.99	0.9973	638.54 KB/s
2560 KB/s	0.95	0.9798	2509.12 KB/s
1280 KB/s	0.90	0.9444	1209.36 KB/s
640 KB/s	0.85	0.9004	576.44 KB/s
1280 KB/s	0.60	0.6705	858.65 KB/s

TABLE 3: *Requested and achieved quality for a disk (IBM Ultrastar 36Z15) loaded with five concurrent streams. [11]*

Even quality levels only slightly below 100 % push the disk utilization close to the peak best-effort bandwidth.

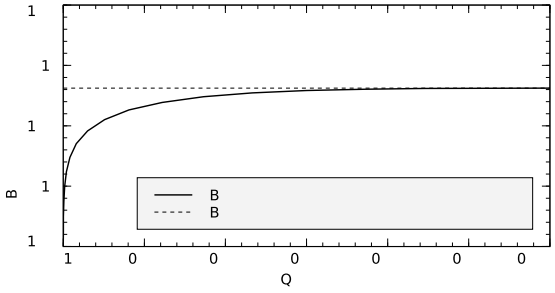


FIGURE 5: *Bandwidth that can be assigned to an optional stream. [11]*

4.2 Ethernet Transmission Delays

With the deployment of switches, Ethernet as the most widely used commodity network becomes interesting for real-time communication. Each port of a switch provides its own collision domain, so collisions do not occur in a star topology network. However, switches generally lack traffic policy features. Thus, if too many Ethernet frames are being sent to a machine that does not receive them fast enough, the switch will enqueue the frames internally, which causes transmission delays. If the internal queueing storage of the switch is depleted, it will even drop frames.

A mathematical model of the network traffic can be used to predict the buffer fill levels in the switch. If the nodes within the network cooperate, this model can be used to parametrize a traffic shaper running on each node that keeps buffer lengths and thus transmission delays within specified bounds [12].

Shaping Interval	Buffer Bound	Calculated Max. Delay	Observed Max. Delay
10 ms	111.8 KB	9357 μs	8759 μs
1 ms	15.7 KB	1380 μs	1300 μs
100 μs	6.1 KB	582 μs	438 μs

TABLE 4: *Buffer bounds in the switch and transmission delay bounds. [12]*

The achievable delay bound mainly depends on the granularity of the traffic shaping. This results in a trade-off between delay bound and CPU load.

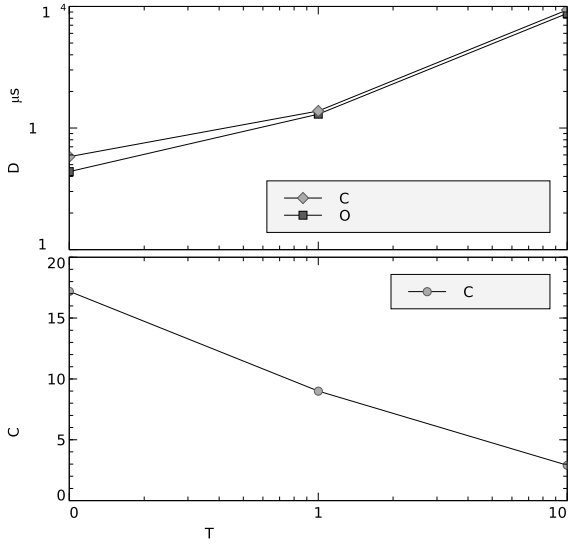


FIGURE 6: *Delay bound / CPU load trade-off. [12]*

Because all nodes on the network must cooperate to ensure the guaranteed delay bounds, each node must run an instance of the traffic shaper. However, not all nodes must run a real-time operating system. The shaping capabilities of the machines influence the delay bound, but we successfully shared the network with Linux machines while still observing predictable delays.

4.3 Screen Real Estate and Screen Updates

Today's modern desktops feature a graphical user interface. Furthermore, modern real-time applications like media players also feature a graphical output. This drives the need for a real-time capable window manager that can provide guaranteed redrawing rates for the real-time windows while providing best-effort services to the remaining non-real-time windows and auxiliary operations such as the user reordering windows. Therefore, the design goal of our DOpE (Desktop Operating Environment) window server [13] was to multiplex the singleton resource of physical screen real estate to client applications. For real-time clients, quality of service is guaranteed even in overload situations, which can be caused by massive screen updates of non-real-time applications. DOpE can therefore sustain real-time client windows running next to L^{4}Linux and X11 on the same desktop.

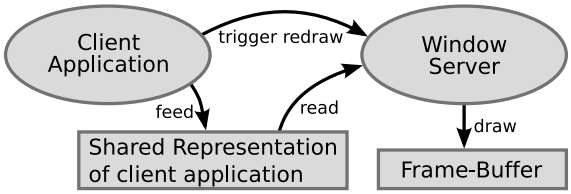


FIGURE 7: *Design of the DOpE window server. [13]*

The architecture of DOpE separates the client's updates to the user interface from the server updates of the representation on screen. The client and the server share a description of the layout and content of the client's user interface. This allows the client to update the shared description without interference of the server and then trigger a redraw operation. The server can then interpret the shared window description and perform the necessary updates to the on-screen representation independently of the client. Because the execution time of such a redraw is known beforehand, the window server can guarantee previously negotiated refresh rates to admitted real-time clients. A real-time client can subscribe to periodic notifications of completed redraw operations. Updating the shared representation in a timely manner is entirely the responsibility of the client.

This separation of cause and execution of redraw requests allows us to display real-time graphics and windows of non-real-time clients seamlessly side by side.

4.4 Second-Level Cache

One easily overlooked resource used concurrently by real-time and non-real-time tasks are CPU caches. They are an especially interesting resource for real-time applications, because every task switch potentially disrupts cache working sets and thus makes execution times unpredictable. To avoid this, the CPU caches should be managed like all the other resources discussed above to isolate the real-time tasks from cache interference by other tasks or the operating system. A well-known solution for this problem is cache partitioning: portions of the cache are dedicated exclusively to specific applications. For our system, we developed a cache partitioning technique that operates without any hardware modifications [14].

A page size of 2^p divides the cache in banks of 2^p bytes, if the cache is direct-mapped. The least significant p bits are used to index an element within such a bank. Assuming a cache size of 2^c, the next $c-p$ bits in the address select the cache bank. The remaining part of the address is compared against the tag. For

an n-way set-associative cache, a cache size of $n2^c$ and a bank size of $n2^p$ are to be used. The division of the cache into banks also divides the main memory into classes, whose physical page frames all fall into the same cache bank. Those classes are called colors. Cache conflicts can only occur between page frames of the same color, so such conflicts can be avoided between any two tasks, if both tasks use disjoint colors. Since the L4 microkernel allows user level memory management, the mapping of physical to virtual addresses can be controlled by a memory server that assigns colors to tasks exclusively.

The problems with this approach are: Being based on the mapping of pages, it can only be applied to physically-indexed caches and only with page granularity. Additionally, if a certain percentage of the cache is to be dedicated to a task, the same percentage of the main memory is implicitly reserved for that task as well. On the other hand, the technique provides a way to close the gap between the average case and the worst case execution time for real-time tasks. This greatly helps when scheduling real-time tasks with hard deadlines, because less CPU resources need to be reserved.

5 Conclusion

With the Fiasco real-time L4 microkernel and the real-time enabled managers for various system resource, the DROPS system described in the previous sections provides all the building blocks for writing real-time applications. Legacy support for running non-real-time software and real-time tasks side by side is provided by the L^4Linux server. The Quality Assuring Scheduling and Quality-Rate-Monotonic Scheduling provide the mathematical foundation to handle overload situations

5.1 Real-Time in Software Development

However, what is still missing is a comprehensive way to open this technology to software developers. All the elegant solutions and advancements in real-time systems research are of limited use, if they are not accessible to the engineers in need. To this end, a joint team of members from our research group and from the software technology group of our department developed the COMQUAD component architecture. This architecture allows to specify non-functional properties like quality levels and resource usage of a component implementation in the component quality modelling language (CQML+). These properties are then used to derive contracts between components which are translated by the component runtime environment into resource reservations. Our real-time operating system and its resource managers enforce these reservations at runtime. This way, a component-based software development process was created, that supports adaptive real-time systems from specification all the way to the running system [15].

5.2 Current State and Outlook

Most of the software discussed here is available for download [16] under the terms of the GNU General Public License. The resource managers are still prototypes, but the foundation of the system is usable. A demo CD is available as well [17]. We hope to spark a wider interest amongst operating system enthusiasts for design, implementation and deployment of real-time systems on everyday computers. With the knowledge we gained from the DROPS architecture, we are currently exploring new ground in the areas of virtualization and hierarchical system design. We expect to present more fascinating research results and we will always ensure that our systems are designed to be useful beyond mere academic purposes.

Acknowledgements

The authors would like to thank the researchers who contributed all the findings that went into this overview paper, including but not limited to (list in alphabetical order): Ronald Aigner, Robert Baumgartl, Martin Borriss, Norman Feske, Claude-Joachim Hamann, Michael Hohmuth, Jochen Liedtke, Jork Löser, Frank Mehnert, Martin Pohlack, Lars Reuther, Sebastian Schönberg, Udo Steinberg, and Jean Wolter.

References

[1] D. Hildebrand: An architectural overview of QNX. In *1st USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Seattle, Washington, April 1992.

[2] V. Yodaiken, M. Barabanov: A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, California, January 1997. The USENIX Association.

[3] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, J. Wolter: The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[4] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, J. Wolter: DROPS: OS Support for Distributed Multimedia Applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[5] H. Härtig, M. Hohmuth, J. Wolter: Taming Linux. In *5th Annual Australasian Conference on Parallel And Real-Time Systems (PART)*, Adelaide, Australia, September 1998.

[6] F. Mehnert, M. Hohmuth, H. Härtig: Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 124–133, Austin, Texas, December 2002.

[7] U. Steinberg, J. Wolter, H. Härtig: Fast Component Interaction for Real-Time Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, Palma de Mallorca, Balearic Islands, Spain, July 2005.

[8] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, H. Härtig: Quality-Assuring Scheduling – Using Stochastic Behavior to Improve Resource Utilization. In *Proceedings of the 22th IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[9] K. J. Lin, S. Natarajan, J. W. S. Liu: Imprecise results: Utilizing partial computations in real-time systems. In *Proceedings of the IEEE Real-Time System Symposium*, 1987.

[10] A. Atlas, A. Bestavros: Statistical Rate Monotonic Scheduling. *Technical Report 98-010*, Boston University, May 1998.

[11] L. Reuther, M. Pohlack: Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS). In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS),* Cancun, Mexico, December 2003.

[12] J. Löser, H. Härtig: Low-latency Hard Real-Time Communication over Switched Ethernet. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, Catania, Italy, June 2004.

[13] N. Feske, H. Härtig: DOpE – a Window Server for Real-Time and Embedded Systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December 2003.

[14] J. Liedtke, H. Härtig, M. Hohmuth: OS-Controlled Cache Predictability for Real-Time Systems. In *Proceedings of the Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[15] H. Härtig, S. Zschaler, M. Pohlack, R. Aigner, S. Göbel, C. Pohl, S. Röttger: Enforceable Component-Based Realtime Contracts – Supporting Realtime Properties from Software Development to Execution. To appear in the *Real-time Systems Journal*.

[16] `http://os.inf.tu-dresden.de/drops/`

[17] `http://demo.tudos.org/`

The Data Platform of Field Observation Stations Base on Grid Environment

Zhang Yaonan[a,b], Zhang Baoshan[a,b], Lu Yu[a,b], Gao Meirong[c], Kang Jianfang[a,b], and Zhao Xueru[a,b]
[a]Cold and Arid Regions Environmental and Engineering Research Institute, CAS
320 Donggang West Road Lanzhou, Gansu 730000, China
[b]Gansu High Performance & Grid Computing Center
Lanzhou, Gansu 730000, China
[c] Institute of Mountain Hazards and Environment, CAS
#.9, Block 4 , Renmingman Road, Chengdu, 610041, China
{yaonan,skytiger}@lzb.ac.cn

Abstract

The Data Platform of Resource and Environmental Science is a foundation of data and model simulation for geography research, and the data mainly comes from several field observation stations in distributed location, spatial observation data such as MODIS and some data service site such as USGS of internet. The data collection, management, assimilation and the service of share is a main task of the Data Platform. According to the distributed characteristic of the data source dispersion and the service object and new technology of the Grid environment, in this paper we discuss the foundation of the Data Platform of the field observation stations' Resource and Environmental Science, and the process framework of the data gathers with the management, and defined and analyzed the key functions and implementary techniques for each module. Base on simulated the environment of the field observation stations in Linux operation system, we validate some functionality test about the key techniques and pathway for data replication, data synchronization and a uniform data service in data platform, and using acquired some techniques to build a primary Data Platform of the field observation stations' Resource and Environmental Science on Grid environment.

1 Introduction

In the research of the resource and environment , there have been built a lot of field observation and have accumulated a large amount of observation data, the management and use of there data is very complicated because of these data are dispersed in different field observation stations. So, to build a platform, which can integrate the data of all field observation stations it is necessary for easy and effective to access, obtain, share and use these data , and to reduce the access difficulty and increase the using efficiency of these data[1]. As a result, the propose of the resource and environmental Data Platform has offered the solution to achieve above mention targets, and defined a series of standard of data of the field observation station, a metadata standard of data attribute of management, some integrated method of geographical information, and a lot of technology and standard which needed, such as accessing, processing, storing, distributing of these data information ,etc.

Because of the geographical distribution, real-time, variety, long-term, mass of the field observation data, and the diversity of data management system which build in every field observation station, the platform which will be built must have stability, reliability, extensibility and transplantable. Based on new technology that can be adopted and our experience of managing data in the past, and the data management system that already have been build in every field observation station at present. The choice of Grid technology and metadata technology is a reasonable way to build a Resource and Environmental Data Platform in network to solve integration problem of spatial distributed data.

Grid can through high-speed Internet to integrated computing resources that distributed in different geographical position, to obtain an ability to offer a kind of high-performance data management and information service. Using Grid technology to build a resource and environmental Data Platform can effectively stored and managed the observation data that distributed in a lot field observation stations, and can provide service for researchers through the internet in the way of reduce the acquisition cost and improve the utilization of data.

Metadata generally considered to be description for data, is about data of datasets. In the solution of the Resource and Environmental Data Platform, metadata describe from one data method in index expand to indispensable tool and one of the methods in the whole information management that including data discovery, data transition, data management and data use, it is one of the key technique of the Resource and Environmental Data Platforms[2].

Based on Linux, Grid technology, Oracle technology and metadata technology, in this paper we discussed the architecture of resource environmental Data Platform and framework of data gathered and managed which stored in the field observation station in different geographical position, defined and analyzed the key functions and implementary techniques for each module. Base on simulated the environment of the field observation stations in Linux operation system, we validate some functionality test about the key techniques and pathway for data replication, data synchronization and a uniform data service in Data Platform, and have acquired some key techniques to build a primary Data Platform of the field observation stations' Resource and Environmental Science on Grid environment.

2 Framework analysis of the Data Platform

2.1 System structure of the Data Platform

While designing the Data Platform system, we have refered five layers of sand filters model of Globus and model structure of Web Service, according to the architecture of the dataGrid, we divide the Resource and Environmental Data Platform into three part as Fig. 1 show: server layer, middleware layer, interface layer.

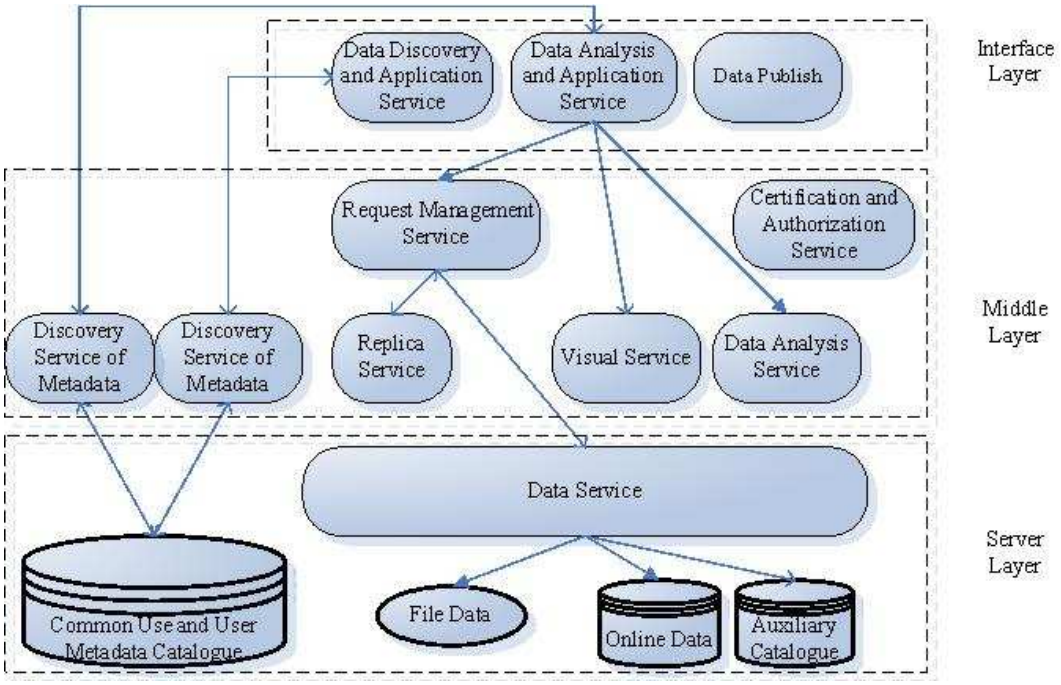


FIGURE 1: *The sketch map of the Architecture of the Data Platform*

In architecture of the Data Platform above Fig. 1, among them the server layer offers data service mainly, including file data, local data, online data, auxiliary catalogue of data, in common use and user metadata resource type relevant services of data. The focal point of middleware layers offers discovery service of metadata, data replica service, metadata replica service, request management service, certification and authorization service, visual service, data analysis service. Interface layer mainly offers the data release, the analysis of the data application, data discovery and application service.

2.2 Data Discovery Service

In the Resource and Environmental Data Platform based on the Grid environment, the data may be stored in the standing memory equipment in different field observation stations. How to access these data located in a given position is one of the key questions to this Data Platform. Here, we offer unanimous data view of application program with the storage system abstract and Grid API, and access these data through unanimous way.

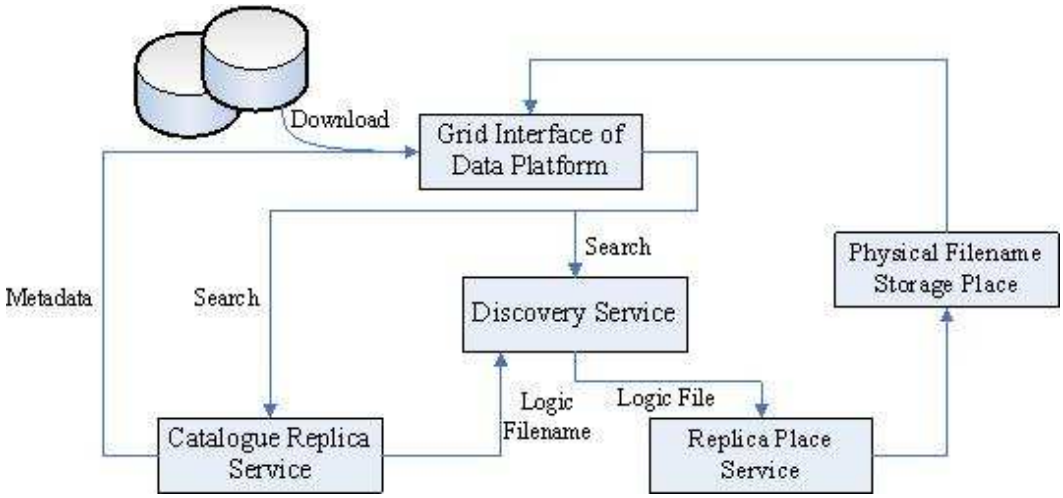


FIGURE 2: *The Process of Data Discovery*

The process of data discovery such as Fig. 2 shown, Through API of the Grid storage and Grid interface of Data Platform, we can locate the file instance that store in the field observation station through searching, and Grid storage API carries on remote reading, writing and looking over the operation of attribute to these file instances, and through storage server of file instance setting up the third party's transfer control to transfer the instance of the file between different memory systems.

2.3 Data Replica Service

The data stored in the Data Platform are distributed, the need data sometimes are distributed storage in the node of different field observation station, if we carry on the set of the data from each node sequentially each time, systematic overhead increases, and it is unfavorable to the use of the data. In the platform, we will according with certain condition collect data into a dataset using Grid service and data replica service function which have reduced the time accessing data, improved the efficiency of accesss. With regard to granularity of data replica, regard file as the minimum granularity of replica. And relevant metadata and all copy of associated documents belong to an entity of storehouse for metadata to, together with instance of the file display and become a logic structure called "logic file". In the data replica serving, we get the relevant logic file together, called replica catalogs.

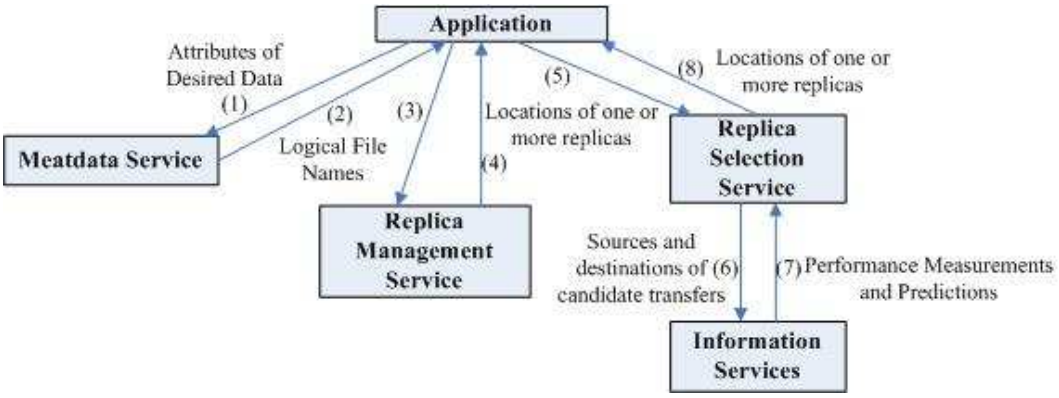


FIGURE 3: *The Process of Data Replica*

The process of data replica such as Fig. 3 shown , data replica service from the gather of the metadata service to the localization of replica management service, and then reach the replica selection service, reach information service finally, the middle needs passing 8 steps on Fig. 3, it is separately (1)Attributes of Desired Data,(2,3)Logical File Names, (4,5)Locations of one or more replicas,(6)Sources and destinations of candidate transfers,(7)Performance Measurements and Predictions,(8)Locations of one or more replicas

2.4 Management Model of Metadata

In resource environmental Data Platform, we use a index structure is it construct system of metadata to come. Utilize the index structure, can solve the operation for metadata under the Grid environment, conflict that may appear clear and empty, make metadata keep unanimity. Fig. 4 is the model sketch map of metadata management and service.

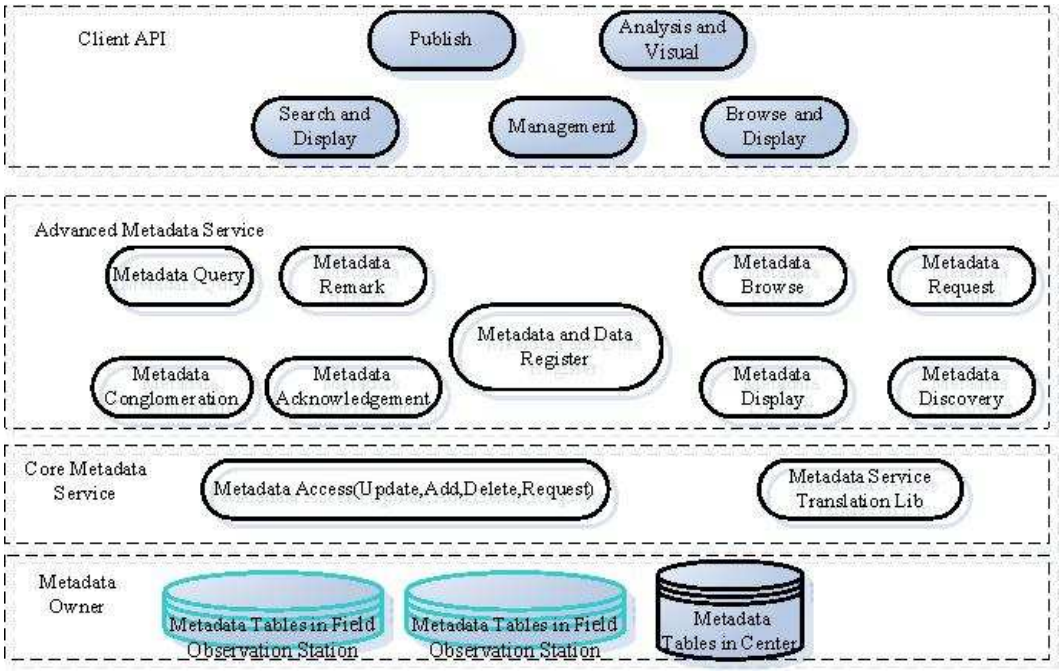


FIGURE 4: *The Process of Data Replica*

As Fig. 4 shown, metadata service is formed as hierarchical structure, which can be used to solve the distributed high-efficient complexity searches questions of index in the Grid through the data organization. A naming structure and data model is structured to support the distributed environmental level of Grid. The information of the catalogue can be organized, replica and distributed in this kind of model structure with catalogue hierarchical structure of LDAP. It can supply metadata service as many way for application and mechanism to keep metadata unanimous without to consider how the data stored and where the data deposit in.

2.5 Overall Analysis

The Data Platform system is mainly divided into a date node parts and a data center, the main function of the data center is to coordinate the information between the data nodes and user's authority , and collect all data with data nodes, offer information service to the outside at the same time. The data node is responsible for appointing to work in the arrangement of this nodal data and user's authority of access this kind of data mainly, offer information service to the outside too at the same time. The relation between them is as Fig. 5 shows.

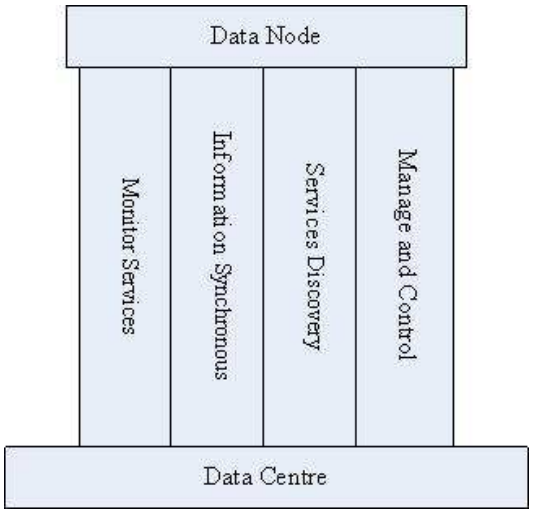


FIGURE 5: *Relation between the data center and data nodes*

As Fig. 5 shown several important relations between the data centre and data nodes:

1. Date center can monitor and control the work on the data nodes through various kinds of monitor services at the data nodes. The services include the monitor of the netflow, the data of users query, running situation of every service etc.

2. The data nodes transfer the data to the data centre through the synchronous function of information.

3. Data node can achieve the service discovery through data center, thus make clients know whether data information services exist and how to intercommunicate with each other.

4. The configuration information of the data nodes can be adjust by the data center.

2.6 Module Functions Description

The modules of the Data Platform are mainly constitutes by user administration system, information service system, metadata management system and data management system, its function is described as follows.
User administration system is mainly responsible for managing users' information, authority and synchronization of users' information in the platform system.
Information service system, responsible for offering information service to the outside mainly, including metadata information service and data information service, as well as the data and the metadata discovery service and analyse service.
Metadata management system, is mainly used to the metadata that use in the Data Platform, for example, the management, the upgrade, the move, the synchronization, the register and the backup of the metadata.
Data management system, is mainly responsible for carrying on effective management, synchronism and backing up the job to the data that are stored in the database.

3 System Realization

3.1 System Environment

1. The operating system adopts Redhat Linux AS 4.0[3] which includes stability, high performance and good security.

2. Web server we choose Tomcat which includes Servlet API 2.2 and JSP 1.1 compatible Servlet/JSP container, it is a swift and high-efficient jsp/servlets operation platform.

3. The foundation Grid chooses Globus Toolkit[4] basic platform which supports open source code. Globus Toolkit is one software library of open structure, opens service resources, and support the Grid and Grid application. The purpose is to supply middleware service and

program library for structuring Grid application.

4. Database selection Orecal10g[5] which has been added in the function of support Grid.
5. The Grid stores uses GridFTP to expand API interface, to guarantee data safe and implement the data parallel transfer in third party's data transmission.
6. For data replica tools, we select GDMP[6] that supports subscription.

3.2 The Core Realized Technology

3.2.1 Data Synchronization

The data stored in the data nodes include relational one and non-relational one two types at present, we can use rsync[7] +ssh shell[8] in linux system combined together to realize data synchronization for non-relational data files. The non-relational data files in data nodes will be mirror to data centre by using secure certification mechanism of SSH Shell and rsync transfer method. In actual operation, when we configured rsync and ssh service, it can timing automatically synchronizes the data between data dodes and data center with crontab in linux system but needn't artificial participation.
The relational data synchronization between data node and data center can be done by using advanced replica function in Oracle.

3.2.2 Data Replica Service

Using GDMP (Grid Data Mirroring Package) tool to implement data replica, and the process of data replica can be carried by the course of data replica through GDMP: gdmp_register_local_file, gdmp_publish_catalogue and gdmp_replicate_get; Then the catalogue management of the data replica through HTTP Redirection. And the synchronization task of data replica through file replacement, increased upgrading and log management; Finally in choice of the copy, the cost model we established must also take into consideration of the load of network and server where data is on.

3.2.3 Realization of Procedure Storing Data in Database

Fast effective input the observation data into database come from observation instrument of some field observation stations, is an important problem in this system. we have tried using JAVA to structure one import tool in this scheme to realize import initial data file into Oracle database. This process is that got standardized data from instrument and save it as a data file with fix format, then use this import tool to connect with Oracle database, import data files into designated form to implement insertion process of the data.

4 Experiments

4.1 Experiment Environments

We have set 3 machine in Internet environment to simulate environment of field observation stations. One machine simulation a date centre and other two machines simulation the data nodes of Node-A and Node-B of field observation stations. The software configure as showing in table 1.

	Center	Node-A	Node-B
Operating System	Redhat Linux AS 4.0	Redhat Linux AS 4.0	Redhat Linux AS 4.0
Grid Support	Globus Toolkit 4.0.3	Globus Toolkit 4.0.3	Globus Toolkit 4.0.3
Database	Oracle10g	Oracle10g	Oracle10g
GDMP	v3.2.6	v3.2.6	v3.2.6
IP	210.77.67.xx	210.77.68.xx	210.77.69.xx

TABLE 1: *Experiment Environments*

4.2 Experiment Tests

4.2.1 Data Replica

Using replication of time serial data of the groundwater water level as an example to test replication functional of data. The data file of the water level of Lanzhou of Gansu province from January of 2000 to May of 2000 stored in Node-A, The data file of the water level of Yuzhong of Gansu province from June of 2000 to December of 2000 stored in Node-B. Lanzhou and Yuzhong we can think as same place at large-dimension, the purpose of this experiment is that to replica the data of these 2 parts to form an integrated time serial dataset file of from January of 2000 to December of 2000 of the water level of the region of Lanzhou of Gansu of 2000 year in Node-B. After we install and configure GDMP 3.2.6 in Node-A and run that:

```
gdmp_register_local_file -d /var/filedb
```

to the regist a catalogue file on Node-A, then run:

```
gdmp_publish_catalogue
```

on Node-A to publish catalogues of data files of Node-A, and finally operate:

```
gdmp_replicate_get
```

in Node-B to replica data from Node-A to Node-B.

4.2.2 Data Synchronization

The purpose of this experiment is to test synchronization functional of data located in different place for relation data of database and non-relation type data in storage.

1. For relation type data, here we used the advanced function of replicating in Oracle to synchronize data that transfer form Oracle database of Node-B to Oracle of Center.
   First we set up Node-B as a Master to build a CREATE SNAPSHOT LOG and add replica support function, and then register the forms needed to move simultaneously. Second we configure SNAPSHOT in Center and add a snapshot group, put SNAPSHOT in a snapshot group and then set up snapshot REGISTER, run following command in Node-B to implement the data Synchronization after finishing the configures of Center and Node-B.

   ```
   BEGIN
      DBMS_REPCAT.RESUME_MASTER_ACTIVITY(
      gname => '"NODEB_MASTER"');
   END;
   ```

   Among them NODEB_MASTER replica group's name of the theme for what has been set up on Node-B. Dispose through the above, we check in Center end, find that has already got the data of Node-B end, so the function can be realized through this kind of method.

2. For the non-relation type data, the method that we can use to synchronization data is either the data replica or other ways, in our experiment we use rsync +ssh to realize the non-relation type data synchronization work ,and implement automatic synchronized increment of timing back up data from Node-A to data center every day.
   First we point filedb as non-relation type data store directory locating in position /var/filedb of Center, and set subdirectory /var/filedb/nodea in Center. like above process to establish a directory /var/filedb/nodea in Node-A second configure service of ssh and rsync in Node-A to successfully connect with Center. Third we set up a shell script file on Center, and operate automatically timing while putting the file into crontab. This script file is as follows:

   ```
   #!/bin/sh
   BDIR=/var/filedb/nodea
   EXCLUDES=/var/excludes
   BSERVER=nodead
   export RSYNC_PASSWORD=XXXXXX
   BACKUPDIR=‘date +%A‘
   OPTS="--force --ignore-errors
   --delete-excluded
   --exclude-from=$EXCLUDES
   --delete --backup
   --backup-dir=/$BACKUPDIR
   -a"
   export PATH=$PATH:/bin:/usr/bin
   :/usr/local/bin
   [ -d
   /var/emptydir ] || mkdir /var/emptydir
   rsync --delete -a
   /var/emptydir/ $BSERVER::$USER/$BACKUPDIR/
   rmdir /var/emptydir
   rsync
   $OPTS $BDIR $BSERVER::$USER/current
   ```

5 Conclusion

We are now useing ideas and key technologies of the resource and environmental Data Platform in Grid environment that overall discussed In this paper to build the platform with functions of data collection, data management and data share for field data observation stations that distributed in different geographical location. The primeval Data Platform built tentatively proves that the whole structure of the Data Platform and design of every main function module are rational, and some mainly realizing technology obtained is feasible. Because of adopting Grid technology, Java technology and Linux system, the whole system structure is clear, and reuse ratio of program code is high. On the existing experiment platform, the following work is that we will further improving the key techniques about data replicate, data copy, metadata management and application of data, and perfect system of data collection, managing and application integrated in platform for field observation stations. Though the paper focus on the data integrated services for field observation stations, there has certain reference value to the problem of data management of other situations.

6 Acknowledgment

This work is supported by the Incubation Foundation for Special Disciplines of NSFC (National Science Foundation of China) project (grant number: J0130085), the Knowledge Innovation Program of the Cold and Arid Regions Environmental and Engineering Research Institute, Chinese Academy of Sciences (grant number: CACX2003102)and the Knowledge Innovation important Program of Chinese Academy of Sciences(grant number: NF105-SDB-1-21)

If you need to quote a book reference [**?**] you can do it like this. A paper by Author(s) [1] in a conference proceedings or journal can be quoted in a similar manner.

References

[1] Zhang Yaonan,Cheng Guodong,Wei Wuzhou,Tan Zhenhua, 2004.26(2), *The Data Platform of Resources and Environments in Cold and Arid Regions Based on Dawning 3000 High-performance Computing Environments*, JOURNAL OF GLACIOLOGY AND GEOCRYOLOGY, p224-229.

[2] PENG Cheng, LI Jing ,LIU Chun-bo, LIAO Tong-kui, LEI Xiao-feng, 2006.2, *The Study and Design of Geo-spatial Metadata Managing System Based On USDI*, REMOTE SENSING INFORMATION, p53-58.

[3] Redhat AS 4: http://www.redhat.com/docs/manuals/enterprise/

[4] Globus Toolkit: http://www.globus.org/toolkit/

[5] Oracle 10g: http://www.oracle.com/

[6] GDMP: http://project-gdmp.web.cern.ch/project-gdmp/

[7] rsync: http://samba.anu.edu.au/rsync/

[8] OpenSSH:http://www.openssh.com/

Linux in Polyester Automation – A Case Study of Reliance Industries Ltd, Polyester Manufacturing

Rakesh Kumar Gupta and Jitendra Kumar Sasmal
Reliance Industries Ltd.
Village Mora, Post:Bhatha, Hazira, Surat, Gujarat, INDIA
rakesh.gupta@ril.com
jitendra.sasmal@ril.com

Abstract

Reliance has expanded Polyester manufacturing facilities by 550000 Tons per annum. These have a large scale automation for manufacturing control and automation, manufacturing information, product handling, product quality controls, product storage, retrieval for despatch and supply chain logistics. Deviation has been made from Unix to Linux all over the systems and the experience has been most satisfied. This is the largest and firstever complex application of Linux. The paper presents the application part of Linux in Polyester manufacturing automation. Information Technology is extensively used to achieve advantages in cost, productivity and customer satisfaction in every sector of economy. This article is all about strategic use of IT in achieving these objectives in one of the world's largest single location POY manufacturing set up. Linux OS is the key element in this mission .The success of this example should strengthen the decision to use Linux in mission critical operations by others.

1 Introduction

Reliance Industries Ltd. is one among the top ten producers of petrochemical products in the world. Its activities span exploration and production (E&P) of oil & gas, refining, petrochemicals, synthetic fibers, textiles, infrastructure development and life science initiatives. RIL contributes 2.6% towards India's GDP, 7.7% towards total exports.

In the global market RIL ranked the largest producer of Polyester fiber and yarn. This case study is based upon the IT system implemented in the recent expansion of polyester manufacturing facility at Hazira, Gujarat State, India.

2 Background

The manufacturing stages of POY is shown in Figure-1

It includes the following steps.

1. Continuous polymerization (CP)
2. Spinning (SP)
3. Bobbin transport
4. Quality Gradation
5. Bobbin Inventory Management
6. Packing /labeling

Step 1 and Step 2 are governed by the technology supplied by the key suppliers in Polyester industries (Dupont, Barmag)

It is supplied as standard product with slight customization.

Therefore it does not provide scope for the buyer (RIL) to define the IT platform.

At best the supplier of technology agrees to implement the guideline for data exchange defined by the prospective customer at the start of the project.

But Steps 3 thru 6 provides for selecting customized solution.

These steps are characterized by

- Intensive manual work in labeling, physical testing, sorting, packaging of bobbins day in day out
- Needs more floor space

- Potential source for product down gradation and product mix while packing which results in customer complaint

The key to sustenance of the thin margin in sales price per unit (Kg) of POY depends upon

- Reduction in cost of handling
- Reduction in cost of packaging

Over the years the industry has evolved to implement various **automated handling and packaging solutions to achieve the following key objectives.**

- Reduce the chances of quality down gradation of bobbin during handling
- Reduce the cycle time in quality sampling/testing and QC release to achieve minimum time from spinning to warehousing.
- Provide integrated MIS to quickly trace the life cycle of Bobbins from customer end to spinning and CP.

Any automated solution provides a great opportunity for achieving optimum results in term of quality and cost with help of IT

The present case provided us one such opportunity.

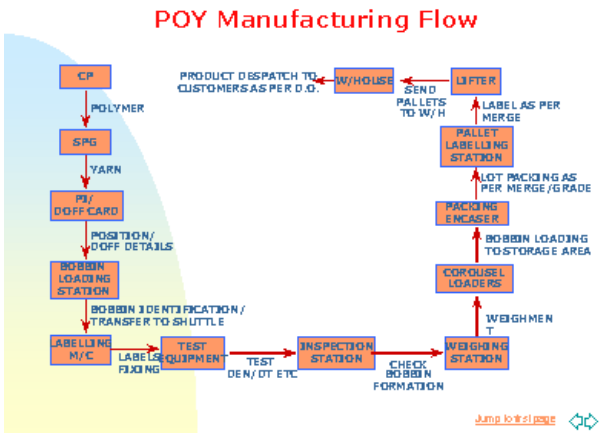


FIGURE 1: *POY Manufacturing Flow*

3 Scenario:

3.1 Year 1995:

The automated solutions in old plants during these period relied upon servers operating on with Unix OS and C language.

The total cost of ownership of these systems based upon

- High cost of investment in RISC processor based severs
- License fees for Unix and other development tools
- License fees for SW products for message exchange with PLCs and work stations

3.1.1 Operating cost consisting of

- Annual Comprehensive maintenance cost for HW and software
- Cost of obsolescence (end of life cycle of products)
- High replacement cost of servers.

On the whole we were stuck with solution leading to escalation of cost with passing of time.

There is tremendous pressure to bring change in this process.

At the turn of century Open Source systems were talked about more.

And prominent vendors like IBM and HP started offering technical support.

3.2 Year 2004:

The following decision was made for future projects.

- Linux OS platform on Intel processor based servers
- Oracle RDBMS
- Java developer as front-end developer

With a view to achieve the following objectives.

- Reliable and cost effective IT infrastructure
- Minimize the cost of obsolesce
- Stop the escalation in TCO with passage of time
- Scalable systems
- Open connectivity

These technologies are used to develop customized solution for the following applications

- Automatic Bobbin handling system
- POY manufacturing system
- Automated ware housing system

3.3 Why Linux?

- Linux provided the stability of in real time operation similar to Unix (Windows never attained stability in this category)
- It can run on any HW platform (we used Intel processor to achieve maximum cost benefit)
- Open source enabled to implement customized software solution for integrating different automation HW (Special Printers, Scanners, RFID readers, PLCs etc) which are important components in automation system
- Strong Networking capability
- Not proprietary

4 Brief description of Automated Bobbin Handling system (Ref Figure-2):

- POY bobbins are automatically transferred from Spinning position on to Doffers running on mono rail
- Bobbins from the doffers are transferred to Shuttles having pegs running on bus bar powered rail system
- Each bobbin is automatically applied with identification tube labels and weighed in auto weighing system while transported on the shuttles
- Shuttles carrying bobbins routed thru Physical testing lab for capturing physical properties (Denier, draw tension, Waviness, cross section etc) by means of tab testing instruments connected to LAN
- Bobbins on shuttles are transferred to Storage (Carousal system) for intermediate storage before packing
- Bobbins on carousal are automatically given quality gradation stamp and released for packing in the database system
- Bobbins transferred from carousal with the help of transfer devices on to shuttles and to packing line where different robot packs the bobbins to meet requirement of packing manager.
- These packages are labeled and transferred to multi product automated warehouse inside the plant.
- From the automated ware house the packages are automatically taken out to load in containers to meet the delivery order requirements given by logistics

5 Control systems:

The plant is managed with the help of IT systems configured in three-tier hierarchy.

5.1 Level-1:

Consists of PLCs to control movement of different equipment and devices distributed all the over the area of operation.

5.2 Level-2:

Consist of database servers and workstations for interfacing human operation. This layer provides intelligence to different devices for the required operation and in real time transaction mode to control all movements. In short works more than supervisory system. The work stations are thin clients which when powered on receives the configured files from the server.

5.3 Level-3:

Consist of Common servers, which provide the integrated view of production statistics and commercial transaction.

All these devices are connected on Ethernet network using TCP/IP communication protocol.

Typical network diagram is given in the Figure-3.

6 Key Features of Linux used for our purpose:

6.1 Communications with PLCs:

- Socket program used for direct read and write in the PLC registers thru TCP/IP.
- Most vendors of PLC do provide open connectivity using TCP/IP thru 100 mbps Ethernet NIC
- Use high speed ether net connection for transmission of signal and data thru different physical path

- Panel views in field could be configured to provide real time status of the device and logical data directly fetched from database. This enables trouble shooting of alarms in jiffy. There are number of handy panel view stations positioned all thru the plant for this purpose

6.2 Thin Client: (DHCP services)

Approximately 100 workstations are disk less PCs, which receive the required application component and drivers from central server to execute required function at the location. With this the management of special software drivers and EPROM cards which otherwise were required to be fixed on individual PCs are eliminated. This has achieved simple to manage infrastructure that demand quick resolution.

6.3 Other services of Linux:

FTP, SSH, Web services, Time Protocol and Linux Cluster are used

6.4 Details of software:

- Linux Red Hat Release 3.0
- Services like DHCP, FTP, SSH, VNC, Webservices, Time Protcool
- Oracle 10 g with RAC

6.5 Other issues:

- Oracle cluster provide fault tolerance functionality to provide data access from any of master / slave servers
- Data communication between different systems achieved thru dblink that is the fastest we can achieve in any interface of this nature
- Java client provide the GUI for user friendly operation environment

6.6 Communication Network

- The devices in Level-1, Level –2 and Level – 3 are connected thru fiber optic network with the help manageable network switches providing functionalities like VLAN, VPN connectivity and network security.
- It provides connectivity between any devices form anywhere thru appropriate authorization defined in the network switches.

7 Major data communications are as follows:

- Spinning control system ♢ Automation level-2 server (Bobbin detail)
- Automation server ♢ Doffer Control PLC (destination for spinning position)

There are many to and fro communication between PLC to PLC and Server to PLC

- Level –2 Server ♢ Automated Warehouse Server (Package details)
- SAP Server ♢ Automated Warehouse Server (Customer Order Information)
- Automated Warehouse Server ♢ SAP Server (Packing List information)

References

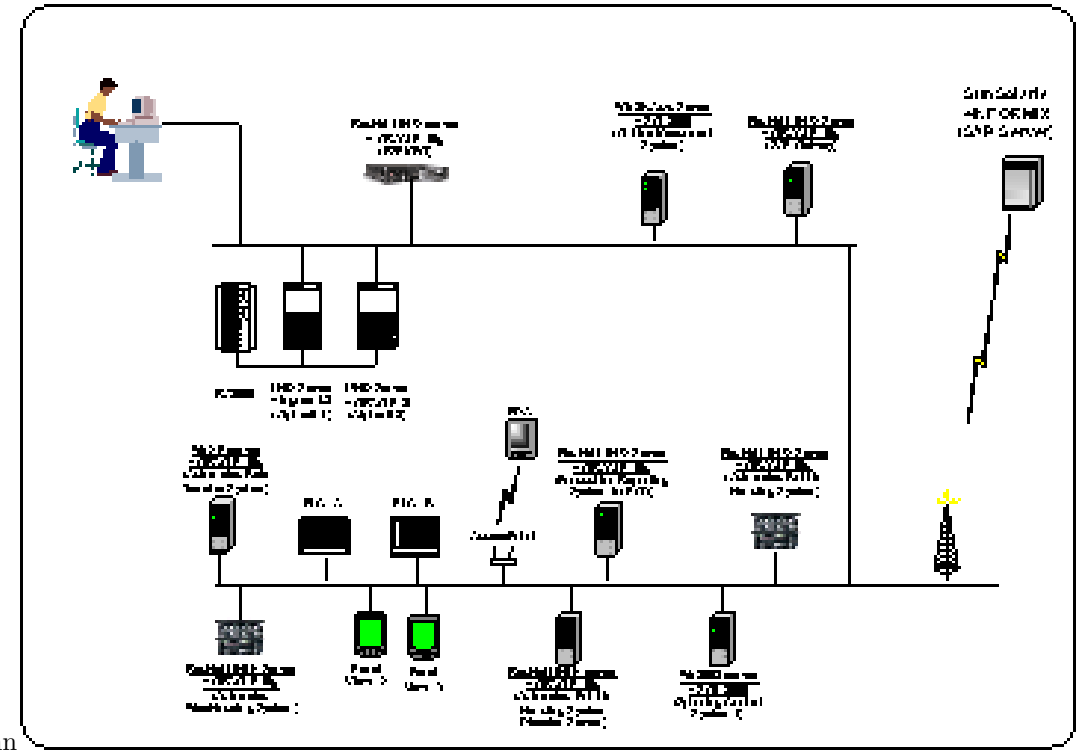
[1] http://www.salmoiraghi-spa-monza.com/

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

[3] http://www.ameinfo.com/75175.html

[4] http://www.michaelhorowitz.com/Linux.vs.Windows.html

[5] http://www.devx.com/opensource/Article/16969



nn


14 5 2001


100 100

GNU/Linux Based TDOA Localization Using COTS Hardware

Georg Schiesser
OpenTech EDV-Research GmbH [1]
Lichtenstein Str 31, 2130 Mistelbach, AUSTRIA
e0307201@student.tuwien.ac.at

Abstract

The goal of this project is to determine the position of a sound source using an array of four microphones, a good quality audio card and a GNU/Linux based PC. The microphones are located at exactly known positions somewhere in the room. Time-Difference-Of-Arrival (TDOA) localization is based on the fact that the sound signal is received at the microphones with relative time-differences, depending on the distance between microphone and sound source and the speed of sound in air (c $\approx$ 343 m/s). At present either a frequency generator or a test-signal are used as the sound source. Later it will be possible to use other sound sources like clapping hands but recent measurements showed that this is difficult because of nonlinear recording effects like echo and distortion. Up to now the audio signals are recorded using ALSA. In general every tool can be used to record audio. One of the project's goals is to use the real-time audio toolkit RAT in a later version to guarantee minimal latencies in the recording interface. The calculation of the TDOA values between the four audio streams is based on an iterative cross-correlation algorithm optimized for low latencies. Every channel is compared with every other channel at different time lags ranging from -WINDOW to +WINDOW. The maximum value indicates the time difference at which two channels are correlated. Traditional algorithms are based on FFT or other transformations, but these techniques cannot be used if the TDOA values must be updated after every sample to keep the latencies at the minimum. In general TDOA localization can be thought of calculating the intersection of multiple hyperboloids. The equation system is too complex to be solved exactly. So the position of the sound source is approximated using Newton's method of multidimensional-root-finding. The algorithm is based on the GNU Scientific Library (GSL).

1 Introduction

The aim of this project is to implement a TDOA localization in GNU/Linux [2] [3] to determine the position of a sound source with multiple microphones and a good soundcard.

I am not sure but i think that one motivation for this project was to get rid of the flies in my room in summer. I thought that it would be nice if there was a servo-controlled 10MW laser-beam which could burn all the flying insects. I quickly realized that using image-recognition to localize the position of a $1cm^3$ object is not possible. So i decided to try TDOA localization of the moving wings.

There are a lot of papers about TDOA localization available, but most of the work is proprietary and not available to the public, especially in the field of Passive Surveillance [4] [5] [6] [7].

So the main motivation for this project is to implement the basic functionalities using mainstream hardware and a GNU/Linux system provinding results under GPL and FDL [8] [9].

TDOA localization can be used in various fields like robotics, automation, and multimedia.

But keep in mind that all localization stuff done is based on the speed of sound in air. If you want to determine the position of your favorite radio station, then you will need other hardware than a soundcard.

2 Recording Interface

Although recording is a very important part of the project, I will only describe the most important things because there is a lot of documentation available about how to get a sound signal into your PC.

Up to now recording has been done using only one multi-channel soundcard. Later there will be the possibility to record the audio signals with distributed nodes and timestamp synchronisation. Dis-

tributed data aquisition is much more difficult, especially if no external trigger is available. Recent investigations showed that there are hard-real-time linux drivers available, for example R.E.D.D. [10] [11] and RTnet [12]. They could be used to implement a global-time protocol to synchronize distributed data aquisition nodes.

2.1 Hardware

The basic hardware setup consists of a standard PC speaker, a microphone-array and a good quality sound card.

FIGURE 1: *Hardware Setup*

The speaker is used to play a test-signal to make the localization easier. Four microphones are placed at exactly known positions somewhere in the room. A Terratec Phase88 PCI sound card is used to record the signals.

Keep in mind that some kinds of microphones need phantom power and must be pre-amplified before you can connect them to the soundcard.

2.2 Software

Up to now recording has been done using the well-known ALSA tools [13]. After configuring the soundcard using `alsamixer` you can start recording.

```
# arecord --file-type=raw   \
          --channels=4      \
          --format=S32_LE   \
          --rate=96000      \
          --duration=1 > data.raw
```

Note that `envy24control` [14] can be used to configure the non-standard functionalities of Terratec soundcards based on the **envy** chipset. Furthermore it can be very usefull to install the linux sound exchange tool `sox` [15] if you want to apply filters or change the data format.

At present all data is written into a file before starting signal processing because it is easier to verify the measurements and the TDOA localization has a very high cpu-usage.

2.3 Test-Signal

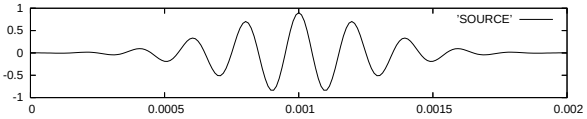


FIGURE 2: *Test-Signal*

During development a test-signal was used to test the cross-correlation algorithm. The test-signal should be a short pulse with a constant period which must be high enough to produce unique positions within a specified area. Rectangular pulses are not appropiate because they consist of too much frequencies, especially high frequencies are a big problem. Pure sine waves don't work because they would yield multiple possible positions during cross-correlation. The test-signal was generated using a sine wave multiplied with a gaussian curve as shown in figure 2.

3 Cross-Correlation

This section is about how to determine the time-differences-of-arrival (TDOAs) by calculating the cross-correlation of the four audio streams.

3.1 Basics

In signal processing, the cross-correlation is a measure of similarity of two signals, commonly used to find features in an unknown signal by comparing it to a known one. It is a function of the relative time between the signals. For discrete functions f_i and g_i the cross-correlation is defined as

$$(f \star g)_i \equiv \sum_j f_j \, g_{i+j} \qquad (1)$$

where the sum is over the appropriate values of the integer j. [16]

3.2 Optimization

We want to keep the latencies at a minimum, so we must update the cross-correlation after every sample. Furthermore we assume that the TDOAs must be in the range of $-WINDOW$ to $+WINDOW$ samples. Using the standard algorithm we would have to calculate about

$$SAMPLERATE * (2 * WINDOW)^2 \qquad (2)$$

multiplications per second. The most important part of the implementation is that we only update the cross-correlation instead of calculating it.

```
c[N] += MULTIPLY(a_new, b_new);
c[N] -= MULTIPLY(a_old, b_old);

for (i = 1; i < N; i++)
{
  c[N + i] += MULTIPLY(b_new, a[i]);
  c[N + i] -= MULTIPLY(b_old, a[N + i]);

  c[N - i] += MULTIPLY(a_new, b[i]);
  c[N - i] -= MULTIPLY(a_old, b[N + i]);
}
```

Using the modified algorithm we can reduce the number of multiplications per second to

$$SAMPLERATE * 2 * WINDOW \tag{3}$$

The maximum's index of the cross-correlation now specifies the TDOA at which the two signals match. Furthermore we can define the fitness of the solution by the maximum itself.

3.3 Redundancy

The cross-correlation must be calculated between every pair of audio streams. Using four microphones we have to calculate a total number of 6 time-differences between AB, AC, AD, BC, BD and CD. In general we are only interrested in the TDOAs AB, AC and AD, but recent measurements showed that calculating the TDOAs is the most critical part and should be done in a redundant way to minimize the error.

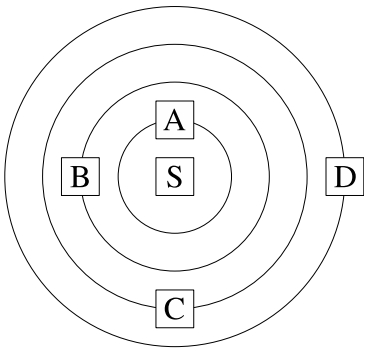


FIGURE 3: *Redundancy*

$$ABC := AB + BC - AC = 0 \tag{4}$$
$$ABD := AB + BD - AD = 0 \tag{5}$$
$$ACD := AC + CD - AD = 0 \tag{6}$$

As this application shows unfavorable error propagation some form of error detection is needed, this equation system can be used to determine the errornous part of the calculated TDOAs or in case of "out of bounds errors" drop the result entirely.

3.4 Runtime Optimization

The calculation of the time-differences between each channel is very cpu-intensive. After each sample the cross-correlation must be updated to determine the time-difference of arrival. There are several ways to optimize for runtime.

- Samplerate: Reducing the samplerate, for example from $96kHz$ to $48kHz$, would of course speed up the algorithm dramatically. But keep in mind that only using half of the time-resolution would also reduce the accuracy of the cross-correlation.

- Samplesize: The soundcard is able to record at $96kHz$ and $24bit$ sample resolution. During the cross-correlation the input samples are multiplied with other input samples. This is done $WINDOW$ times. Assuming that we have a $WINDOW$ of 512 samples, we would at least need $58bit$. So either the samplesize must be reduced to do $32bit$ calculations or you have do it the $64bit$ way. During the development of this project both the $32bit$ and the $64bit$ implementation have been tested and the measurements showed that only using $11bit$ input samplesize to do $32bit$ calculations is enough. There are only minor differences in the results of the two versions.

- Windowsize: Most of the measurements have been done using a $WINDOW$ of 512 samples. The $WINDOW$ depends on the samplerate, the microphone locations and the speed of sound.

One of the most recent feature is the input trigger. It is used to avoid calculating the cross-correlations if all input channels are silent. During this period no calculation is done. The cross-correlation algorithm only starts when the input volume of one one channel increases above the trigger-level.

So the cpu-usage can be reduced most of the time when all input channels are low. Furthermore this optimization does not have any impact on the accuracy of the cross-correlation.

3.5 Example

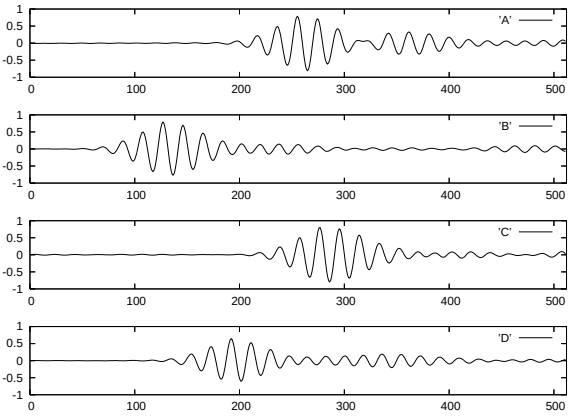


FIGURE 4: *TDOA in Samples*

This measurement was done using the test-signal and four microphones. In figure 4 you can see the channes A, B, C, and D recorded by the soundcard. The signal from the first channel A is disturbed and contains echo from a wall, which should be avoided because the echo signal can cause wrong cross-correlation results.

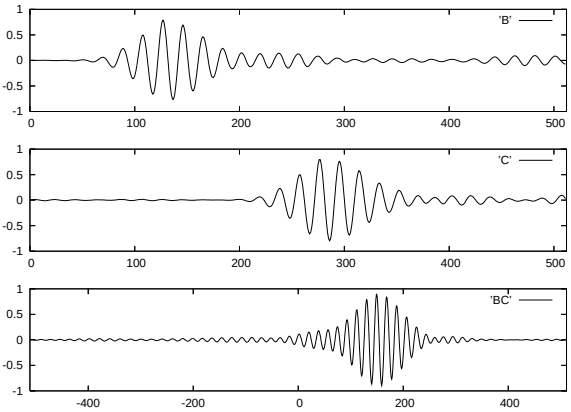


FIGURE 5: *Channel B and C and the Cross-Correlation BC in Samples*

In this example the cross-correlation is calculated between channel B and C. The time-difference-of-arrival between the channels is about 150 samples. Figure 5 shows the two channels and the cross-correlation with a maximum value at the index 150 which indicates the TDOA between the channels.

4 Position approximation

4.1 Multilateration

Multilateration, also known as hyperbolic positioning, is the process of locating an object by accurately computing the time difference of arrival (TDOA) of a signal emitted from the object to three or more receivers. It also refers to the case of locating a receiver by measuring the TDOA of a signal transmitted from three or more synchronised transmitters. [17]

4.2 Principle

If a pulse is emitted from a platform, it will arrive at slightly different times at two spatially separated receiver sites, the TDOA being due to the different distances of each receiver from the platform. In fact, for given locations of the two receivers, a whole series of emitter locations would give the same measurement of TDOA. Given two receiver locations and a known TDOA, the locus of possible emitter locations is a hyperboloid (a surface approximately shaped like two cones joined at the origin). In simple terms, with two receivers at known locations, an emitter can be located onto a hyperboloid. Note that the receivers do not need to know the absolute time at which the pulse was transmitted - only the time difference is needed.

Consider now a third receiver at a third location. This would provide a second TDOA measurement and hence locate the emitter on a second hyperboloid. The intersection of these two hyperboloids describes a curve on which the emitter lies.

If a fourth receiver is now introduced, a third TDOA measurement is available and the intersection of the resulting third hyperboloid with the curve already found with the other three receivers defines a unique point in space. The emitter's location is therefore fully determined in 3D. [17]

4.3 Problem Statement

For two 3D points, $P = (p_x, p_y, p_z)$ and $Q = (q_x, q_y, q_z)$, the distance $\|P - Q\|$ is computed as

$$\sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2} \qquad (7)$$

The relationship between the microphone positions and the position of the sound source is defined as

$$\|P - P_i\| - \|P - P_0\| - c * TDOA_i = 0 \qquad (8)$$

where P is the unknown position of sound source and P_i is the position of the i-th microphone. $TDOA_i$ denotes the time difference of arrival between the microphones P_i and P_0.

Using four microphones the position of the sound source P can be computed as

$$\|P - P_1\| - \|P - P_0\| - c * TDOA_1 = 0 \quad (9)$$
$$\|P - P_2\| - \|P - P_0\| - c * TDOA_2 = 0 \quad (10)$$
$$\|P - P_3\| - \|P - P_0\| - c * TDOA_3 = 0 \quad (11)$$

The equation system is too complex to be solved exactly. Instead, an approximation algorithm is used to determine the position of the sound source. In this project I decided to use the GNU Scientific Library [18] to implement the position approximation.

4.4 Root-Finding

The most common example of root-finding is the newton iteration defined as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (12)$$

Multidimensional Root-Finding is similar to onedimensional root-finding [19]. It requires the simultaneous solution of n equations, f_i, in n variables, x_i, $f_i(x_1, ..., x_n) = 0$ for $i = 1...n$.

In general there is no way of knowing whether any solutions exist. All algorithms proceed from an initial guess using a variant of the Newton iteration,

$$J\Delta x = -f(x) \quad (13)$$

where x, f are vector quantities and J is the Jacobian matrix $J_{ij} = \partial f_i / \partial x_j$. [18]

4.5 Implementation

At first an initial starting position is choosen. The initial guess is based on TDOA values. Then one newton iteration is executed to calculate a new approximation of the source position. The distance between the current and the previous position yields the accuracy of the current approximation. This is done until the accuracy is below a given tolerance or the maximum number of iteration has exceeded.

5 Conclusion

Although this first prototype shows that TDOA localization with standard Linux is in priciple possible, it also showed a number of limitations both in the algorithmic as well as at the system level. The main issues emerging are:

- precision of the sampling system
- error propagation and resulting numerical effects
- computational resource demands

Due to these problems we decided to continue this project based on a distributed real-time Linux system, outlined below.

Further it can be concluded that even if the final system has hard real-time demands and extensive numerical work to do, mandating distributed resources, prototyping of real-time systems with COTS based systems using mainstream Linux is reasonably possible even in systems with very sensitive signals. Both development and initial verification of the algorithms were significantly simplified by building on mainstream Linux due to debugging capabilities and lack of the RT-Linux inherent constraints (i.e. library access).

Aside from the technical problems a design limitation emerged, in a TDOA system an automated initial calibration method is almost mandatory, measuring the distances of emitter and receiver is not only a painstacking procedure but it is very hard to precisely measure the distances as the point of emission is generally not easily accessible (i.e. what point to you take on a speaker? what point to you take on the microphone?).

6 Future Work

Usage of standard audio devices introduces some uncertainty with respect to the processing of audio interrupts. In mainstream Linux interrupts can be off for a significant time limiting the accuracy of positional calculation. For this reason future works are planed to utilize the hard real-time variants of Linux - in our case RTLinux/GPL [20]. To make the problem more interesting and to improve the scalability (currently a high end number cruncher is a requirement - which does not always fit the demands for hard real-time too well)

6.1 RTLinux recording interface

RTLinux/GPL is a POSIX PSE51 driven RTOS on top of Linux, it is not actually a typcial audio platform, but provides a basic deterministic execution context on which a TDOA system can be implemented with increased precision of sampling. Unfortunately there are some limitations of this kernel-space real-time Linux implementation, notably the inability of directly accessing user-space libraries like the GNU Scientific library, thus a change of design is needed providing a data interface from kernel space, in which the RT-threads execute, to user-space in which the non time-critical calculations are executed. RTLinux/GPL provides fifos and shared memory as basic means of interfacing to user-space, on top of

this a sound driver would need to be implemented to access the sound cards as the standard Linux drivers are not RT-safe. Instead we decided to utilize a previous project, the Real-Time Audio Toolkit RAT.

6.2 Real-Time Audio Toolkit RAT

The Real-Time Audio Tools [21] implemented by Arvid Staub provides a low level driver for high-end audio cards (TERRATEC Phase88) and an elaborate interface for channel management and data routing. The basic concept is to abstract each channel of a multichannel card to a logical channel and then describe the data route that the respective data items are to take.

Though originally designed for audio work and not as general data sampling interface RAT has the advantage that it was designed to deliver data to user-space from the very beginnning, basically for hard-disk recording and playback - thus this fits well with the modfified design that executes the numerically intensive parts of the TDOA problem in user-space while guaranteeing deterministic sampling.

6.3 Global Time Synchronisation

To implement a distributed system there are two main demands:

- Global time
- Real-time communication

The two are to a certain extent coupled. For the real-time communication the Real Time Ethernet Drivers [11] in the latest variant [22] are to be used. The decision of how to implement global time, TDM based or via time-diffusion, has not been found yet as detailed evaluation is still pending. So the next steps for the global time will require some prototyping and benchmarking to find a technology that can work efficiently with COTS hardware components under RTLinux/GPL.

6.4 Distributed Data Aquisition

Obviously a distributed system for TDOA will require a system level knowledge of the data, further more if the responsiveness of the system should be sufficiently (i.e. to trace moving objects) the distribution of data will be a crucial point, as noted above the Real-Time Ethernet Device Drivers are to be used for the communication over dedicated ethernet links (point-to-point for now). This will introduce a substantial latency in the overall data aquisition compared to single node data aquisition thus compensation based on precise global time is mandatory.

6.5 Generalized Cross-Correlation

In signal processing it is common to use the generalized cross-correlation to determine the time-difference between noisy signals. This could be another way to optimize the accuracy of the cross-correlation.

6.6 TDOA verification using Neural Networks

Up to now simple condition statements have been used to throw away bad TDOA values calculated by the cross-correlation. Instead, the calculated TDOA values could be verified using neural networks.

6.7 Multidimensional Minimization

During development we had some problems concerning the convergence of the approximation algorithm. This problems could be due to the fact that the error propagation in the overall system is very unfavorable, especially if one assumes general signals and not the test-signals only, it showed that increasing the bit-width of the samples would more or less have no effect due to the integrative nature of the cross-correlation, though an increase of sampling frequency seems promising furthermore an uncertainty with respect to the effective jitter of the signal needs clarification. To clarify or rather confirm that the bit width of the samples is not relevant we ran measurements with reduced bit width (down to 11bit samples) which showed no significant change in the outcome. The second possibility, to increase the sampling frequency, seems limited with COTS components (though a factor of two or four seems realistic, but hardly much more). To improve the jitter of the sampling system a switch from a GPOS (mainstream Linux) to a RTOS (RTLinux/GPL) is proposed.

All these changes together never the less mandate searching for better methodologies than the newton approximation for the convergence estimation. The newton approximation has the severe limitation that a badly selected starting point will in the worst case not converge at all, but in any case the execution time of this method is not known. Possible alternatives under concideration include Multidimensional Minimization based on the GNU Scientific Library.

References

[1] OpenTech, *OpenTech - RTLinux Support in Europe*, 2006-09-14, http://www.opentech.at/

[2] GNU, *GNU's Not Unix! - Free Software, Free Society*, 2006-09-15, http://www.gnu.org/

[3] Linux, *The Linux Kernel Archives*, 2006-09-15, http://www.kernel.org/

[4] Briukhov, Kalinichenko, Zakharov, Panchuk, Vitkovsky, Zhelenkova, Dluzhnevskaya, Malkov, Kovaleva, 2005, *Information Infrastructure of the Russian Virtual Observatiry (RVO)*, Russian Academy of Sciences.

[5] Roke, *Vigilance - wide area multilateration system*, 2006-09-15, http://www.roke.co.uk/vigilance/

[6] Baniak, Baker, Cunningham, Martin, 1999, *Silent Sentry - Passive Surveillance*, Lockheed Martin Mission Systems.

[7] ERA Radar Technology, *Multilateration Surveillance Radar P3D*, 2006-09-15, http://www.era.cz/en/msr-3d.shtml

[8] GNU, *What is Copyleft?*, 2006-09-15, http://www.gnu.org/copyleft/

[9] FSF, *Free Software Licensing*, 2006-09-15, http://www.fsf.org/licensing

[10] Sourceforge, *REDD: RTLinux Ethernet Device Drivers*, 2006-09-14, http://redd.sourceforge.net/

[11] Sergio Perez, Joan Vila and Ismael Ripoll, 2003, *Building Ethernet Drivers on RTLinux-GPL*, Proceedings of the 5th Real-Time Linux Workshop, pp279–286.

[12] Jan Kiszka, Bernardo Wagner, Yuchen Zhang, Jan Broenink, 2005, *RTnet - A Flexible Hard Real-Time Networking Framework*, 10th IEEE International Conference on Emerging Technologies and Factory Automation.

[13] ALSA, *Advanced Linux Sound Architecture*, 2006-09-15, http://www.alsa-project.org/

[14] ALSA OpenSRC, *Envy24Control*, 2006-09-15, http://alsa.opensrc.org/?page=Envy24Control

[15] Sourceforge, *SoX - Sound eXchange*, 2006-09-15, http://sox.sourceforge.net/

[16] Wikipedia, *Cross-correlation*, 2006-09-01, http://en.wikipedia.org/wiki/Cross-correlation

[17] Wikipedia, *Multilateration*, 2006-09-17, http://en.wikipedia.org/wiki/Multilateration

[18] Gnu, *GSL - GNU Scientific Library*, 2006-09-15, http://www.gnu.org/software/gsl/

[19] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, 1992, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, ISBN 0-521-43108-5.

[20] RTLinux/GPL , *Real-Time Linux*, 2006-09-27, http://www.rtlinux-gpl.org/

[21] Arvid Staub, 2004, *RAT - Real-Time Audio Tools*, Proceedings of the 6th Real-Time Linux Workshop, pp57–63.

[22] Florian Bruckner, 2006, *Realtime Ethernet Device Drivers*, Proceedings of the 8th Real-Time Linux Workshop, tba.

A Linux Based System to Monitor Train Speed and Doors for the Light Rail System in Guadalajara, Mexico

D. W. Carr, R. Ruelas, H. Salcedo-Becerra, and G. A. Ponce-Castañeda
Departamento de Ingenieríade Proyectos, Universidad de Guadalajara
Apdo. Postal 307, C.P. 45101 Zapopan, Jalisco, MEXICO
doncarr@gmail.com, rruelas@newton.dip.udg.mx

Abstract

In developing countries, the cost of proprietary operating systems and tools can add substantially to the cost of control projects. At the same time, Linux has become very robust and extremely easy to port to single board computers and other hardware used for real-time systems. Linux is becoming the OS of choice for many board manufacturers for the reasons just mentioned, and also because there are no up-front costs or royalties to pay. There are also a wealth of tools that customers can use for no cost, reducing further the cost to the end user. For this reason, we have chosen Linux running on an arm based single board computer for pilot tests of a control system to be installed on each train for the light rail system in Guadalajara, Mexico. This control system monitors the speed of each train to prevent dangerous over-speed conditions, and also to prevent the driver from opening doors on the side of the train where there is no passenger platform. This system will also be CBTC (Communications Based Train Control) ready for use when there are funds to purchase the interfaces with the digital radio system to allow constant communications with each train.

1 Introduction

At the light rail system in Guadalajara (SITEUR), there is a desire to reduce human factors and increase safety of the system. Currently, the drivers are responsible for maintaining the correct velocity of the train for all sections of track, and also to open the doors on the correct side of the train at each passenger station. A failure in either of these areas could result in injury or loss of life. At the same time, there are limited funds available to correct these and other safety issues, and a desire to create safe systems even if the millions of dollars required for a proprietary system are not available. For example, the type of funding available in New York City [1] for the CBTC system they are installing is completely out of reach in Mexico. The references [2] and [3] include some requirements for a CBTC system.

The required functionality for the first part of the project, which is described here, is to constantly monitor the speed of the trains and compare this against a predetermined set of profiles, and to make sure the driver never opens the doors on the side without a passenger platform. In addition to monitoring the speed of the trains, four speed profiles were proposed such that the operator is warned before dangerous conditions are reached, and finally to stop the train if necessary. At the light rail system in Guadalajara (Figure 1 an example line), for most of a round trip, the passenger platforms are to the right of the trains, except at the ends of both trajectories where the platform could be at either side of the trains; it depends on which track the train uses to enter these stations, though the trains normally cross over and discharge passengers on the left.

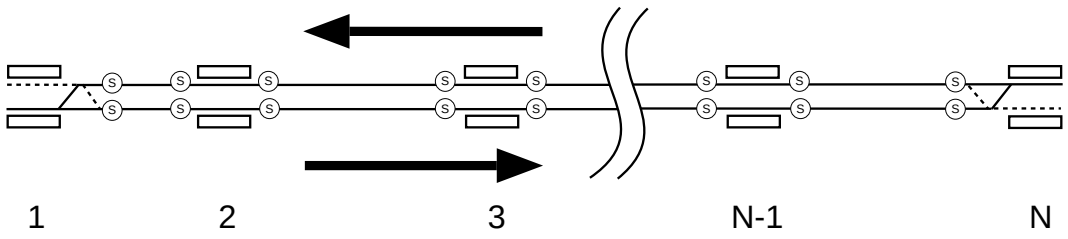


FIGURE 1: *An Example Train Line.*

Even though the initial functionality is for monitoring the speed and doors, this platform will be CBTC ready [4] [5] when there are funds to purchase the components necessary for constant communication with the central system. In this mode, the train will constantly communicate speed and position, as well as receive commands from the central system. The central system will then perform the overall decision function, sending orders to slow down or to stop trains according to the situation just detected. Of course the goal of the on-board system is to have the sufficient autonomy in order to control the train without intervention for every action from the central system.

In the second section we describe the hardware used for the implementation of the monitoring system. In third section a description of the software design is presented which is based on N-version programming. In the fourth section we present some results. The fifth section contains a discussion about standard compliance. And in the sixth and last section we give the most important conclusions about this work.

2 Hardware

The hardware used for the implementation of the monitoring system in one train is basically made up of one arm based single board computer with serial ports, high speed counters, network, discrete I/O, on-board user-programmable FPGA, Ethernet to wireless bridge, and two 4x40 LCD displays for the front and rear cabins. We have started testing with the TS-7300 arm based single board computer, Orbital Matrix 4x40 LCD screen with keypad interface and serial port, and Ethernet-Wireless bridge. The signal conditioning was created in-house at the light rails system. This system is a pilot test for a control system to be installed on each train for the light rail system in Guadalajara, Mexico.

We have also started testing a very small Freescale Coldfire based single board computer. This computer also has two counter inputs and other features needed for this application. Nevertheless, we have decided to use the TS-7300. An important feature of this hardware is that it has two high speed counters on board, to be used for counting the pulses from two encoders attached to the wheels of the train, and a wealth of standard digital I/O.

The TS-7300 (Figure 2) has a 200 MHz Cirrus Arm processor and probably has 100 times the processing power needed for the actual application, but ewill be ready for future needs. The single board computer comes standard with Debian, but can also be loaded with a more compact version of Linux created by the manufacturer (TS-Linux).

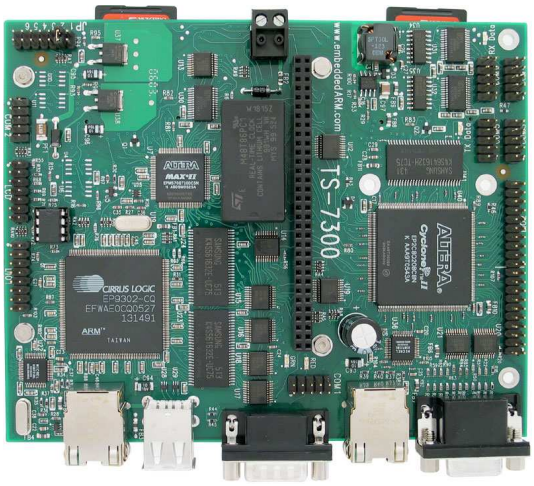


FIGURE 2: *TS-7300 Single Board Computer.*

The TS-7300 board has six standard on-board serial ports. One which is reserved for the console, two for the LCD displays and keypads in the front and rear cabins, one is RS485 ready in the case that we need to add additional I/O, another is reserved for connecting to the digital radio, and the last one is left as a spare. Figure 3 shows a picture of the computer with all of its interfaces.

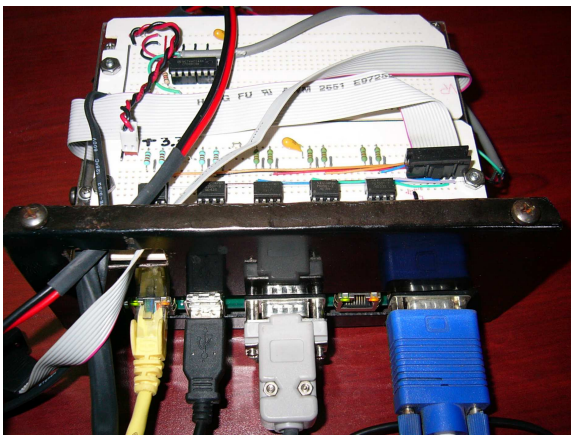


FIGURE 3: *Computer with signal conditioning included.*

The prototype uses the Matrix Orbital LK404-55 display, which is four rows by forty columns with fifty five key keypad interface and serial port. This display also supports vertical and horizontal bar graphs. In Figure 4 there is an example of the kind of graphs used for the application. In this case, two bars indicate the desired speed and the real speed of the train. In the same figure we can see the keyboard installed to allow the users to interact with the computer.

FIGURE 4: *Display and keyboard.*

The speed of the train is measured through two encoders attached to the wheels. The pulses provided by these encoders needed signal conditioners which were developed in-house, in Figure 5re there is a picture that shows these circuits and how they are installed in the same box with the computer.

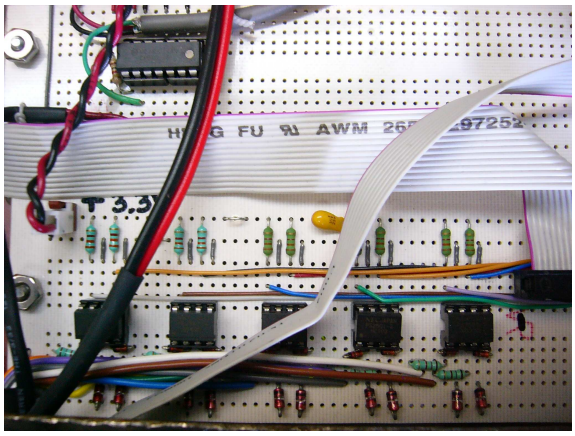


FIGURE 5: *Signal conditioners.*

For the moment, the single board computer has two SD memory cards that allows the available non-volatile memory on the board to be increased or to share data if the wireless network is not available, or, if it is preferred directly reading of the memories. The two memory sticks are connected to the rear of the board, as can be seen in Figure 6. The computer boots from the one on the right and it must always be present when running.

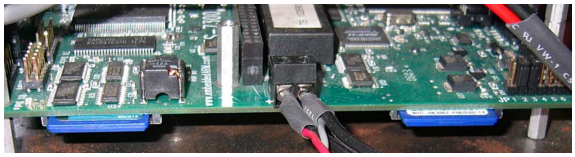


FIGURE 6: *Back of the computer and memory cards.*

All the trains will be connected to a local network for which we are using an Ethernet-wireless bridge, in order to download driver performance and speed profiles after each round trip. Figure 6 shows the communications of the trains with a central system. However, there is a wireless access point only located at the extreme of each line of the light rail system for now. That means that data can be downloaded or new profiles/configurations uploaded to the trains at both ends of the line when considered necessary, and these information must be accessible in local network of the enterprise. This is an important design feature that eliminates the need to visit each train car with a laptop to download data, or update the profiles/configurations.

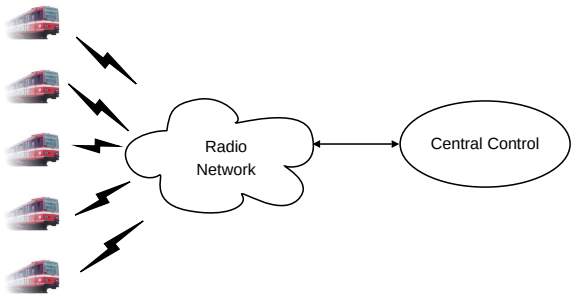


FIGURE 7: *Communications Based Train Control.*

3 Software Design

For the software design we have chosen Linux running on an arm based single board computer, while the software for the application is based on N-Version programming [6][7], that is, multiple versions of the same program, each written by a different group, are developed, implemented, and compared in order to verify correct operation of the trains.

When at least one program, among the N-version programmed, provides an output different to the rest, the safest operation is taken. For example, if one algorithm votes to stop the train, the train is stopped. For this application we employ three of each algorithm for the final product. The priority here is safe operation over quality of service, trying to keep this last one as good as possible.

Of course the software must be extremely reliable since interruptions in service are considered very expensive. This means that each version must be extremely reliable. There is a growing consensus that given the same resources, safer software can be created with N-versions rather than putting all resources into writing one "good" version [8].

The algorithms of monitoring and control of speed and doors use N-Version program. In Figure 8 we show the scheme of a single speed algorithm, whereas Figure 9 contains the scheme of a single door algorithm.

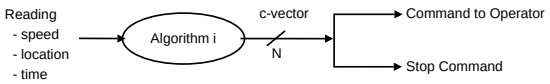


FIGURE 8: *A Single Speed Algorithm.*

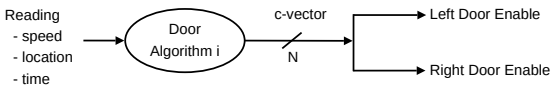


FIGURE 9: *A Single Door Algorithm.*

4 Results

Figure 4 shows the departure from the Urdaneta station. The desired speed, the actual speed, and the advanced distance for the present segment between two stations are displayed on the screen. This allows to the drivers of the train to know the desired speed, the actual speed, and the approximate location at every moment.

Monitoring speed requires speed profiles. In our case, we have four of them corresponding to under speed, desired speed, high speed, and very high speed. For under-speed, the driver will be warned that he is going too slow, but the train will never be stopped. For high speed, the driver will be warned, but will not be allowed to remain above the high speed limit for more than 15 seconds, after which time the train will be automatically stopped. For very high speed, the train will be stopped immediately; Figure 10 illustrates the speed profiles for a single segment. Obviously, these curves serve only to illustrate the profiles. The real ones begin softer for security and comfort reasons, and there are some straight segments, but there are others with curves or street crossings. Then, the profiles are more complicated that those shown in the figure.

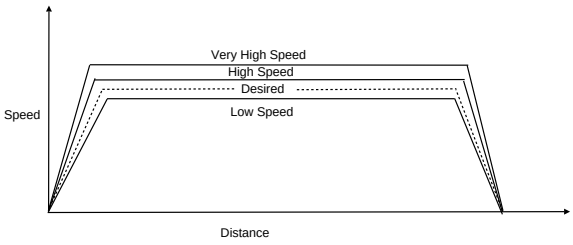


FIGURE 10: *Example of Speed Profiles.*

A real profile taken live from a train during service is shown in Figure 11. This is for the segment from Avila Camacho to División del Norte, where there is a curve about 700 meters after leaving Avila Camacho. The limits shown were simply automatically generated from a hand created desirable profile, and are obviously not good enough for a final implementation. We are working with the supervisors in central control to create actual workable profiles for all segments.

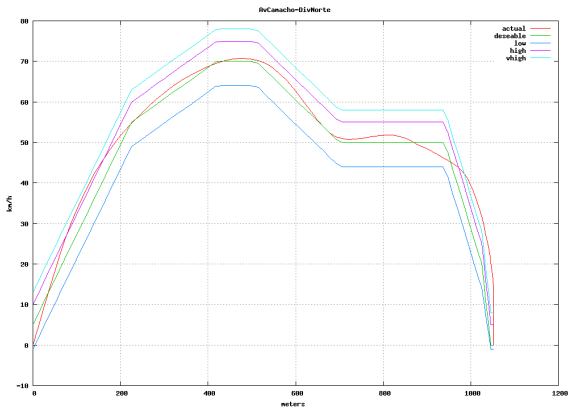


FIGURE 11: *Real Speed Profiles.*

On the other side, in order to prevent the operator of the train from opening the doors on the wrong side, there is a simple table that shows, for each station, on which side the passenger loading platform is. This enables the program to prevent the driver from opening doors where there are not platforms.

5 Standards Compliance

We plan to certify this software for appropriate standards when the money is allocated for the final implementation of the project, and also for the process of standards compliance. At this time, we are only testing the system just described and it is running in parallel with a proprietary safety system that only monitors the train position and displays the position by illuminating the appropriate lights on a large panel with plastic squares.

Part of the trajectory of the trains is underground and the other part is on the surface. For the part on the surface of Line 1, there is no safety system, other than attempting to maintain adequate train spacing and precaution on the part of the drivers. To monitor train spacing, we have developed a system [9] that has been in operation for over a year now, that monitors departure performance and train spacing at the extremes of each line.

In any case, it is the opinion of the authors that a control system based on research methods such as design patterns and N-Version programming, with external review, even if not submitted for standards compliance, will be much safer than the existing conditions at the light rail system. This can (and will) be run in parallel with the existing safety systems until money is available for standards compliance.

6 Conclusion

For developing countries, there is a need to develop low cost solutions for safety critical control systems since the hundreds of millions of dollars needed for commercially developed systems are just not available. We believe that open source software and GNU/Linux can drive down the cost and increase the quality of software systems for safety critical systems just as it has done for commodity server based systems. We believe that it is in the interest of governments to have this source code in the public domain for all to evaluate. Proprietary systems are very expensive and create vendor lock-in. We believe that independent peer review should be possible for public, safety critical, control systems. For proprietary systems, peer review by any independent group is just not possible.

References

[1] G. P. Hubbs, E. A. Mortlock, 2000, *NYCT's communications based train control standard: a look at the leader's system* Proceedings of the 2000 ASME/IEEE Joint Railroad Conference, pp125–133

[2] IEEE, 2004, *IEEE Standard for Communications Based Train Control (CBTC)Performance and Functional Requirements, IEEE Std 1474.1-2004*, IEEE.

[3] IEEE, 2003, *IEEE Standard for User Interface Requirements in Communications Based Train Control (CBTC) Systems, IEEE Std 1474.2-2003*, IEEE.

[4] D. W. Carr, R. Ruelas, J. F. Gutierriez-Ramirez and H. Salcedo-Becerra, 2005, *A Communications Based Train Control System Using a Modified Method of N-Version Programming*, Congreso de Instrumentación (SOMI XX), León, Gto., Mexico, 2005.

[5] D. W. Carr, R. Ruelas, J. F. Gutierriez-Ramirez and H. Salcedo-Becerra, 2005, *An Open On-Board CBTC Controller based on N-Version Programming*, Proc. IEEE International Conference on Computational Intelligence for Modelling Control and Automation (CIMCA'05), Vienna, Austria.

[6] A. Avizienis, 1995, *The Methodology of N-Version Programming*, Software Fault Tolerance, Ch. 2, John Wiley & Son Inc, pp23–46

[7] J. Dugan and R. Lyu, 1994, *System Reliability Analysis of an N-version Programming Application*, IEEE Transactions on Reliability, Vol. 43, No. 4, pp513–519

[8] L. Hutton, 1997, *N-version design versus one good version*, IEEE Software, November/December 1997, pp71–76

[9] H. Salcedo-Becerra, D. W. Carr, R. Ruelas, G. A. Ponce-Castaeda, 2006, *Performance Monitoring for the Light Rail System in Guadalajara, Mexico* Int. Conf. on Dynamics, Instrumentation and Control, August 13-16, 2006, Querétaro, Mexico.

A New Design of RADIUS Client for Integrated Broadband Access Based on Embedded Real-Time Linux

Xiaoyun Chen and Cheng Wan
SISE,Lanzhou University
Tianshui South Road 222,Lanzhou,China
chenxy@lzu.edu.cn, wankao123@163.com

Ding Tang
Institute of Acoustics, Chinese Academy of Sciences
North Fourth Ring Road West 21,Beijing,China
tangd@dsp.ac.cn

Abstract

Through analyzing main wire and wireless broadband access technologies: Web, PPPoE(Point-to-Point Protocol over Ethernet), IEEE 802.1X,IEEE 802.11i and IEEE 802.16, we find that different internet access technology uses different AAA (Authentication,Authorization,Accounting)system which can not make efficient utilization of the network resources, optimize operational networks, integrate user's databases, and also can not provide uniform billings, do business analysis for users. Consequently, a platform suitable for integrated broadband access draws people's much attention. The author attended iBAC(integrated Broadband Access Controller) project putting forward improvements on providing wire and wireless integrated broadband access under IPv6 environment for CNGI(China Next Generation Internet), designed and made RADIUS(Remote Authentication Dial In User Service) client based on wire and wireless integrated broadband access realistic on platform of Real-Time Linux.

1 Introduction

As the construction of broadband network and the number of users increases rapidly, traditional authentication system in which different internet access technology uses different AAA (Authentication, Authorization, and Accounting) system brings users and ISP many problems such as:

1. Because of different internet access technology ISP has to maintain the same information of users in different positions, which increases the cost.

2. ISP can not provide business analysis for users and also can not provide uniform billings.

Thus, to satisfy the urgent need of ISP, a more manageable and more operated network should be developed. We design a new integrated blue point iBAC(integrated Broadband Access Controller) which includes almost all of import wire and wireless internet access technologies.

iBAC stands where various access network meets IP Core network. It provides an integrated AAA service to users who want to visit internet. AAA system is an infrastructure which ISP uses to decide whether users have privileges to visit internet and manage users' accounting information. In this paper we will introduce the designing and implement of AAA system in iBAC. We choose RADIUS(Remote Authentication Dial-In User Service) as iBAC's AAA protocol.

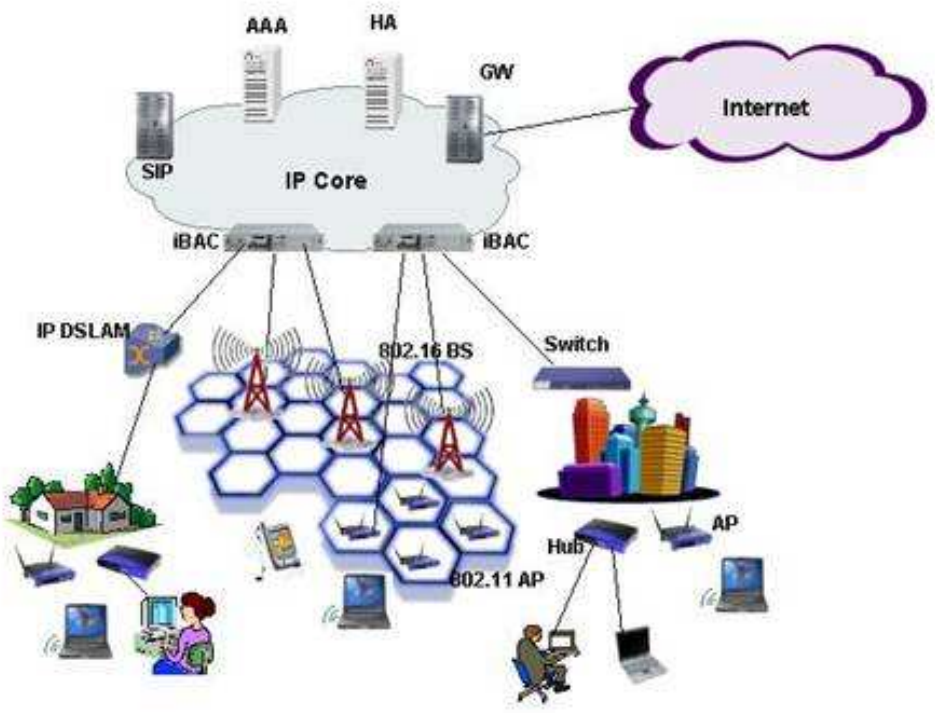


FIGURE 1: *integrated access network topology*

2 RADIUS protocol

Please read Reference [1] [2] to learn more about RADIUS protocol.

RADIUS is an industry-standard protocol for providing Authentication, Authorization, and Accounting (AAA) services.

1. Authentication is the process of verifying a user's identity and determining whether the user is allowed on the network.

2. Authorization is the process of controlling network access values, such as privileges or time limits that the user can exercise with respect to the protected network.

3. Accounting is the process of generating log files that record session statistics used for billing, system diagnosis, and usage planning.

2.1 Client/Server model

RADIUS client is always a NAS(Network Access Server). RADIUS client sends information to RADIUS server, and processes responses from RADIUS server. RADIUS server receives the message from RADIUS client, identifies users, and returns users' configuration information.

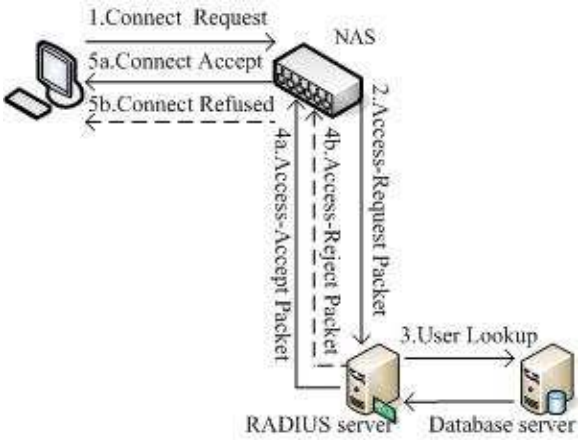


FIGURE 2: *RADIUS Authentication and Authorization sequence*

2.2 Password protocol used by RADIUS protocol

2.2.1 Password Authentication Protocol (PAP)

When the Password Authentication Protocol (PAP) is used, a remote user negotiates with the NAS "in the clear," and no encryption is used to send the password to the NAS. After the NAS has enough information from the user to create an Access-Request, the NAS encrypts the password before sending an Access-Request packet.

2.2.2 Challenge Handshake Authentication Protocol (CHAP)

The Challenge Handshake Authentication Protocol (CHAP) avoids sending passwords in clear text over any communication link. Under CHAP, during password negotiations the NAS generates a challenge (a random string) and sends it to the user. The access client creates a digest (the password concatenated with the challenge), encrypts the digest using one-way encryption, and sends the digest to the NAS.The NAS sends this digest as the password in the Access-Request.

2.2.3 Extensible Authentication Protocol(EAP)

EAP is an authentication framework which supports multiple authentication methods. EAP typically runs directly over data link layers such as Point-to-Point Protocol (PPP), IEEE 802.1X, IEEE 802.11i, without requiring IP protocol.

EAP supports several authentication methods when users use it to negotiate with servers, EAP-TLS, EAP-TTLS, EAP-PEAP etc included. These three EAP all use TLS(Transport Layer Security) protocol and certificate of X.509. Once EAP authentication has been finished, a safe tunnel will be built to protect data safety.

Please read Reference [3] to learn more about EAP,read Reference [4] to learn more about TLS, read Reference [5] to learn more about EAP-TLS.

3 Main access technology

3.1 Web access technology

Web access technology uses relocation technology. Remote users do not need install any client software for authentication, all they need is to input web address in their internet explorer. Web access module will relocate users' internet explorer to login page and lead users to be authenticated.

3.2 PPPoE access technology

PPP(Point-to-Point Protocol) is designed for simple links to transport packets between two peers. These links provide full-duplex simultaneous bi-directional operations, and are assumed to deliver packets orderly. To learn more about PPP, please read Reference [6].

PPPoE (PPP over Ethernet)is PPP running over Ethernet, it builds one PPP link connection between users and NAS. PPPoE integrates the extensibility and the manageability of PPP with the cheap cost of Ethernet. To learn more about PPPoE, please read Reference [7].

3.3 IEEE 802.1X access technology

IEEE 802.1X is called Port Based Network Access Control protocol. IEEE 802.1X message includes EAP message called EAPoL (EAP over LAN), which is transmitted between remote users (called suppliant) and NAS. EAP message is also used between authentication servers and NAS; it is encapsulated into RADIUS packet be transmitted to RADIUS server through complex network. To learn more about IEEE 801.1X , please read Reference [8].

3.4 IEEE 802.11i access technology

IEEE constituted Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications 802.11-1999 in 1999, but the specification can not protect users' privacy information, so IEEE did amendments on ANSI/IEEE Std 802.11, constituted Amendment 6: Medium Access Security Enhancements which is called IEEE Std 802.11i-2004. Please read Reference [9],[10] to learn more about IEEE 802.11 and IEEE 802.11i.

4 iBAC RADIUS client runtime environment

iBAC RADIUS client must provide uniform interface to different access methods (include Web, PPPoE, IEEE 802.1X, IEEE 802.11i). Different access methods use different password authentication protocol. Web and PPPoE use PAP, IEEE 802.1X and IEEE

802.11i use EAP-PEAP.

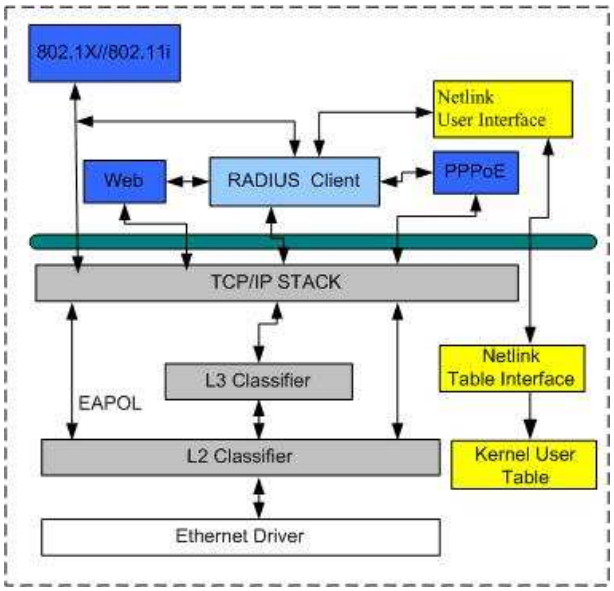


FIGURE 3: *RADIUS client runtime environment*

5 iBAC RADIUS client design in Linux user space

5.1 Initialization procedure

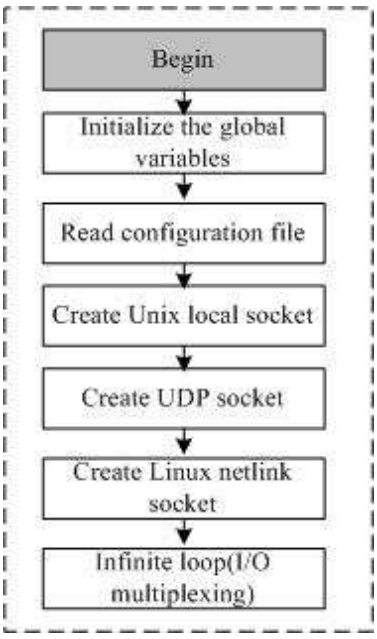


FIGURE 4: *RADIUS client initialize*

1. Initialize global variables, such as some hash tables which are used to store remote users' information(include users' IP address, MAC address etc.),and initialize one timer link list.
2. Read configuration file.
3. Create local socket which is used to do IPC(interprocess communication) with several access modules.
4. Create UDP socket which is used to communicate with RADIUS server.
5. Create Linux net link socket which is used to communicate with Linux Kernel.
6. Infinite loop, use I/O multiplexing and timer link list to control the process logic.

5.2 Timer mechanism

Timer is used in packet retransmission when time is out, and is used to do corresponding process when session time of users is out. RADIUS client maintains a timer link list. Each node in the link list maintains executed time, the interface and arguments of function. Nodes in timer link list are sorted by time. Timer's time is controlled by I/O multiplexing function select () and timer link list. When a node is time out and corresponding process is done, the node will be deleted.

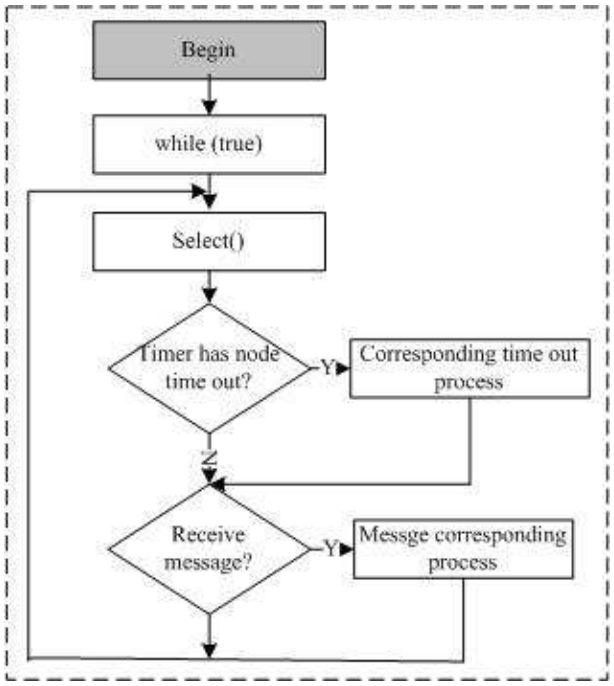


FIGURE 5: *RADIUS client timer mechanism*

5.3 Task scheduling

5.3.1 RADIUS client authentication and accounting start scheduling

RADIUS client has to do different process when using different access technology, the task scheduling is almost the same. Here we take Web access technology as an example.

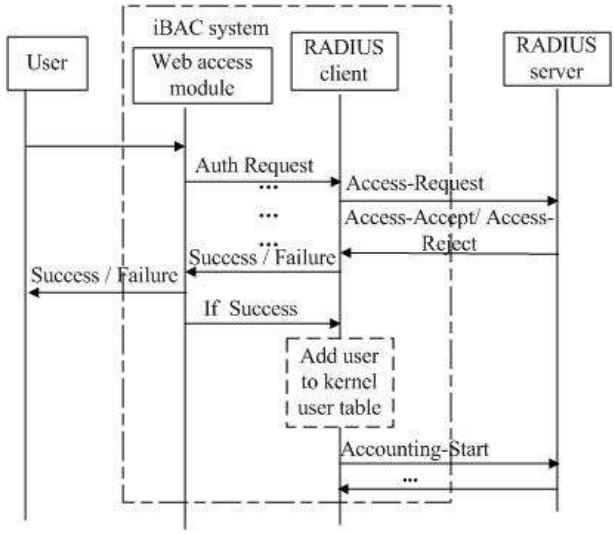


FIGURE 6: *RADIUS client authentication and accounting start scheduling*

1. User logins and sends http packet to Web access module.

2. Web access module encapsulates Auth Request packet which includes user IP address, user MAC address etc. Then sends the packet to RADIUS client by local socket.

3. RADIUS client receives packet from Web access module, adds user into hashtable(each user adds only once, so each time RADIUS client decides weather user is already in hashtable), and encapsulates RADIUS Access-Request packet. The packet is added to timer link list and is sent to RADIUS server by UDP socket.

4. RADIUS client receives Access-Accept or Access-Reject packet from RADIUS server by UDP socket. RADIUS client searches in the timer link list to find the corresponding request packet node to this response packet.

5. If the request packet node is found, delete the node in link list and goto 6; if the request is not found goto 7.

6. If it is an Access-Reject packet, RADIUS client informs Web access module the authentication is failure, Web access module informs user the authentication is failure, and Radius client delete user from the hashtable. If it is Access-Accept packet, RADIUS client informs Web access module the authentication is success, and Web access module informs user the authentication is success, the additional is Web access module informs RADIUS client the authentication is success, then RADIUS client adds user information into Linux kernel user table by netlink socket, at the same time accounting starts, process of accounting packet is as the same as authentication packet, we don't explain it any more.

7. Discard the response packet.

5.3.2 RADIUS client user offline scheduling

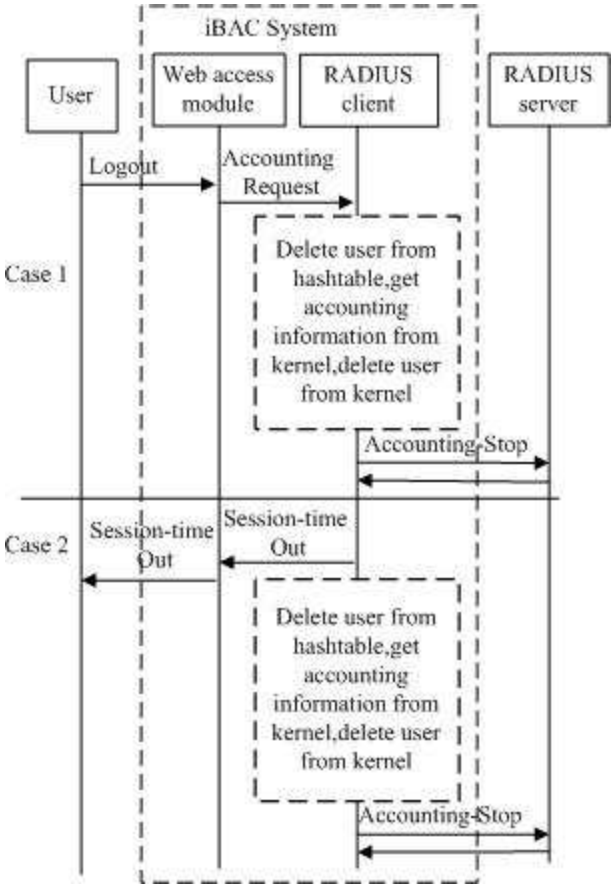


FIGURE 7: *RADIUS client user offline scheduling*

There are tow cases.
Case 1:

1. User logouts and sends http packet to Web access module.

2. Web access module encapsulates Accounting Request packets. Then sends packets to RADIUS client by local socket.

3. RADIUS client receives packets from Web access module. RADIUS client deletes user from hashtable, gets accounting information from Linux kernel by netlink socket, deletes user from kernel user table, then accounting stops.

Case 2:

1. User's session-time runs out, RADIUS client does 2,3 at the same time.
2. RADIUS client informs Web access module Session-Time Out, Web access module informs user Session-Time Out.
3. RADIUS client deletes user from hashtable, gets accounting information from Linux Kernel by netlink socket, deletes user from kernel user table, then accounting stops.

6 iBAC RADIUS client design in Linux kernel space

iBAC maintains an user table in Linux kernel, if user is in this table, it allows user to visit internet, otherwise it denies user to visit internet. Linux kernel does statistic of input packets, output packets, and user time. The statistic will be sent out, when RADIUS client in Linux user space asks for it.

7 Integrated test and conclusion

In CNC(China NetCom), one of iBAC's development organizations, we tested all access technologies in iBAC, and let iBAC be running about 1 week, all the time it was working well. Now we are doing more work and doing higher test. iBAC will be checked and accepted at November 2006.

8 Last Section

References

[1] C. Rigney, S. Willens, A. Rubens, June 2000, *Remote Authentication Dial In User Service (RADIUS)*,IETF RFC2865

[2] C. Rigney, June 2000, *RADIUS Accounting*, IETF RFC2866

[3] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, H. Levkowetz, June 2004, *Extensible Authentication Protocol (EAP)*, IETF RFC3748

[4] T. Dierks, C. Allen, Certicom, January 1999, *The TLS Protocol Version 1.0*, IETF RFC2246

[5] B. Aboba, D. Simon, October 1999, *PPP EAP TLS Authentication Protocol*, IETF RFC2716

[6] W. Simpson, July 1994, *The Point-to-Point Protocol (PPP)*, IETF RFC1661

[7] L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, R. Wheeler, February 1999, *A Method for Transmitting PPP Over Ethernet (PPPoE)*, IETF RFC2516

[8] LAN MAN Standards Committee of the IEEE Computer Society, 13 December 2004, *Port-Based Network Access Control*, IEEE Std802.1X-2004

[9] LAN MAN Standards Committee of the IEEE Computer Society, Reaffirmed 12 June 2003, *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, ANSI/IEEE Std 802.11-1999

[10] LAN MAN Standards Committee of the IEEE Computer Society, 23 July 2004, *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications Amendment 6: Medium Access Control (MAC) Security Enhancements*, IEEE Std 802.11i-2004

[11] W.Richard Stevens,Bill Fenner,Andrew M.Rudoff, 1 July 2004, *Unix Network Programming, Vol. 1: The Sockets Networking API, Third Edition*, China Machine Press,ISBN 7-111-14685-9

[12] Daniel P.Bovet,Marco Cesati, 1 March 2006, *Understanding the Linux Kernel, 3rd Edition*,China Southeast University Press,ISBN 7-5641-0276-4

Research on Real-Time Video Network Transmission System on Linux

Zhiyi Qu and Yachong Guo
Lanzhou University
298 Tianshui Road, Lanzhou,Gansu,China
quzy@lzu.edu.cn
guoyachong@gmail.com

Abstract

A detailed analysis of the key technology of a real-time video network transmission system on Linux is made in this paper, which includes video capture, encoding, network transmission, decoding and playback. The system captures real-time video stream by USB camera, designs video capture program by Video4Linux APIs, according to MPEG4 algorithm, video stream is encoded by libavcodec APIs, then its transmitted by JRTPLIB of stream media program method, decoded by MPEG4 algorithm and played back by SDL APIs. The embedded video transmission platform can be designed by integrating the embedded operating system.

1 Introduction

With the popularity of the Internet and multimedia technology, real-time transmission of video, audio, multimedia animation and other medias has become a hotspot, such transmission technology is known as streaming media transmission technology. Streaming media transmission technology is dedicated to send real-time voice, video, animation and other media documents continuously by the streaming media server for users, so users have not to need to wait until downloading the entire document but only need to wait a few seconds. The remaining part of the documents will continue to download from the server during these media data is palying. MPEG-4 standard with a high compression ratio, high-quality, low transmission rate becomes a major network multimedia transmission formats and standards, the technology of real-time network transmission of streaming media based on MPEG-4 is necessary and meaningful. In a word, streaming media system based on RTP and MPEG-4 has become very important to the development of multimedia technology. With multimedia are becoming more and more important on the Internet, real-time transmission of multimedia is used in many occasions, such as IP telephony, video-on-demand, online meetings and so on.

2 Framework of the system

An integrated real-time video transmission system includes video capture, video encoding, real-time network transmission, video decoding and playback. Its function is to provide service of video applications on UDP-based network. The whole flow of video stream processing and transmission processing is as follows : The system samples analog video at the video transmitter to obtain digital video ,or irectly inputs digital video, then encodes it to generate network communications-oriented video stream that meets certain standards and adapts to the network transmission, the next step is transmitting the encoded bit stream to the video receiver via real-time network, finally decodes the video stream, deconstructs the signal and plays the signal back to output equipment at the receiver.

3 The realization of Video Capture

This module is realized by two steps : The first step is to compile and install the driver of the camera for the Linux kernel, and the next step is to grab video by programming.

1.1 Installation of the driver We use Logitech

QuickCam Web in the system, the driver has not yet been integrated into the mainline kernel.To install the driver for the Linux kernel,the following requirements must be met: a. Kernel ¿= 2.2.18, kernel 2.4.x, or kernel 2.6.x with USB and V4L support. If you are running a version 2.2 kernel, you really need to upgrade to at least 2.4. b. Kernel source for the kernel you are running. The symbolic link /lib/modules/‘uname -r‘/build should point to the source directory. c. A working installation of gcc ¿= 2.95. The procedure is as follows: a.Download the driver that supports the camera. b.Unzip the tarball,enter the directory and compile the source by executing make and make install. Then the compiler will produce a loadable kernel module (LKM). c.If the USB and V4L modules are already loaded, then you can load the module by executing modprobe modulename. The camera can work normally now, you can test it by xawtv. 1.2How to use VFL APIs After the driver is installed, the only need is to prepare an application to capture the video stream. This part will explain the basic steps involved in setting up simple video capture via the Video For Linux (VFL) API. Provided VFL has been setup properly on your system, the file videodev.h should be on your include path at ¡linux/videodev.h¿. First, you will need to include various header files that define structures and constants used by VFL as well as various functions needed to communicate to the VFL device. First, open the video equipment using the system call open to gain the equipment descriptors, the equipment is corresponding to the file /dev/video0.To initialize video capture,the next step is to test some of the capabilities of the device. After testing, set the number of channels, the size and the depth of images and so on. There are two ways to perform frame capturing, The first way is the simplest but it is also the least efficient. The function we use requires our program to wait for a frame, and then it copies the frame into a user provided buffer. In addition, this method may not be supported on all devices. The second method requires some additional setup, but has the added benefit of running synchronously and does not require any buffers to be copied. However, this method may not be supported by all devices either. when we are finished, the final thing we need to do is to free the structures we allocated and disconnect from the VFL device

4 Video Encoding and Decoding

In order to compress the original data at the transmitter, and to decompress the compressed data at the receiver, the system uses libavcodec APIs. libavcodec is a library containing all the FFmpeg audio/video encoders and decoders. FFmpeg is must be installed in the system, if not, install it first. MPEG-4 standard requires low transmission rate and narrow bandwidth, it compresses data through reconstruction of frames and obtains best quality of image through least quantity of data, it provides high compression ratio, at the same time, lesser data losses. We should initialize libavcodec to register all available file formats and codecs with the library at first. Then find the video encoder/decoder-mpeg4 and set the environment of the encoder/decoder according to users needs. Width and height of images, frame rate motion estimation algorithm must be set during encoding. One important piece of information that is stored in the AVCodecContext structure is the frame rate of the video. Then allocate a video frame to store the images that encoder need or decoded images. Most encoders need images in YUV 420 format (one luminance and two chrominance channels, with the chrominance channels samples at half the spatial resolution of the luminance channel) and most decoders return images in YUV 420. Depending on what you want to do with the video data, you may want to convert RGB to YUV or YUV to RGB. Fortunately, libavcodec provides a conversion routine, which does conversion between YUV and RGB as well as a variety of other image formats. After allocating appropriate buffer, we can encode or decode data. when we are finished, all that is left for us to do is to clean up.

5 Real-time Transmission

Real-time Transport Protocol is the standard protocol and key technology for real-time streaming media transmission. JRTPLIB is an object-oriented RTP library written in C++, which offers support for RTP and can be conveniently used in the Linux platform for real-time streaming media programming. It makes it very easy to send and receive RTP packets and the RTCP (RTP Control Protocol) functions are handled entirely internally. The library uses the JThread library to automatically poll for incoming data in the background, so you should install JThread and JRTPLIB. To use RTP, you'll have to create an RTPSession object. To actually create the session, you'll have to call the Create member function which takes three arguments: the first one is of type RTPSessionParams and specifies the general options for the session. One parameter of this class must be set explicitly, otherwise the session will not be created successfully. This parameter is the timestamp unit of the data you intend

to send and can be calculated by dividing a certain time interval (in seconds) by the number of samples in that interval. The second argument of the Create function is a pointer to an RTPTransmissionParams instance and describes the parameters for the transmission component. The third parameter selects the type of transmission component which will be used. By default, an UDP over IPv4 transmitter is used, and for this particular transmitter, the transmission parameters should be of type RTPUDPv4TransmissionParams. Now, we're ready to call the Create member function of RTPSession. If the session was created with success, this is probably a good point to specify to which destinations RTP and RTCP data should be sent. This is done by a call to the RTPSession member function AddDestination. This function takes an argument of type RTPAddress. This is an abstract class and for the UDP over IPv4 transmitter the actual class to be used is RTPIPv4Address. If the library was compiled with JThread support, incoming data is processed in the background. If JThread support was not enabled at compile time or if you specified in the session parameters that no poll thread should be used, you'll have to call the RTPSession member function Poll regularly to process incoming data and to send RTCP data when necessary. Information about participants in the session, packet retrieval etc, has to be done between calls to the RTPSession member functions BeginDataAccess and EndDataAccess. This ensures that the background thread doesn't try to change the same data you're trying to access. We'll iterate over the participants using the GotoFirstSource and GotoNextSource member functions. Packets from the currently selected participant can be retrieved using the GetNextPacket member function which returns a pointer to an instance of the RTPPacket class. When you don't need the packet anymore, it has to be deleted.

6 Video Playback

We use APIs that Simple DirectMedia Layer (SDL) supports to play video back , SDL can visit hardware at a low level and sometimes its more efficient than X11 adapter layer. We should initialize SDL and set video mode at first, such as width, height, depth and so on. Then draw pictures on screen using functions of writing pictures into the frame buffers directly and updating the screen.

References

[1] *Alan Cox Video4Linux Programming..*

[2] *Overview of the MPEG-4 Standard.*

The Design of a Wireless ECG Monitoring System Based on Linux and GPRS

Li Jia, Wu Shui-cai, Li Yan-zheng, and Bai Yan-ping
Center of Biomedical Engineering
Beijing University of Technology, Beijing,China
lijia0813@emails.bjut.edu.cn

Abstract

A kind of wireless ECG monitoring system with real-time detecting and alarm function is described in this paper. The system was designed using S3C2410 and based on Linux. The program can be divided into three parts: data acquisition and display, ECG analyzing, automatic alarm. This paper mostly introduces the program of ECG analyzing and automatic alarm. Experiment results indicate that the system can monitor the patient's ECG real-timely and can detect 5 kinds of arrhythmias. The system sends alarm information automatically through GPRS when ECG signal is abnormal.

1 Introduction

With the development of the society and improvement of people's living , Home Health Care (HHC) is becoming more and more popular, especially in ECG monitoring [1]. At present, MCU (Micro Controller Unit) with 8 bit and 16 bit is usually used in the embedded ECG monitoring system. The ECG monitoring system with 8 bit or 16 bit MCU has some disadvantages, less computing speed, low capacity of storage, lack of interaction. We describe a ECG system based on 32 bit MCU (S3C2410X) ,which uses GPRS (General Packet Radio Service) module to send alarm information automatically when ECG signals are abnormal. The system can monitor and analyze ECG signals real-timely based on Linux.

2 System Design

2.1 Hardware design

The hardware of the system consists of ECG module, communication module and main module. The structure of the system is shown in Fig.1.S3C2410 based on 16/32 bit embedded MCU in the core of ARM920T is used in the system. The dominant frequency of S3C2410 is 203MHz and 10 bit A/D converter is integrated in it. The core of ARM920T consists of ARM9TDMI, MMU(Memory Management Unit) and High-speed Memory. The ECG lead system uses 2-lead ECG. The ECG signals which got from human body surface are sent into S3C2410 to be analyzed and processed after being amplified and filtered. Then the results and ECG waves can be displayed on the LCD. Once detecting the abnormal ECG signals, the system will send alarm information via GPRS automatically.

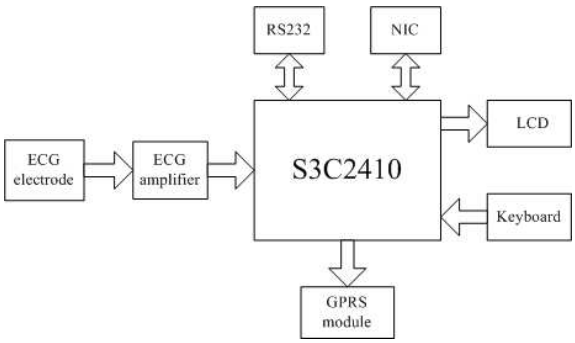


FIGURE 1: *The structure of the system*

2.2 Software design

Linux is a popular operating system (OS) with ample network function, and supports most 32 bit and 64 bit CPU. The source code of Linux is open. Using Linux as the OS is a trend of embedded system in the future [2].

We designed the wireless ECG monitoring system using Linux OS to program based on S3C2410 and GPRS. The program can be divided into three

parts: data acquisition and display, ECG analyzing, automatic alarm. This paper mostly introduces the program of ECG analyzing and automatic alarm which includes the following program modules: sampling interrupt, detecting of QRS waves, arrhythmia analyzing, automatic alarm and so on. The flow chart of system software is shown in Fig.2. The system acquires data every 5ms via the timer of MiniGUI. After filtering out the noise, the ECG waveform is shown on the LCD. At the same time, the system will analyze and monitor the filtered ECG signals. When ECG signal is abnormal, the system will send alarm information automatically through GPRS.

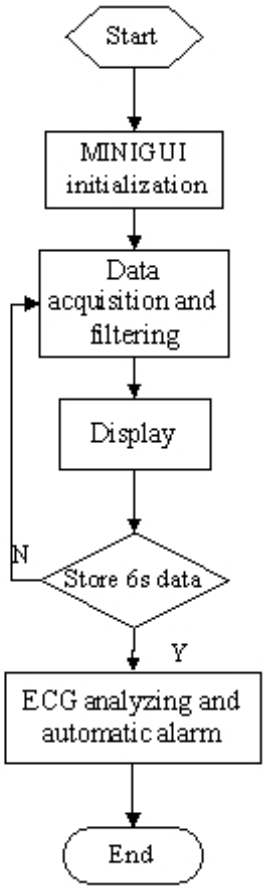


FIGURE 2: *The flow chart of software*

3 Automatic analyzing of arrhythmia

3.1 QRS complex detecting

This paper utilizes threshold method to detect QRS complex [3]. The amplitude of R waves is maximum in the ECG, while others is inconspicuous. The power spectrum of a normal ECG signal has the greatest value at about 17Hz. Therefore to detect the QRS complex, the ECG is passed through a bandpass filter with a center frequency of 17Hz and a bandwidth of 6Hz.

Generally speaking, there is at least one R wave in 1.2s and another R wave in the next 1.2s. Finding the maximum amplitude in these two moments can get the threshold of R wave's amplitude. Once one point gets the threshold, the max point in the next 0.2s is the position of R wave. It's implemented with the following difference equations. Amplitude threshold:

T=0.33(max1[X(n)]+max2X[(n+240)])

if X(n) is ECG sequence. The coefficient is differ from person to person and can be adjusted by experience.

3.2 Arrhythmia analysis

After R_R interval is computed, HR (heart rate) = 60 /R_R (times/min). The system can detect 5 kinds of arrhythmias: Tachycardia, Bradycardia, Arrhythmia, Asystole and Dropped. The threshold of arrhythmia is shown in table 1[4].

Arrhythmias	The distinguish qualifications
Tachycardia	HR>120 times/min,quick_sign>3
Bradycardia	HR<40 times/min,slow_sign>3
Arrhythmia	R_R4-R_R3>R0,R_R3-R_R2>R0, R_R2-R_R1>R0
Asystole	R_R>2.4s
Dropped	2R<R_R<2.4s

TABLE 1: *The distinguish qualifications of arrhythmia*

In table 1, R_R is the time internal of two adjacent R wave, R means the average value of former 4 R_R intervals. Once instantaneous HR>120, quick_sign = quick_sign + 1. When quick_sign>3, the system sends alarm message "Tachycardia" automatically through GPRS. The following is the source code:

```
if (HR > 120) //R_R < 100
{
    quick_sign++;

    if(quick_sign>10)
    {
        quick_sign--; // avoid symbol overflow
    }
}
else
    quick_sign=0;
```

When instantaneous HR<40, slow_sign= slow_sign +1. When slow_sign>3, the system sends alarm information "Bradycardia". If all of the absolute value of differences of the sequence 3 R_R interval is over R/5(R0=R/5) , namely R_R4-R_R3>R0, R_R3-R_R2>R0, R_R2-R_R1>R0, the system will send alarm information "Arrhythmia". The following is the source code:

```
if ((fabs(R_R4-R_R3) > R0) &&
    (fabs(R_R3-R_R2) > R0) &&
    (fabs(R_R2-R_R1) > R0))
    R_buqi=1;
else
    R_buqi=0;
```

When R_R interval is longer than 2.4s, the system will send alarm information " Asystole". When R_R interval is between 2R and 2.4s, the system will send alarm information " Dropped".

4 The program of GPRS alarm

The system can send arrhythmia message through GPRS while arrhythmia is detected . GPRS is short for General Packet Radio Service, and is a kind of data transfer technology based on Global System for Mobile (GSM). Once GPRS is started, it will be online all the time. GPRS is especially suitable for sending discontinuous emergent little data, or sending a lot of data some time. It is suitable for alarming in ECG monitoring.

We use the ARM9 developing system, UPNETARM2410-S, which is made by Beijing Universal Pioneering Technology. We use GPRS module (SIM100-E) made by SIMCOM company. Linux is chosen as the embedded OS, and compiled software gcc is used to program. GPRS affords wireless functions of audio transferring, short message and data service. GPRS module is connected with ARM9 (S3C2410) via RS-232 serial port and is controlled by command AT. The baud-rate of GPRS is 9600 in default condition, and it can also be changed by the function of "get_baudrate(int argc,char **argv)" . Baud-rate can be set at 4800,19200,38400,57600,115200. The mostly function of GPRS module is sending short message to the doctor and patient's family when the abnormal ECG signals are detected. The flow chart of GPRS program is shown in Fig.3.

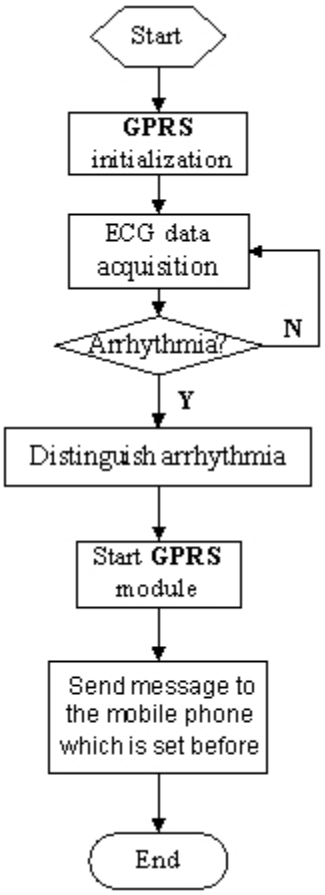


FIGURE 3: *The flow chart of GPRS program*

At first, GPRS module should be initialized. After that, ECG signals are analyzed. When arrhythmia is detected, corresponding alarm message, such as tachycardia, will be sent out, and the system will call the alarm function: void gprs_msg(char *number, int num), and outputs the message of tachycardia. The following is the program code :

```
if ((quick_sign > 3) && (s1==0))
{
     fflush(stdout);

     //calling gprs modulesending
     gprs_msg(numb,text1);

     text1Tachycardia;
     printf("$\backslash$nsending......");

     //after sending message,
     //s1==1the system will
     //go on without alarming again

     s1=1;
}
```

5 Experiment results and analysis

In order to validate the veracity of system designed above, the ECG signals are sampled and analyzed real-timely. Experiment results indicate that the system can monitor the patient's ECG real-timely, detect 5 kinds of arrhythmias and send alarm information automatically through GPRS when ECG signal is abnormal.

References

[1] WANG Congzhi, ZHANG Jupeng, ZHANG Yonghong , et al.,2004, *Development of a Portable ECG Monitor Based on real-time Warning and GPRS Model*, Beijing Biomedical Engineering, pp164–167.

[2] MA Xuewen, ZHU Mingri, CHENG Xiaohui, 2004, *The design of internet communication based on uCLinux and S3C4510B*, Microcontrollers & Embedded Systems, pp30–33.

[3] Willis J.Tompkins, wrote, LIN Jiarui, XU Bangquan, translated, 2001, *Biomedical Digital Signal Processing*,Huazhong University of Science and Technology Publishing House, 7-5609-2579-0/R.21.

[4] HOU Liya , WU Shuicai , BAI Yanping, 2005, *The Development of Embedded ECG Monitor Instrument*, Beijing Biomedical Engineering, pp418–420.

Automotive X-by-Wire System Based on Linux An Open Source Project

Fernando H. Ataide, Alan C. Assis, and Carlos E. Pereira
Departamento de Engenharia Elétrica
Universidade Federal do Rio Grande do Sul
90035-190 Porto Alegre, RS, Brasil
{fhataide, acassis, cpereira}@ece.ufrgs.br

Abstract

Important researches have presented different approaches in the real-time embedded communication domain aiming to cover the growing demands of performance, predictability and reliability of emerging applications. Such requirements involve low latency, reduced jitter, time composability, fault-tolerance and finally, support for future extensions. In agreement, the FTT (Flexible Time-Triggered) model is an approach which aiming to provide flexibility on the time-triggered traffic together with timeliness, differently of others protocols. This paper present some results of an enhanced FTT-CAN implementation for automotive application which was developed over an embedded Linux with RTAI extension. Additionally some design issues are provided and discussed.

1 Introduction

The drive task in vehicle, aircraft, among others that are carry through of mechanical and/or hydraulic systems are being replaced for electronic intelligent systems focusing on weight reduction, costs, maintainability, among others factors. These electronic systems are called drive-by-wire, e.g. steering system (steer-by-wire), braking system (break-by-wire) and flight control system (fly-by-wire).

The communication platform is the main component of these kind of systems. In this domain, severe timing constraints predominates hence the need of predictable service behavior in both time and value dimensions even in the presence of faults as the system can be strictly involved with either integrity of people or expensive equipment. The main requirements involved in Real-Time Embedded Communication Systems (RTECS) applications are fixed protocol latency with low jitter, composability, support for fault-tolerance techniques and also some degree of flexibility in order to account for future modifications during the design phase.

In this context, some existing protocols claim being able to fulfil these requirements. The first one is the TTP/C [4] protocol and its Time-Triggered Architecture (TTA) – a result of many years of work started in 1979 at the Vienna University of Technology with the MARS project. The other one is the FlexRay Communication System [5] developed by the FlexRay Consortium, formed by a group of strong companies of the automotive area. A common characteristic of these protocols is their static communication behavior. Bus arbitration is essentially TDMA-based so the transmission times of all messages must be known in advance. Each network station has a fixed time slot for transmission that must be specified at design phase. The FlexRay protocol in particular combines both time- and event-triggered traffic by defining a dynamic bus access window in which bus arbitration is performed in a FTDMA (Flexible Time Division Multiple Access) scheme thus providing some degree of flexibility. This feature is important for many of modern digital control systems (e.g. industrial control process and automotive systems) mainly when the controlled object dynamics cannot be completely defined before runtime thus requiring the control system to be adaptive. A RTECS with a high flexibility degree can provide a more efficient resource utilization, enhanced maintainability and easy integration/allocation of new functions in response to new functionality demands. The reader is referred to [6] for a detailed overview about the main advantages of providing enhanced flexibility support in distributed

control systems.

The FTT-CAN [7] (Flexible Time-Triggered Communication on Controller Area Network) consists in a complete communication platform which was proposed as a solution to meet the flexibility requirement with guaranty of timeliness and efficiency. It combines a dynamic time-triggered with event-triggered traffic by means of temporal isolation of bus access time over CAN protocol. In contrast to TTP/C and FlexRay, which have a static time-triggered schedule, all communication load can be dynamically scheduled at runtime. However, the FTT-CAN protocol has inherent drawbacks which affect its response-time. The first one is the jitter on its time-triggered phase generated by bit stuffing method used in the physical encoding [8] of CAN. Other one is the blocking condition that generates messages priority inversion. This condition is caused by release-time variation on message dispatch process of the FTT-CAN stack. These problems have been already argued in a previous paper [3] which identifies and adopts methods to improve these drawbacks.

In a embedded real-time application the choice of a suitable Real-Time Operating System (RTOS) may be decisive in all life cycle of the project. In this project we adopt one to reach more reduction code size, maintenance improvement, reuse, and reduction of the implementation complexity. And mainly, this RTOS have to provide hard precision on the scheduler considering low latency and variability of the context switch and interrupt handler.

This paper is organized like follows. It starts with a brief introduction on the Baja-by-Wire project and FTT-CAN protocol, followed by details of the software and communication architecture with its mains components. Finally, we argue the practical results of the communication stack implementation followed some conclusions.

2 The Baja-by-Wire Project

This work is part of a drive-by-wire project, called Baja-by-Wire[1], which is being developed at the Electrical Engineering Department of UFRGS. The main goal is to develop a drive-by-wire dynamic system for an off-road race car. The FTT-CAN protocol was chosen as the underlying protocol due to its enhanced support for operational flexibility requirements. The necessary fault tolerance mechanisms for FTT-CAN system will not be argued in this paper because it is not the our focus here. However the reader is referred to [1] and [2] for details about some purposes around fault tolerance techniques in FTT-CAN. A simplified scheme of the car electronic system with its Electronic Control Units (ECUs) and dashboard are depicted in Figure 1.

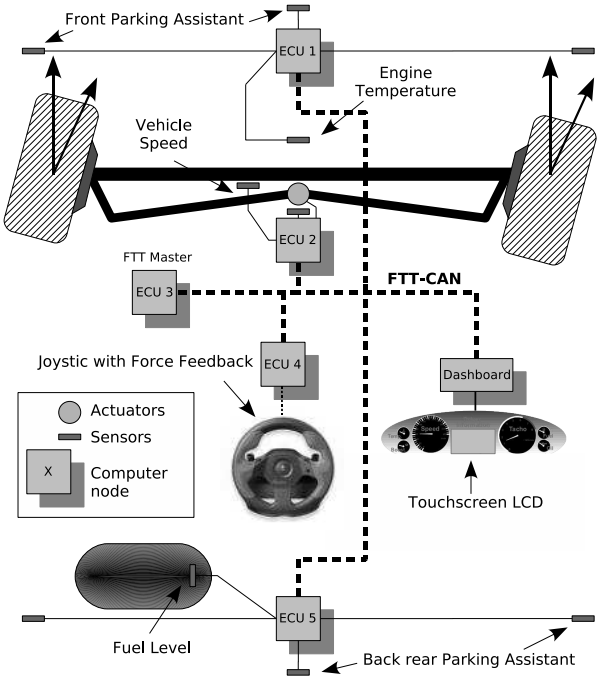


FIGURE 1: *Baja-by-Wire system architecture*

This by-wire system encompasses a steer-by-wire sub-system (ECUs 2 and 4) with a dashboard computer (ECU 5) which collects telemetry information and can make some diagnosis. Additionally, an interactive visualization of the parking assistant subsystem was proposed, which have its functionalities distributed among ECU 1 and ECU 5 with proximity sensors (fixed in the back and front of the vehicle) and dashboard computer that it will present a screen image of the distributed sensors behavior during a parking situation.

2.1 Hardware Platforms

The project has two distinct hardware platforms, one for ECUs which are components present in the electronic body and electronic systems of the vehicle – parking assistant and steer-by-wire sub-systems respectively. And other hardware platform for the dashboard computer.

Each ECUs is composed by an embedded hardware based on Freescale Coldfire 5282 66MHz microprocessors, which has one CAN controller built-in, 16MB of RAM and 2MB of Flash memory.

The dashboard hardware platform is an embedded main-board powered by Freescale PowerPC 5200

[1]This project has been partly supported by the Brazilian research agency (CNPq) and FreeScale Semiconductor of Brazil.

processor. This board has 64MB of RAM and 32MB of Flash memory, the processor runs at 400MHz. The main devices in the processor are: CAN, PCI (Peripheral Component Interconnect) and Ethernet controllers built-in. Attached to PCI slot is a common 3.3V video card, this card is connected to 8" LCD widescreen touchscreen.

The software platform for these hardware is based on Linux with RTAI (Real-Time Application Interface) extension, and both need to support the FTT-CAN communication stack. Protocol overview and software architecture are detailed in the next sections.

3 FTT-CAN Protocol Overview

The FTT-CAN was proposed to cover the flexibility requirement on critical systems through online message admission control of the time-triggered traffic without jeopardizing the overall system timing behavior [7]. Admission control is performed by a central scheduler running at a particular node – called master node. It can receive dynamic transmission requests which are processed and evaluated by a feasibility algorithm.

The time-triggered traffic is based on a relaxed master-slave medium access control. A specific message – the Trigger Message (TM) – is transmitted by the master node in order to trigger the beginning of an Elementary Cycle (EC), inside each slave station going to transmit their messages either in the event-triggered (ET) phase or in the time-triggered (TT) phase. The TM carries schedule information that indicates which producer or consumer task must be released and which messages must be present in the next TT phase as is depicted in Figure 2.

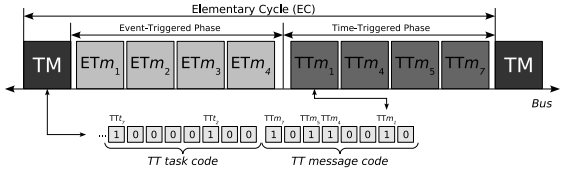


FIGURE 2: *The Elementary Cycle (EC)*

Eventual bus access collisions between slaves messages are resolved by the native arbitration mechanism of the original CAN protocol. The TT message exchange of FTT-CAN follows a producer-consumer model whereas the ET mode offers only send and receive basic services to be used when the application layer has aperiodic data to deliver. Slaves with pending aperiodic messages may transmit only during the ET phase which is the remaining time that is not used for TT traffic in an EC. In the FTT-CAN, periodic message exchange takes place in an autonomous manner, i.e. the protocol stack process is responsible for the transmission of all messages inside the TT phase hence the application tasks do not need to invoke send and receive primitives. In the ET phase, however, messages are transmitted in response to explicit requests from the application layer.

4 ECU Software Architecture

The use of a RTOS is mandatory in certain circumstances when multiple instruction flows must be multiplexed for execution in a single processor with guarantees of timeliness. Depending on the complexity of the application, the use of a single execution flow to accommodate all system functions may result in unclear code and a time-consuming design phase. For that purpose a RTOS provides many advantages such as reduced code size, maintenance improvement, reuse, and so on. As a result of a comparative analysis and considering our hardware (with ColdFire microprocessor) the choice was to adopt the μClinux[2] [9] together with a real-time extension, the RTAI [10] (Real-Time Application Interface). An advantage of Linux-based operating systems are their modular kernel architecture which increases the flexibility of the system at runtime. In other words, kernel modules can be dynamically loaded for a particular situation, for instance, an Ethernet device driver and protocol stack can be loaded whenever necessary in order to provide gateway functionality over a short period of time. The factors that consolidated that choice were: 1) μClinux and RTAI are open source project; 2) they have a mature code base; 3) there are free tools available, 4) small footprints; 5) active development community and 6) support to a wide variety of architectures such as x86, PowerPC, ARM, MIPS and m68k. μClinux together RTAI gives us all present features of a RTOS.

[2]Linux operating system for microcontrollers without Memory Management Unit (MMU)

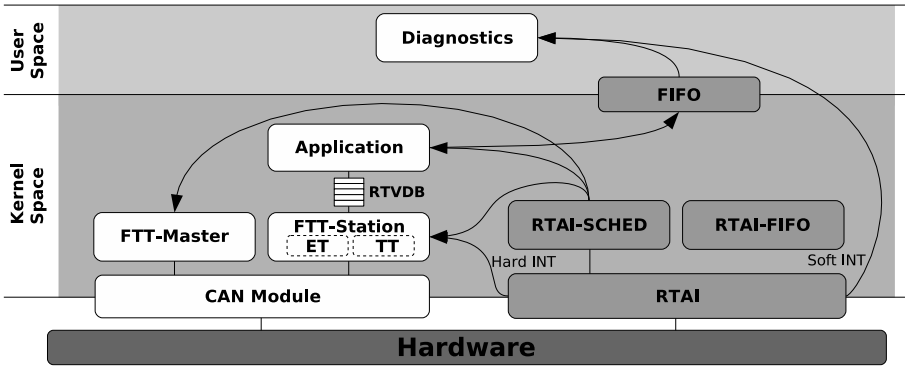


FIGURE 3: *ECU software modules*

Since the RTAI extension provide means to add time-critical functionality to the μClinux kernel, the FTT-CAN protocol stack and application software for the ECU were included in the Kernel Space Level aiming to ensure proper real-time execution behavior. Likewise, on the User Space Level it was decided to attach diagnostic tasks which are not time constraining. The logical architecture of the implementation is depicted in Figure 3 in which the arrows demonstrate the relationships between its components, where RTAI modules and device are the gray blocks and developed modules are the white blocks. The RTAI extension has its own modules providing a set of basic services which allow the implementation of fully preemptive scheduling. On the left side the communication protocol stack is shown and on the top is the `Application` module that represents the process control tasks like sensor reading and calculation of output results. At the kernel initialization procedure of slave nodes must be loaded with the `FTT-Station` module whereas the FTT-CAN master must be loaded with the `FTT-Master` module.

4.1 Communication Stack

On the base of FTT-CAN stack in the Figure 3 there is a `CAN Module` which supply some methods to access an CAN controller internal registers. It was decided to code a dedicated primitives aiming to reduce additional overhead that would be generated if a COTS driver was used. The `CAN Module` consists only in a restricted set of access primitives which provides means for both the `FTT-Station` and `FTT-Master` to interface with the controller.

The `FTT-Station` module is implemented as two high priority tasks (ET and TT in Figure 3) and it is responsible for all message exchange on the system. For instance, it accesses data base either to update it with new data from the network or to load one sample for transmission inside a CAN message. The RTAI interrupt service is the main resource of this module. Each time a new Elementary Cycle is started by the transmission of the TM, an interrupt raises to trigger the execution of the `FTT-Station` module. Thereafter the contents of the TM is decoded for the module to know which Real-Time Variable (RTV) in the application level must be sent in the next TT phase and which Real-Time Task (RTT) must be released in the respective EC.

A RTV is any application variable whose validity is restricted in time. Each RTV data structure has the following attributes:

- Identifier
- Data size
- Pointer to the data buffer
- A flag indicating whether it is produced or consumed
- Validity information over time
- Pointer to its corresponding real-time task
- A flag indicating whether it is a TT or an ET variable

The set of RTVs of interest in a station node is allocated in the Real-Time Variable Database (RTVDB) representing the interface between application software and the protocol stack (Figure 3). Also known as Communication Network Interface (CNI) in others protocol, such as TTP/C and TTP/A. The creation of the RTVDB in memory is performed during the startup phase. Once the `FTT-Station` is informed about the reception of the TM it starts an internal timer in order to schedule the beginning of the following TT phase at the appropriate point in time that was calculated. ET messages which has been previously load for transmission

are sent if and only if there is enough time before the TT phase begins.

On Message exchange level of TT phase the control is autonomous, i.e. the transmission and reception is performed by FTT-CAN protocol level without interference from of application level. The transmission sequence is carried out through a buffer defined at TM message decoding moment. In ET phase the message exchange has implicit requests of application level through a specific API in the external control way. Each request is queued in agreement message identifier which define the message priority. Finally, the `FTT-Master` module is responsible for the dynamic message scheduling of the TT phase. Prior to sending the TM message it defines the sequence of messages to be transmitted based on a schedulability analysis method that takes into account timing requirements like period T_i, priority P_i, worst-case transmission times C_i, relative phase Ph_i and deadline D_i. On the other hand aperiodic traffic is scheduled based on fixed priority methods.

5 Dashboard Software Architecture

In this project we propose to use a set of hardware and software together with LCD touchscreen to replace analog meters and gauges on the automotive dashboard. Although this solution can has a cost increase regarding to mechanical dashboards this approach provide some improvement to automotive dashboards, which are:

- No mechanical problems caused by wastage;
- Themability - the dashboard's apparency can be defined, look like mp3 player skins;
- Configurability - the meters and gauges can be placed or removed to/from panel;
- Customization - the rangers to meters can customized, i.e. new ranger to low fuel level;
- Alarms - alarm can be created to indicate warnings or error conditions.

Nowadays mechanical parts used in the automotive panels get wastage's problems caused by attrition, twist and pressure. With this new approach these problems can be eliminated. Themability is an interesting resource where user can define the "look and feel of your panel". Then user can choose an apparency which identify himself and he can select what meters and gauges will be displayed at your panel.

In the some case user can customize dashboard's behavior to display information in accordance with his necessities. For example when driving for long freeway where have no gas station the user can reconfigure the low fuel level indicator and alarm to warn him very before than the same situation on driving on cities.

Using a LCD touchscreen the user can do all customization through a just touch on the screen selecting desired features.

5.1 Software components

The dashboard software running over Linux with RTAI extension. All application software runs in the User Space Level, whereas the FTT-CAN communication stack runs in the Kernel Space Level due to its necessity for previsibility.

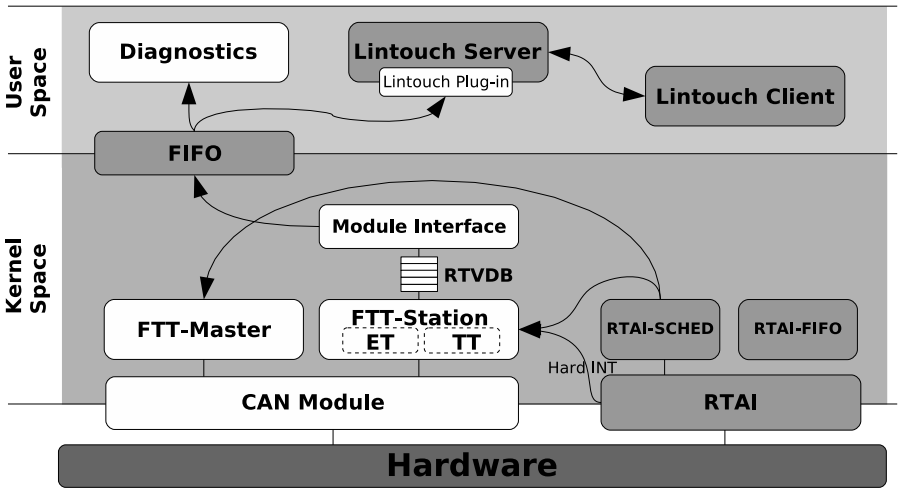


FIGURE 4: *Dashboard software modules*

The application software catches all data from `Module Interface` through a real-time FIFO, and the `Module Interface` in its turn catches from FTT-CAN stack.

Instead develop all graphics screen components of the application software from scratch this project propose a new approach: to use a lightweight Human-Machine Interface (HMI) to show all telemetry informations. These informations are: vehicle speed (produced by ECU 2), fuel level (produced by ECU 5), engine temperature (produced by ECU 1) and the screen image of parking assistant sub-system which is mounted according to the proximity sensors value. The Figure 4 shows the software architecture of the dashboard system.

This HMI is Lintouch[3], it is a open source software mainly used to interface process automation. Lintouch has some features which are very important to the project:

- **Multiplataform -** Lintouch was been developed in C/C++ using Qt graphic library, so it has few dependencies. Currently it run on Intel processor (on Windows and Linux) and PowerPC processor (on Linux).

- **Lightweight -** as referred before, it is so much lightweight. The server size is approximately 100KB and the client is approximately 900KB. 32MB of RAM is sufficient to run both server and client on the same machine.

- **Extensible -** Lintouch architecture is modular, each resource (plug-in and/or template) can be added easily.

- **Open source -** Lintouch is open source software. All source code is released under the GNU GPL.

To use Lintouch in this project was necessary to develop a plug-in to get all telemetry data from real-time FIFO and make available to the Lintouch environment.

The Figure 5 present a screenshot of the current dashboard graphic user interface. The screen shows some Lintouch dial components displaying the vehicle speed, engine temperature, fuel level and a clock.

[3]Access the Lintouch project at www.lintouch.org.

FIGURE 5: *Dashboard screenshot*

6 FTT-CAN Stack Responsiveness - Practical Results

The purpose of this section is to evaluate the FTT-CAN responsiveness developed over the RTAI/μClinux running at each ECU. The dashboard node is not evaluated because it only consumes messages (data) from the network. The RTAI interrupt latency and its jitter have a direct impact on the protocol response time. This developed stack present some improvements concerning to the standard FTT-CAN proposal, however this subject is not focus in this paper. This improvements are presented in [3].

For approach evaluating was considered the set of periodic (time-triggered) messages of the Baja-by-Wire system. Some of these messages are involved in the control loop of the steer-by-wire sub-system, which are the messages 1, 2 and 3 of the Table 1.

	Description	Node	P_i	T_i (ms)	Ph_i (ms)
1	Angle for actuation	4	7	5	0
2	Current wheels Angle	2	6	5	0
3	Current speed	2	5	5	0
4	Current engine temp.	1	4	100	50
5	Current fuel level	6	3	100	50
6	Front sensor	1	2	200	100
7	Back rear sensor	6	1	200	100

TABLE 1: *Set of TT messages*

The FTT-CAN elementary cycle (EC) inherits the lower messages period, in this case $5ms$. Measured release jitter is presented to the most affected application messages[3]. These messages are 1, 3, 5, 7 and the TM transmitted by FTT Master node. In accordance with Table 1, messages 5 and 7 are always the first ones in a given TT phase in an EC due to its

priority (P_i) and relative phase (Ph_i). And messages 1 and 3 are in the end of the TT phase, however, they will suffer the bit stuffing effect. All measurements were made for a set of 3000 sampled messages, with a CAN bus at 250kbps and 4μs of timer resolution.

Message	J_{max} (μs)	J_{min} (μs)	σ
1	20	-16	4,00
3	16	-16	3,53
5	4	-4	2,80
7	8	-4	2,53
TM	4	-4	2,02

TABLE 2: *Obtained Results*

The Table 2 present the results through maximum (J_{max}) and minimum (J_{mim}) release jitter in μs and standard deviation (σ) of the selected messages in a FTT-CAN implementation described in this paper. As can be observed, the FTT-CAN stack over the RTAI/μClinux has an excellent response time with hard precision. Therefore, the results justify FTT-CAN applicability for this system domain.

Additionally the proposed approach presented in [3] enables some advantages regarding to standard FTT-CAN, which are:

- the use of periodic messages with different data length, thus eliminating a restriction of the standard FTT-CAN;
- and also it guarantees the response-time of all periodic messages even in a fault condition in a node, in other words, it fixes the release-time for each TT message.

7 Conclusions

The paper presented an enhanced FTT-CAN protocol implementation for an automotive system with a spot over the performance obtained by implementation, which was developed over an embedded real-time Linux.

It is very important the choice of a suitable RTOS with hard precision on the scheduler and interrupt handler, which may be decisive in all life cycle of project. The RTAI/μClinux has presented excellent performance for hard real-time application.

This project was developed entirely with open source tools.

References

[1] Marau, R., Silva, V., Ferreira, J., Almeida, L., 2006, *Assessment of FTT-CAN master replication mechanisms for safety-critical applicationse*, SAE CONGRESS 2006 paper n°: 06AE-278

[2] Marau, R., Silva, V., Ferreira, J., Almeida, L., August 2006, *Using FTT-CAN to combine redundancy with increased bandwidth*, IN PROCEEDINGS 6TH IEEE INTERNATIONAL WORKSHOP ON FACTORY COMMUNICATION SYSTEMS

[3] Fernando. H. Ataide, Carlos. E. Pereira, Walter. F. Lages, and Alan. C. Assis, August 2006, *On the design of an embedded FTT-CAN platform with improvement of its inherent jitter*, IN PROCEEDINGS OF THE 4TH INTERNATIONAL IEEE CONFERENCE ON INDUSTRIAL INFORMATICS, INDIN

[4] TTA-Group, November 2003, *Time-Triggered Protocol TTP/C High-Level Specification Document Protocol Version 1.1*,TTA-Group

[5] FlexRay Consortium, 2004, *FlexRay Communications System Protocol Specification Version 2.0*, FlexRay Consortium

[6] Almeida, L., January 2003, *A word for operational flexibility in distributed safety-critical systems*, IN PROCEEDINGS OF THE EIGHTH INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS (WORDS 2003), pp177–184

[7] Almeida, L., Pedreiras P., Fonseca J., December 2002, *The FTT-CAN Protocol - Why and How*, IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, pp1189–1201, n6, v49

[8] CiA, 1999, *CAN physical layer*, http://www.can-cia.org/can/physical-layer

[9] Dionne J., Durrant M., 2002, *Embedded Linux/Microcontroller Project*, http://www.uclinux.org

[10] Mantegazza P., 2001, *The DIAPM RTAI project homepage.*, http://www.rtai.org/

An Extensible Object-Oriented Instrument Controller for Linux

D. W. Carr and R. Ruelas
Departamento de Ingeniería de Proyectos, Universidad de Guadalajara
Apdo. Postal 307, C.P. 45101 Zapopan, Jalisco, MEXICO
doncarr@gmail.com, rruelas@newton.dip.udg.mx

R. Reynoso-Orozco and A. Santerre
Departamento de Biología Celular y Molecular, Universidad de Guadalajara
Carretera Nogales Km. 15.5. Las Agujas Nextipac, Zapopan, Jalisco, MEXICO
rreynoso@cucba.udg.mx, asanter@cucba.udg.mx

Abstract

We have developed a simple-to-program, extensible, Linux based control engine (REACT) to make programming of a chromatograph and other applications as simple as possible, specifically for scientists or others who are not experts in programming or control. The developed system is programmed by creating objects that represent real-world components such as valves, pumps, and motors, and it is based on an object script language to control the instrument and record data. The result is a chromatograph with a control system where modifications to the program are extremely easy to implement.

1 Introduction

In the Chromatography Laboratory of the Cellular Biology Department, ionic interchange chromatography is applied as a way to study the effects produced by natural products in the cancer cellulae, and in the healthy tissue in the Linfoma Murino Model L5178Y [1]. This methodology is used in the separation and quantification of polyamines [2]. The polyamines are molecules that have many applications in the study and treatment of diseases, particularly in neoplastic processes [3], as the one previously mentioned.

The specific application for liquid chromatography is to detect the presence of various chemicals in the urine of mice, so that various cancer treatments can be evaluated. Here, the scientists understand how the chromatograph works, and need to define the inputs and outpues, and also create the experiment sequences. The experiment sequences involve closing and opening valves, turning pumps on and off, injecting the sample, prompting the user, etc.

In 2003, we undertook the project to develop an instrument controller for an existing chromatograph that had very limited functionality, specially because it only saved the result in printed format on thermal paper. One big problem was that the raw data was thrown out after analysis of data and printing of the results. The analysis consisted of integrating the area under the peaks, but, it was impossible to verify the analysis independently since the raw data was lost.

Instruments can often be very simple in their basic design, but a packaged commercial instrument with electronics, computer, and software can be quite expensive and out of reach, especially for scientists in developing countries. Also, it could be very difficult, as in the case we are dealing with, to extract the raw data for analysis, visualization, and sharing.

The goal here is to create an extensible Linux based control engine with object scripting such that, persons without advanced control and programming experience, can easily create experiment controllers and save data in order to analyze the data, and share the data with peers and other interested parties. This gives the end user (scientist, engineer, or other person that is not necessarily a control and/or programming expert), the ability to quickly create control sequences for instruments, batch control, testing, manufacturing stations, as well as recording and handling data.

The second section presents the REACT system which was used for the control system. In third section we give some examples of the programmed objects, whereas in the fourth section we show an

example of the results obtained with the chromatograph which is controlled by the system described here. The last section gives some conclusions regarding the results of this work.

2 The REACT system

React is a control engine applied to the control of a chromatograph. React is an extensible object oriented control engine with an object-script language that allows new object types to be created. React also supports numeric expressions whose return type can be either analog or boolean. The basic object types are discrete input (DI), discrete output (DO), analog input (AI), analog output (AO), pulse count input (PCI), timed discrete output (TDO), PID control (PID), and data collection objects as described below. React is tagname based, and all objects are assigned a tagname. These tagnames, are then used to identify objects, and execute script functions on individual objects using the script functions. Drivers and user defined object types are loaded dynamically as shared objects when React is started. React then reads the basic analog inputs, discrete inputs, and counters at regular intervals specified by the user (see Figure 2). After the basic inputs, the rest of the objects are updated in the following order: 1) analog and discrete calculations, control objects, and user defined objects. Finally, all active scripts are executed as described below.

Executing scripts — Each update cycle, currently active scripts are given a chance to execute. Scripts execute continuously until one of the following types of functions is reached: wait, wait-until, getkey, or any user defined script function that returns "call again". It is up to the script writer to create scripts that will not execute more than what can be done in one update interval, though this would typically only be an issue with very short update intervals. Scripts are tokenized on load so that they will execute faster. When a script reaches the end, it is terminated. For a batch, testing, or experiment type system, when the last script reaches the end, this is the end and and React stops.

Among the alternatives for programming an instrument are: directly programming in C/C++, or other structured languages, the IEC 61131-3 languages [6] which are: ladder logic, function block diagram, structured text, instruction list, and sequential function chart, Statecharts [4][5], Labview [7], and more than we can mention here. Obviously C/C++ are only for programmers, ladder logic and the other IEC languages are typically for persons with electrical and control experience, Labview (a graphical environment) is also quite complicated and not amenable to non technical users. Probably one of the most interesting options here is Statecharts, also similar in concept to sequential function charts, which is a graphical environment using state diagrams to define the control sequences. We are actually planning on incorporating Statecharts like functionality into React, using our object scripting language and expressions to augment the functionality, as an alternative to only scripts. We believe what has been missing is a simple scripting language for creating instrument sequences, together with a simple environment for defining and naming the inputs, outputs, and data collecting objects. This in a nutshell is what React is about.

Another feature of REACT is that we can save files to a directory that is viewable from a web browser via Apache, or the files can be synced to a public web server using "rsync" if it is not desirable to run a web server on the local machine running the control engine. This is what is done at the University of Guadalajara. A graph of the output of the chromatograph for an experiment is shown in the results section in this work. However, you can currently view all the results of the experiments from the Cellular Biology Department run since 2003 at:

```
http://alumnos.cucba.udg.mx/~croma/
```

For an example experiment with graphs see

```
http://alumnos.cucba.udg.mx/
      ~croma/equipo1/2006/04/26/exp02/
```

At this address, you can see five graphics generated by GNU Plot. These are: opa.png, and peak1.png to peak4.png. The first (opa.png) is the voltage output of the chromatograph for the entire experiment. The four other graphics are the details of the peaks for the integration of the area (volt-seconds). The correct quantification of the peaks is important because each area represents the concentration of substances in the sample. Figure 1 shows the of seven experiments run on standards to determine the relationship between area under the curve and concentration of a substance in a sample. This is basically how the instrument is calibrated. You can see the linear relation between the area under the curve and the concentration. The file opa.txt is the raw data from the test run that can be used by others to verify the calculations and analysis. We also create files with the raw data used to generate the graphs of the peaks and do the numerical integration: peak1.txt to peak4.txt.

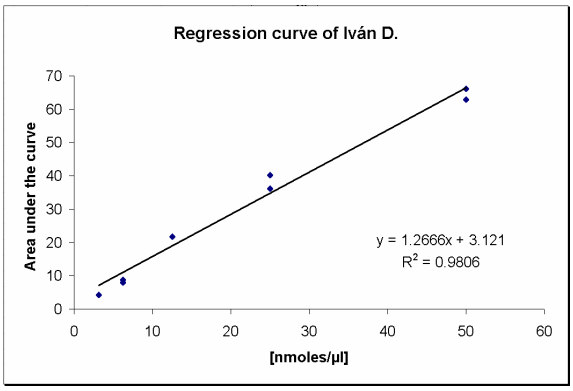


FIGURE 1: *Integration results.*

3 Examples of Linux based control engine

In this section we show how input/output and data collection objects are defined and the syntax of scripts. We show how the data collection objects can be used to collect single scans or scan continuously at a predetermined rate. Also, we show the scripts which are global or object specific.

Data collection objects that can scan a set of objects on demand are commonly needed for experiments and tests. These objects sample a group of objects on demand when the scan function is called. An syntax of the definition is show, followed by and actual example, and then the call to perform a single scan.

```
<tagname>,<description>,<file name>,<tagname 1>,
     <tagname 2>, . . . , <tagname N>

scan1|My Scan Point|sout1|10|AO0|AI0|AI1|

scan1.scan()
```

The second data collection type continuously samples a group of objects at a predefined interval of time, where the start time and the stop time are controlled by script functions. Following we first show the syntax for the definition, followed by at actual example, followed by example calls to start and stop data acquisition.

```
<tagname>,<description>,<file name>,<tagname 1>,
     <tagname 2>, . . . , <tagname N>

data1|My Data|dout1|25|AO0|AI0|AO1|AO2|

data1.start(10)
data1.stop()
```

There are global (system) script functions and object specific script functions. First we show the syntax for global/system script functions:

```
sys.<function-name>(parameter 1,
    parameter 2, . . . , parameter N)
```

Following is the syntax for a script function specific to an object:

```
<tagname>.<function-name>(parameter 1,
    parameter 2, . . . , parameter N)
```

It is also possible to run one script from another using the run command of the following form:

```
run.<script-name>(parameter 1,
     parameter 2, . . . , parameter N)
```

Below we show the actual scripts used to control the chromatograph. The first script shown (Example 1), is the script called to start the experiment when just one sample is loaded. This script prompts the user to be sure that the instrument is set up correctly, then calls corridastd.txt with two parameters which are the wait times for two phases of the experiment that can be changed. Note that EV1, EV2, and EV3 are the valve outputs, and BOMBA1 and BOMBA2 are pumps that can be turned off and on. NOTHOLD is a signal to control the injection of the sample. When NOTHOLD is asserted, the sample is injected. The system contains a robotic arm such that multiple samples can be loaded and run in batch. The example shown here is for a single sample.

```
Example 1. (std1.txt)
NOTHOLD.send(F);
sys.print("Estar seguro de tener")
sys.print("     1 muestra cargada")
sys.print("Checar que tengas")
sys.print("      suficientes fluidos")
sys.print("Para el inyector necesitas: ")
sys.print("Seleccionar prueba para 1 muestra")
sys.print("Pulsar el boton para correr")
sys.get_key();
run.corridastd.txt(30:00,40:00)
EV1.send(T)
EV2.send(F)
EV3.send(F)
BOMBA1.send(F)
BOMBA2.send(F)
sys.wait(00:05)
EV1.send(F)
```

The following is the script that controls the experiment, and calls "estab.txt" first to stabilize the instrument. Notice the two parameters denoted by

```
$1 and $2
```

.

```
Example 2. (corridastd.txt)
sys.new_test()
run.estab.txt()
sys.print("-- Adquisicion de Datos --")
EV1.send(T)
EV2.send(F)
EV3.send(F)
BOMBA1.send(T)
BOMBA2.send(T)
NOTHOLD.send(T)
sys.startdaq(opa.txt,FLM1)
sys.wait(00:01)
NOTHOLD.send(F)
sys.wait($1)
sys.mark()
sys.print("-- Solucion Basica --")
EV1.send(F)
EV2.send(T)
EV3.send(F)
sys.wait($2)
sys.mark()
sys.stopdaq()
sys.print("-- Solucion de Lavado --")
EV1.send(T)
EV2.send(T)
EV3.send(F)
BOMBA1.send(T)
BOMBA2.send(F)
sys.wait(10:00)
sys.mark()
sys.print("--- Fin ---")
sys.close_test()
BOMBA1.send(F)
BOMBA2.send(F)
EV1.send(T)
EV2.send(F)
EV3.send(F)

Example 3. (estab.txt)
sys.print("-- Estabilizacion --")
EV1.send(T)
EV2.send(F)
EV3.send(F)
BOMBA1.send(T)
BOMBA2.send(F)
sys.wait(10:00)
sys.mark()
```

For now, the parameters to a script are given by simple string substitution, and are referenced in the called script by

```
$1, $2, . . . $N
```

The Figure 2 shows the update sequence for the control engine, where the iterations are defined by the start and stop time. If extensions were necessary for the REACT control engine, this can be accomplished by adding new object types based on the standard object interface and loaded at run-time via the dynamic load api for Linux/Unix (dlopen(), dlsym(), dlclose(), dlerror()). This is the same method used to load modules for the Apache web server.

For non-continuous control systems, for example batch experiments and testing, we support the automatic creation of a directory for all output files using the following format:

```
<root-path>/<device-name>/<year>/<month>/<day>/
          <experiment or batch number>/
```

It is also possible to add analysis routines to process the raw output data as we did for the integration of the area under the peaks.

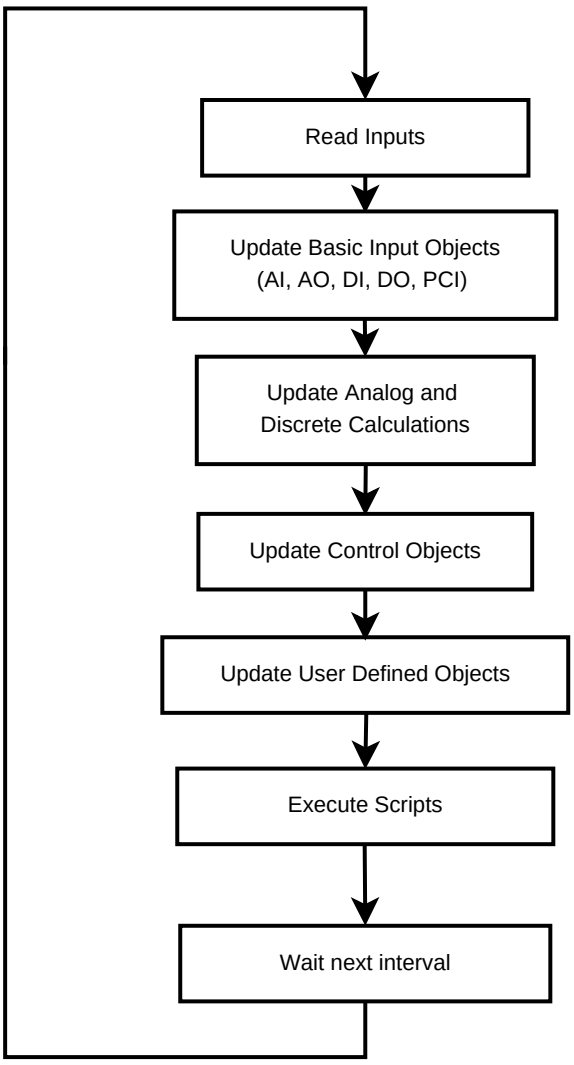


FIGURE 2: *How REACT Updates.*

4 Results

A graph relating voltage to time of the output of the chromatograph is shown in Figure 3. For the given experiment, the graphs have four peaks, that must

be identified by the scientist so that numerical analysis can be carried out on them. Figure 4 shows the graph of just one peak.

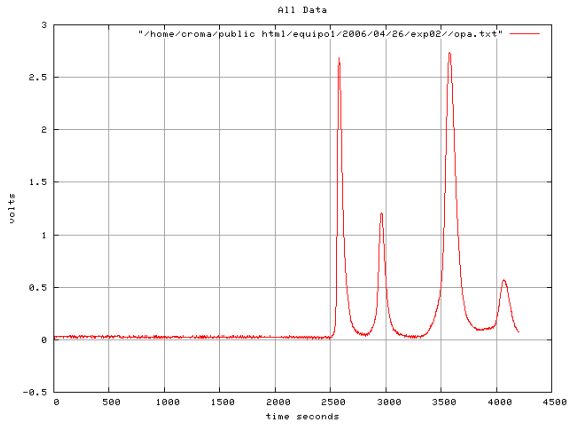


FIGURE 3: *Example Experiment.*

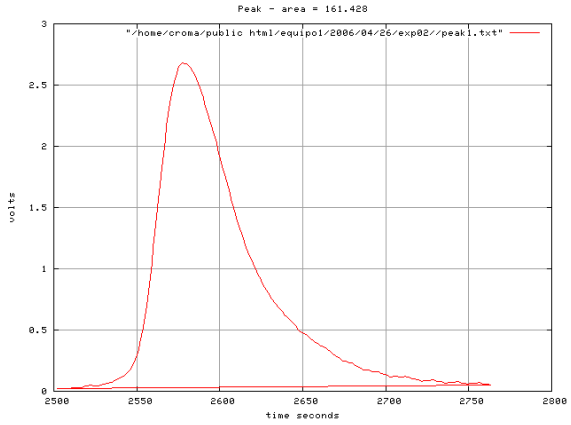


FIGURE 4: *Example Peak.*

According to the shape of the peaks, the integration is an easy task and it can be done even with simple integration algorithms with good numerical approximation of the results. In this case we simply use the trapezoidal rule.

5 Conclusion

In this work we have presented an extensible Linux based control engine with application to control a chromatograph, developed with object scripting, such that persons without advanced control and programming experience can easily create experiment controllers, and record and share data with peers and other interested parties. The control engine is also extensible by C++ programmers with control experience to add features not possible with scripting and standard object types.

References

[1] R. Reynoso-Orozco, 2004, *Las poliaminas como biomarcadores del desarrollo neoplásico y el efecto de Bursera fagaroides sobre su metabolismo.* Tesis Doctoral, Universidad de Guadalajara.

[2] M. Jeevanandam, S. R. Petersen, 2001, *Clinical role of polyamine analysis: problem and promise*, Current Opinion in Clinical Nutrition and Metabolic Care, 4 (5): pp85–396.

[3] R. A. Casero Jr., Y. Wang, T. M. Stewart, W. Devereux, A. Hacker, R. Smith, P. M. Woster, 2003, *The role of polyamine catabolism in anti-tumour drug response* Biochemical Society Transactions Vol. 31 (2), pp361–365.

[4] D. Hare1, E. Gery, 1996, *Executable Object Modeling with Statecharts*, Proceedings of the 18th International Conference on Software Engineering (ICSE '96), pp246–257

[5] D. Harel, 1987, *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, no. 8, pp. 231–274

[6] J. Karl-Heinz, M. Tiegelkamp, 2001, *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Aids to Decision-Making Tools*, Springer, ISBN: 3540677526

[7] G. Johnson, R. Jennings, 2006, *LabVIEW Graphical Programming*, McGraw-Hill, ISBN: 0071451463

An Embedded Intelligent Receiver System for Satellite-Base Distance Education

Wang Tao, Huang Kunlun, and Piao Long
Transmission Center Department,
China Education TV Station, China
wangtao@cetv.edu.cn
huangkl@cetv.edu.cn
piaolong@cetv.edu.cn

Abstract

Embedded system has developed rapidly in these years, and it declares an new era after Personal Computer. Distance education is very important to improve whole people's ability, especially in developing countries. Distance education based on Culture and Education satellite grid will be the firm groundwork in whole-people education and lifelong education system. This paper will describe an embedded intelligent receiver system for satellite-base distance education, which depends on embedded operating system and applications, Digital Video Broadcast - Satellite, Internet Protocol Data Broadcasting, Extensible Markup Language, and so on. It can receive data and manage resource long-timely, automatically and intelligently. It is also easy to be administered and operated, and suitable to be used on a large scale.

1 Introduction

Developing distance education powerfully to improve all people's ability is one of the most important strategies for developing countries. China has the largest population in the world, thus how to change the big population-pressure to human resource is one key task for Chinese government. Now, China is trying her best to set up the groundwork for whole-people education and lifelong education system Inspired by India's education satellite, China plans to implement distance education based on special education direct-broadcast satellite on a super large-scale.

With the development of information technology for education, there are two main types of receivers of satellite for distance education. One is TV, the other one is PC (Personal Computer). The first one is easy to operate but with few function. The second one has many functions, and is suitable for multimedia application but hard to operate.

Considering the people with different operating skill, there is a strong demand for such a receiver system to implement distance education based on satellite on super large-scale, as that combines the advantages of television and PC, is easy to operate, can receive data long-timely, automatically and intelligently and provide plenty study functions.

2 Embedded System and its wide application

The definition of embedded system expands and goes with its development. In this paper, we adopt the one that embedded system is a special computer that makes use of information technology, is composed of custom-build hardware, operating system and soft, and is designed for specific application.

Embedded system grows rapidly owing to the development of information technology, communication technology, and micro-electronics technology. Negroponte, the American futurologist, said when he visited China in 1999, that embedded system would be the greatest invention after PC and Internet in 4-5 years later. It is proved now. And someone even thought that embedded system would take place of PC to declare an new age after PC and change microcomputer to embed-computer.[1]

Embedded system is paying more and more attention to network connection, including Internet, Satellite, and telecommunication net. Because em-

bedded system is easy to manage, operate, and recover, it can be used by almost all over the people. It is so suitable for applications that are oriented to all kinds of people in large scale.[2]

At present, those embedded system based on Linux are relatively outstanding in this field.

3 Education Grid based on Satellite will bring huge demand for receiver system on super large-scale

IP (Internet Protocol) over DVB (Digital Video Broadcasting) data broadcasting is an unilateral transmission technology for IP data. It pushes IP data in multicast packet, encapsulates them into DVB-TS(Transport Stream) stream, multiplexes several TS streams into one, modulates TS stream into carrier, and broadcasts vie satellite or CATV(Community Antenna Television). IP data broadcasting is a new broadcasting service after audio and video, and can be used for distance education, public information notification, finance business, etc.

Satellite broadcasting grid is an advanced application of IP data broadcasting, Culture and Education Grid, which was put forward by Li Youping, an Academician of Chinese Academy of Engineering Physics, is an application based on satellite grid. This grid plans to push thousands upon thousands news papers, magazines, websites and all kinds of study resource to every family in China at low cost, in other words, "store knowledge among the folk". It will be a platform for lifelong education in China to make people study more easily. [3]

To server whole-people education and lifelong education, China Education TV Station (CETV) is implementing a Super Market of Study Project. It plans to collect all kinds of education resource all over the country, set up a intercommunion platform for supplier and demander, meet person's individual and diversified learning needs by various ways, transmit resource via China Education Broadband Satellite NET (CEBSat), provide the corresponding high quality service, intent to be the largest study platform in the world that make it easy, fast, plenty to study with low cost.[4]

Inspired by the first special education satellite in the world launched by India in Sep, 2004, China's the biggest institution of education media, CETV has put forward a plan for China's education-specific direct-broadcast satellite and begun the demonstrating work for it. Till now, CETV has finished the research on demand for Chinese distance education in 10-15 years later. [5]

Culture and Education satellite Grid, Super Market of Study project and Education-Specific Direct-Broadcast satellite will establish the firm foundation for learning-society, thus enable anyone get any-knowledge he wants at anytime anywhere, and it will also bring about a huge demand for satellite receiver system on super large-scale.

4 Design of Embedded Intelligent Receiver System for Satellite-based Distance Education

4.1 General Requirements

According to the distance education needs and embedded system features, this paper describes an Embedded Intelligent Receiver System for Satellite-base Distance Education. It is based on embedded software, DVB-S, IP data broadcasting and XML (Extensible Markup Language) technology, etc. It has several kinds of ports to connect to Satellite, cable TV or Internet. It can receive data and manage resource long-timely, automatically and intelligently, is easy to administer and operate, combines the advantages of television and PC, supports "Content look for readers" and is an intelligent agent.

4.2 Hardware Structure and key components

The hardware structure of the embedded intelligent receiver system designed in this paper is shown in figure 1.

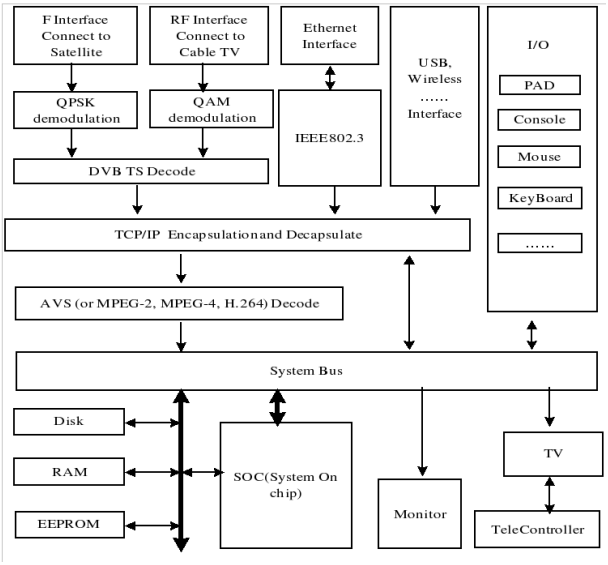


FIGURE 1: *Hardware Architecture*

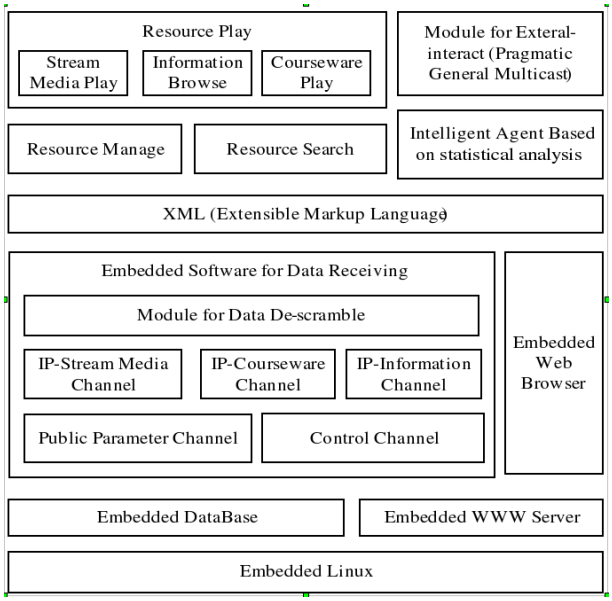


FIGURE 2: *Software Architecture*

The receiver system designed in this paper integrates solidified embedded Linux and application for receiving data base on SOCSystem on Chip.

In the satellite-based distance education, the receiver system may accept the data from satellite directly, or accept the data from the cable television network, or LAN that be forwarded from satellite, the receiver system designed in this paper integrates F interface for satellite, RF interface for Cable TV net, and Ethernet interface for Internet, combines the functions of satellite receiver, cable STB and PC, is able to demodulate QPSK(Quadrature Phase Shift Keying) and QAM(Quadrature Amplitude Modulation), decode DVB-TS.

The receiver system designed in this paper integrates AVS (Audio Video coding Standard) decoder, is able to play multimedia. AVS, which has become China's national standard, will be supported by chip manufacturers. This receiver system can also integrate MPEG-4, H.264 decoders if needed.

The receiver system designed in this paper supports two types of terminal, television and display. The I/O devices may be mouse, keyboard and telecontroller with a roller.

4.3 Software Structure and key functions [6]

The software structure of the embedded intelligent receiver system designed in this paper is shown in figure 2.

The embedded software of receiver system designed in this paper includes Embedded Linux, Embedded database, embedded Web server, data-receiving module, and drivers of adapters. Embedded Linux can be customized flexibly because of its opening source. What's more this receiver system doesn't require strict real-time.

The data-receiving module of receiver system designed in this paper has 5 data channels and 1 Descrambling component. These 5 channels are public information channel, control information channel, IP courseware channel, IP information channel, and IP streaming media channel. This module accepts data, and then retransfers data up to resource management module. This module is relatively stable, so it is embedded in the system.

The receiver system designed in this paper provides resource management and other related functions based on XML. End-users can use the standard web browser to manage resource easily. Additionally, XML, being a metadata language, it defines standard format to describe structured data that can be exchanged easily between applications, suitable for network applications.

The receiver system designed in this paper should have IP data authorizing module to receive data legally. There are two types of authorizing for this system, one is descrambling for real-time stream media, the other one is decrypting for not-real-time data.

The receiver system designed in this paper has external-interact function. Satellite broadcasting has a great advantage in unilateral transmission,

but with high cost in bidirectional communication. External-interact will be a good choice. It sends requests and all kinds of feedbacks via Internet, and gets responds via satellite. PGM (Pragmatic General Multicast) protocol is a key component in external-interact. In addition, PGM packets can be used to transmit mends of data, says, the receiver system sends information of the lost packets to broadcast center via Internet, then broadcast center retransmits the lost packets again. Usually, in the actual application of satellite broadcasting, when things go wrong, a large number of the receiving terminals in a large-scale regional will lose same part of the data. PGM can be used to improve efficiency in mending.

The receiver system designed in this paper can be restored rapidly when something is wrong, that simplifies system management to accommodate more people, and ensures to accept data in time. There are two steps for this function, the first step is the embedded soft described earlier, they can be turn to its original state if needed. The second step is to backup some databases to Flash or EEROM on every normal startup, including data receiving database and resource management database, they are used to record the data receiving process and accepted resource.

The receiver system designed in this paper will be an intelligent agent for data receiving. The front of Satellite Education Grid will broadcast huge amount of resources in parallel, then chose by end-user. In order to avoid end-user being drowned by massive information, besides the function to choose each data stream by PID(Packet Identifier), this receiver support "content looks for readers", that the end-user specifies the issues they want, the receiver system looks for resource about them, and then put for end-user. After a period of statistical analysis, the receiver system will be an intelligent agent.

The receiver system designed in this paper can be upgraded automatically. The data receiving module of the system is relatively stable, so most of upgrades are for resource management. The components that need to be updated are broadcasted in files via control information channel, and replace the corresponding ones of the terminal automatically.

5 Conclusion

This paper presents the development trends of embedded systems and the demand for satellite-based distance education under the background of whole-people education and lifelong education, and proposes a design for an embedded intelligent receiver system for satellite-based distance education. It describes the hardware and software architecture of this receive system, some key components and functions which ,to satisfy the practical needs and puts embedded system and IP data broadcasting technology into full play, tries to be suitable and practical.

References

[1] Herman Bruyninckx, *Real-Time and Embedded Guide*, uclinux howto

[2] John Lombardo, *Embedded Linux*, ISBN-10: 0-7357-0998-X

[3] Li Youping, 2003, *Culture Grid Base on Broadcast*

[4] Kang Ning, Wang Tao, etc, *Key Ideas for Developing Distance Education and Constructing Supermarket of Study*

[5] Kang Ning, Wang Tao, etc, *Demand Prediction of Distance Education for commonweal Education and Emergency Communication Satellite*

[6] Huang Kunllun, Gou Wenxiu, 2004, *A Standard and Modular IP Data Broadcasting Solution*,Audio Engineering,Issue 10

Design and Implementation of the Pre-demodulation Digital Recorder Software Based on Rtlinux

Yongbo Xin
CASC Long March Launch Vehicle Technology CO. LTD
7 Building, 15 Block, NO.188 West Road, the forth South Round, BeiJing
YongboXin@yahoo.com

Li Su
CASC Long March Launch Vehicle Technology CO. LTD
7 Building, 15 Block, NO.188 West Road, the forth South Round, BeiJing
sudanlina@yahoo.com.cn

Abstract

As a device for collecting, storing and replaying signal data, the pre-demodulation digital recorder is used in the field of aerospace widely. In this article, we clarified the software development experience of the pre-demodulation digital recorder with high bandwidth, large storage and reliability based on Rtlinux and XFS. RTlinux can respond to the hardware interrupt promptlyso data collecting is almost in real time. At the same time we select XFS to manage very large data files, and RAID to guarantee the reliability of the data stored. The RTFIFO and MBUFF in RTLinux provide communications interface for kernel procedures and user space. This is a successful case of RTLinux application.

1 Introduction

Data collection is the main goal of the recorder, meanwhile, data processing and replaying are also its tasks. The main performance indicators are the speed of recording and playing, the capacity of storage and stability.

We develop the recorder based on PC for taking full advantage of mature technology, such as memory controller, I/O interface, SCSI controller, and network controller. As for software, OS should supply multitask, desktop function and high speed for responding to system interrupts. So we select RTLinux.

RTLinux leaves the Linux kernel essentially untouched. Via a set of relatively simple modifications, it manages to convert the existing Linux kernel into a hard real-time environment. RTLinux improves the speed of recording and playing. XFS (64-bi t file system) and RAID5 guarantee the stability of large data file.

2 Hardware System Structure

Hardware structure of the recorder is shown as figure 1.The medium of data storage is several 10000RPM SCSI hard disks constituting RAID5.Hardware is based on 64-bit PCI bus, and Software is based on RTLinux and XFS.

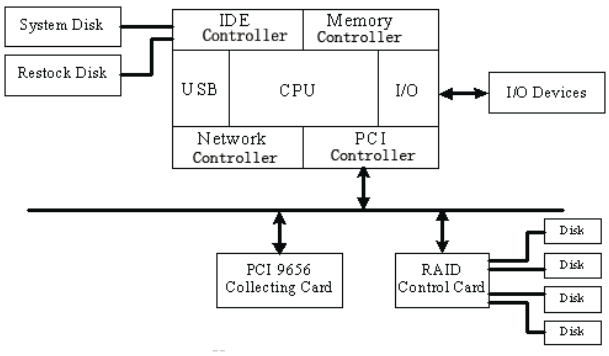


FIGURE 1: *Hardware structure of Recorder*

2.1 RTLinux

GUN/Linux[1] has become a powerful operating system challenging the OS and computer market, because it is open source and adaptable to different hardware and computing problems. As a general purpose operating system kernel, however, Linux optimizes average performance, and is not that appropriate for real time applications.
Nevertheless, there are some groups spread around the world developing modifications to the Linux kernel in order to provide a real time operating system. These Linux modifications are already used in many sophisticated real time applications, and form a good basis for embedded systems.
RTLinux manages interrupts complete. When the RTLinux kernel is idle, the standard Linux kernel runs, however, RTLinux executive is itself nonpreemptible. Figure 2 shows the detail of RTLinux kernel[2].

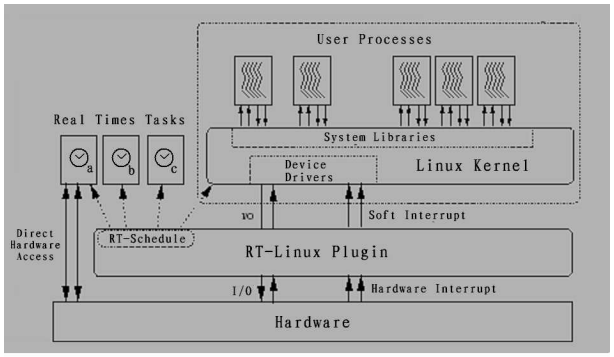


FIGURE 2: *Details of the RTLinux Kernel*

The dual-kernel approach taken by RTLinux requires a slightly different approach to real-time programming. As a kernel module, real-time code is written by RTLinux API that is managed by the real-time kernel. All user management code is run as a normal process managed by Linux,
RTLinux provides several mechanisms which allow communication between real-time threads and user space Linux processes. The most important are real-time FIFOs and shared memory. Real-time FIFOs are First-In-First-Out queues that can be read from and written to by Linux processes and RTLinux threads. FIFOs are uni-directional - you can use a pair of FIFOs for bi-directional data exchange. For shared memory, you can use the excellent mbuff driver.
We select Red Hat 8.0 with Linux-kernel-2.4.18 and RTLinux-3.1. Patch RTLinux kernel and XFS to Linux kernel, and reconfigure and compile again. Then a new real time system is come into being for our pre-demodulation Digital Recorder.

2.2 XFS

The data file recorded is often very large, even hundreds of gigabyte. It's difficult for common file system to manage, so we select XFS based on Linux Kernel 2.4.18.
XFS[3] is an advanced journaling file system originally developed by Silicon Graphics, Inc. The arrival of XFS for Linux is exciting, primarily because it provides the Linux community with a robust, refined, and very feature-rich file system that's capable of scaling to meet the toughest storage challenges. The main characteristics of XFS are as follows.
Data perfectibility: XFS ensures that any unwritten data blocks are zeroed on reboot, when XFS journal is replayed. Thus, missing blocks are filled with null bytes, eliminating the security hole.
Delayed allocation: Its a feature unique to XFS. When it comes time to write the data to disk, XFS can now allocate free space intelligently, in a way that optimizes file system performance. It is able to write the data in one fell swoop, improving write performance as well as reducing overall file system fragmentation.
In a word, XFS is the best file system to use if you need to manipulate large files.

2.3 RAID 5

RAID5[4] uses a form of striping with parity to maintain data redundancy. The parity bit shifts between the drives to increase the performance and reliability of the data. The drive array will still have increased performance over a single drive because the multiple drives can write the data faster than a single drive. The data is also fully redundant because of the parity bits. If some drive failing, the data can be rebuilt based on the data and parity bits on the remaining drives.

3 System Design

3.1 System Software Design

The software of the pre-demodulation digital recorder corresponds with the hardware to record pre-demodulation signal in real time and reply afterwards. The software Structure of the recorder is shown in figure 3.

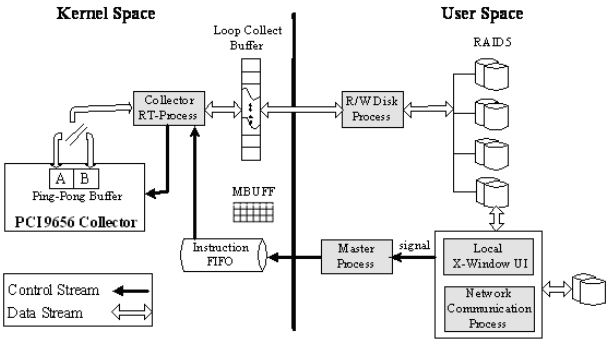


FIGURE 3: *Software structure of Recorder*

The driver module (based on RTLinux module)of data collection card is installed in kernel space, which mainly complete the configuration and controlling of the collection card and DMA transmission.
Master module is common Linux process. It is responsible for the initialization, maintenance ,transmission instructions, and communication between kernel space and user space. This module is more in wait state. When it receives the instructions by signal from the user (local X-Window UI or network Communication process) ,it transmits signal to kernel process by real-time FIFO. If the instruction is data collecting or replying, the master module will create R/W (writing or reading) disk process to record or replay data.
R/W disk process can monitor loop collect buffer in real time for storing and replying data in time. It will stop automatic when data recording and replying stop.
Local X-Window UI module provides a visual controlling platform for users. It displays the states of system, data file, and disk space, as well as ,it receives all kinds of controlling and management instructions from the user.
Network communication process runs in backend. It reports the state of the recorder and receives all kinds of controlling and management instructions from the network user just as Local X-Window UI.

3.2 The work flow

When a user sends a data recording command, the master process will be activated by a signal. The R/W disk process will be created by it, and this command is sent to the driver module through RT-FIFO. Then hardware starts collecting data.
In the process of collecting, data is stored in Ping-Pong buffer A and B in the data collection card by turns. When a buffer is full, a data ready interrupt is triggered. The corresponding ISR will move data from buffer to PCI by PCI9656 DMA controller[5].
As the data input with high-speed, the delay produced by occasional intermittent R/W or seeking track of RAID will lead to data overlay. So loop collect buffer, a larger piece of physical memory is set aside, which is out of control of the operating system. When ISR initiate DMA transmission, the data of a ping-pong buffer will be transmitted to loop collect buffer. The next start writing pointer of loop collect buffer was adjusted, so that data collected in transit without being covered. The R/W disk process checks the loop collect buffer in real time. If there are new data, it will be write to disk immediately. So large data transfer between kernel space and user space is realized. The loop collect buffer is critical to ensure data not losing, and the higher collecting speed ,the larger size of this buffer should be.
The data, however, should be read to the loop collected buffer first when replying, then this command is sent to the driver module through RT-FIFO. So the loop collected buffer ensures the continuity of replying, even if the speed of read disk slow temporary.
Communication mechanisms are necessary to interface RTLinux tasks and Linux. The user process can not access hardware directly, so RT-FIFO is used to transmit control command between two space, at the same time, for shared the key state of system and control data we use the mbuff driver.

4 Conclusions

The trait of real time in RTLinux is closely related to hardware, So we can take full advantage of PC. The testing on our recorder showed that the average of interrupt response time is 10s, and the max is 15s. This performance can satisfy this real time requirement of recorder completely and guarantee data integrity . With the high-performance and great capacity hard disk using, the capacity of the RAID would reach over 1000GB, moreover, collected data don't loss even if one hard disk is demaged.
We had done the test that under the same hard platform, the recording speed has significantly enhanced in RTLinux than in Windows NT. RTLinux is free, open source, retractile, and transferable, so many OS can not bear comparison with it. RTLinux has been applied successful in our pre-demodulation Digital Recorder,moreover, it will be more widely used in the field of real-time control .

References

[1] ,*PCI 9656BA Data Book Version 1.0*,2003, PLX Technology, Inc.

[2] ,*RTLinux3.1 Getting Started with RTLinux*,2001, FSM Labs, Inc.

[3] ,*Advanced filesystem implementers guide, introducing XFS*, Daniel Robbins,01 Jan 2002,Gentoo TechnologiesInc.

[4] ,*http://www.realtimelinuxfoundation.org/home.html*

[5] ,*http://compreviews.about.com/od/storage/l/aaRAIDPage1.htm*

Investigating the Feasibility and Designing of Telecom Billing Systems Based on Real-Time Linux

Fan Xiao-liang, Zhang Rui-sheng, Ning Ting, Du Jin, and Zhang Chun-yan
School of Information Science and Engineering
Lanzhou University, P. R. China
zhangrs@lzu.edu.cn
fanxl05@st.lzu.edu.cn

Abstract

Telecom Billing System is one of key systems among Telecom Operations Supporting Systems, which is mainly responsible for the collection of appropriate usage records, production of timely and accurate bills, and processing their payments. As sophisticated telecom services reflect on huge amount of billing related data, billing systems require high capability of real time response. In order to fulfill this real time requirement, for one thing, software providers could upgrade the priority of real time performance when designing the billing systems, for another, telecom carriers chiefly use UNIX operation systems, such as IBM AIX, on their high-performance servers to provide strong support for their real time services. Real-Time Linux, which acts as one of the best enhancements of Linux to make general Linux kernel appropriate for real time applications, is now widely considered as an alternative choice for academic research to implement efficient real time applications while minimizing the cost. Generally speaking, we investigate the feasibility of developing telecom billing systems based on Real-Time Linux. This paper is organized as follow: Firstly, we present the current market situation of telecom billing systems and what kind of challenge they are facing. Secondly, we describe generic problems for current billing systems when achieving the real-time functionality. Further more; regarding the features and application benefits of RTLinux, we discuss the feasibility of building up telecom billing systems using RTLinux. Finally, high-level architecture of the RTLinux-based billing system is briefly given.

1 Introduction

The telecommunications Operations Support Systems must address many stakeholders: R&D, Product and Requirements Management, Purchasing, Systems Integration, Systems Administration and Users [1]. While the management of next generation networks and services poses significant technical challenges, the present supply chain, market configuration, and business practices of the OSS community are an obstacle to rapid innovation. Telecom Billing System is one of key systems among Telecom Operations Supporting Systems, which is mainly in charge of collecting of appropriate usage records, production of timely and accurate bills, and processing their payments. As sophisticated telecom services reflect on huge amount of billing related data, billing systems require high capability of real-time response. RTLinux bases on General Linux OS, modifies the CPU scheduling algorithm and implements the supporting of real time, whose sustainable support to real-time functionality has the potential advantage to fulfill the requirement of Telecom Billing Systems. Our research interest concerns whether Open-source Development Mechanism as well as using RTLinux/GPL could provide an alternative mechanism for the telecoms industry to realize high capability of real-time response in an effective way in order to solve billing-related problems such as, collecting the data correctly, positioning the trouble tickets, as well as solving the real-time billing issues.

2 Telecom Billing Systems and RTLinux

2.1 Problems Billing Systems Development

2.1.1 The Market for OSS and Telecom Billing Systems

OSS Observer's study of the OSS market reveals [2] that the global spending on Operational Support Systems within the telecoms industry is around $30 billion dollars per year (about 3% of total telecoms revenue) with about 2/3 of this being spent on custom developed systems and 1/3 on COTS software. Some 200,000 IT professionals are employed globally by service provider IT departments, many of whom are working on internally developed solutions.

Telecom Billing System is one of the most important OSS solutions for telecom Operator, which is mainly responsible for dealing with complicated billing related issues. OSS Observer's study predicts that spending in the global Billing market will increase from $3.6 billion in 2005 to 5$ billion in 2010 [3], which clearly illustrate the promising market expectation.

2.1.2 What Does a Billing System Do and its Real-time Challenges

Telecommunications Billing Systems–or billing "solutions"–do a lot of pretty complicated things [4]. Sometimes a single "system" performs these functions. More often, the functions are modularized and may come from different vendors. Sometimes a system is designed to handle only specific types of billing and markets (e.g., consumer or "residence", business, wholesale). As we walk through what billing systems do, we will trace process and information flows between functions. Billing systems get a whole lot of information about a customer and the services and products the customer orders. They get some information about how and when the telecom provider delivers those services and products. This information originates in the sales and provisioning process. They get information about how and when the customer uses those services ("usage data") or if the services fail or need repair. This information originates in the network itself [4].

Within the billing environment, billing systems process large quantities of time- and location-specific usage data into customer-specific usage information. They figure out how much to charge for the usage, as well as for other non-usage-based products and services. They generate a readable bill format and print or electronically distribute the bill. Billing systems also get information on payments and apply those payments to customer accounts. They track delinquencies and flag accounts for "treatment"-collection activities including service suspension. They may generate letters to effect collection.

Major information flows and detailed operations in a telecommunications billing environment involve [4]:

1. Billing Operations Obtaining Usage Rating Calls Aggregating Usage Applying Taxes Rendering Bills Maintenance of the customer account 2. Information Management Establishing Customer Account Information Establishing Product and Service Information Establishing Business and Operational Information 3. Financial Management Managing Payment History Identifying Collectables and Treatment Reporting and Paying Taxes

The reasons why Telecom Billing System is widely considered as one of the most critical and complicated telecom Operations Support Systems are illustrated as follow: Firstly, Billing Systems features large-scale and tanglesome business processes, which result in interactive problems between different departments within the Telecom Company. Secondly, the maintenance issue of billing systems is always raised as there are frequent troubles of positioning the trouble tickets. Finally, but not the least, as 3G mobile networks and IMS services [5] are approaching, billing systems require high-level abilities responding to real-time operations. In short, currently, Telecom Billing Systems are facing tense challenges, especially on real-time aspect, to meet the on-going fast-changing demand of the market.

2.1.3 The Role of Standards

Standards have always played a significant role in Telecommunications OSS development, which provides suitable guidance when building up Operations Support Systems, including Billing Systems. However the rate of change in the industry has required a move from the paper standards development processes originally followed by the PTTs through the CCITT towards less bureaucratic mechanisms pioneered by the IETF which require working implementations before a standard is accepted [6]. Increasingly standards bodies such as the DMTF are promoting the use of open source, of course including RTLinux, to both develop and promote standards [7].The very nature of the Billing Systems market requires a commitment to real-time ability through standards. Unfortunately, this level of commitment seems not to be existing with respect to standardization of components within the OSS space.

Lately there has been a realization that the telecoms OSS market alone does not have the volume to

drive its own software standards. This is reflected in the thinking behind TeleManagement Forum's Next Generation Operations Support Systems (NGOSS) [8]. NGOSS specifies a framework for OSS integration which can bridge legacy OSS technologies with the new component development standards reaching critical mass in the enterprise world (i.e. J2EE, .NET or SOA and Web Services). Similar thinking drives OSS/J [9], which has been merged into TeleManagement Forum, an open standard initiated by SUN for implementing NGOSS compliant solutions leveraging J2EE [10]. But even NGOSS [11]and OSS/J standard, don't ever notice the possibility to drive the real-time capability by providing appropriate software solution.

It is in harnessing this last possibility, by encouraging University research institutes to work to emerging common standards, that there exists the potential for an industry wide open telecom real-time solution development initiative. Maybe RTLinux could act as one of the solutions to leverage this gap.

2.1.4 The Alternatives of Operation System

As we have discussed before, that Telecom Billing Systems require high-level real-time abilities. In the real telecommunication operations environment, operators use high-reliable Unix Operation Systems, such as IBM AIX, to secure this need. On the other hand, Linux acts as a growingly powerful Operation Systems due to its potential merits [12]: First of all, Linux can give us a modern, very stable, multi-user, multi-tasking environment on the inexpensive PC hardware, at no (or almost no) monetary cost for the software; Secondly, Linux's Standard platform includes all the UNIX standard tools and utilities. Linux is VERY standard–it is essentially a POSIX compliant UNIX. Linux is a best-of-the-breed UNIX. The word "UNIX" is not used in conjunction with Linux because "UNIX" is a registered trademark; further more, Linux's unsurpassed computing power, portability, and flexibility makes it be a Universal Operating System. A Linux cluster recently (April 1999) beat a Cray supercomputer in a standard benchmark. Linux is most popular on Intel-based PCs, but it runs very well on numerous other hardware platforms, from toy-like to mainframes. Linux can be customized to perform almost any computing task. Finally, but not the least, there are hundreds of Linux communities and discussion groups all over the world, which initiate thousands of specialized applications of Linux. In many fields, such as astronomy, information technology, chemistry, physics, engineering, linguistics, biology, etc, Linux seems like "the only" operating system in existence.

As the benefits we have listed above, it is possible for academic institutes to obtain another test infrastructure based on Linux. As a result, we are calling for other open-source Operation Systems to fulfill the real-time response requirement and, at the mean time, to lower the cost. RTLinux is an example of how Linux can be used for real-time applications by adding an executive to the Linux kernel that provides real-time services, whilst Linux is seen as a thread [13]. Thus, it can be argued that this mechanism of RTLinux's merits probably could solve the problems we have discussed before in section 2.1.2.

2.1.5 A Summary of the Problem

Telecom Billing System requires high-level real-time capabilities attribute to its characteristics and complicacies. However, the common solution to address this need proves not to be the only way to solve the problems. Thus, we prefer to find an alternative way to carry out billing-related research work within the RTLinux platform.

2.2 What is RTLinux

RT-Linux is an operating system in which a small real-time kernel coexists with the Posix-like Linux kernel. The intention is to make use of the sophisticated services and highly optimized average case behavior of a standard time-shared computer system while still permitting real-time functions to operate in a predictable and low-latency environment [13]. In RT-Linux, all interrupts are initially handled by the Real-Time kernel and are passed to the Linux task only when there are no real-time tasks to run. To minimize changes in the Linux kernel, it is provided with an emulation of the interrupt control hardware. Thus, when Linux has "disabled" interrupts, the emulation software will queue interrupts that have been passed on by the Real-Time kernel [13]. Real-time applications consist of real-time tasks that are incorporated in loadable kernel modules and Linux/UNIX processes that take care of data-logging, display, network access, and any other functions that are not constrained by worst case behavior.

In practice, the RT-Linux approach has proven to be very successful. Worst case interrupt latency on a 486/33 MHz PC measures well under 30 microseconds, close to the hardware limit. Many applications appear to benefit from a synergy between the real-time system and the average case optimized standard operating system. For example, data-acquisition applications are usually composed a simple polling or interrupt driven real-time task that pipes data through a queue to a Linux process that takes care of logging and display [13]. In such cases, the I/O

buffering and aggregation performed by Linux provides a high level of average case performance while the real-time task meets strict worst-case limited deadlines.

3 A Feasibility Study into Billing System Based on RTLinux

3.1 The Potential Benefits of Open Source Development

The software industry has woken up to the potential of open source and several commentators are proposing best practice for open source development [14]. Martin Fink the Director of open source projects at Hewlett Packard has written on the commercial opportunities of open source [15]. He sees open source as a tool to help the industry create better solutions to common problems. On a philosophical level, objections have been raised to open source on the grounds that it will kill innovation in the software industry since it is being used as a way to market "failed software products" [16]. However, this ignores the fact that the business model relies not on software sales but on services. By feeding and watering the plant, many people get to harvest the fruit.

In a 2003 UK survey, into the perception of open source software by Chief Information Officers [17], 16% of the CIOs surveyed said they were actively looking at open source solutions now. This suggests that there is a growing enterprise awareness of the potential value of open source solutions. The telecommunications community has been traditionally more sceptical but a number of service providers are becoming interested in exploring the potential of open source alternatives to internal development. A 2001 Eurescom project suggested a number of benefits could be derived from open source in the Telecoms industry [18]. A number of equipment vendors have identified similar potential benefits of open source as a means to simplify the problem of mapping vendor specific equipment functions to a more generic OSS infrastructure. This is one of the drivers behind the Ericsson, Motorola, NEC, Nokia and Siemens Co-operative Open OSS Project [19].

Academia now appears as the second major source of open source software after independent developer communities [20]. Open source could also provide a mechanism for academia to share work with commercial partners allowing more brain power to solve the OSS problem, which definitely indicates that making RTLinux be applied to Telecom Billing Systems development process is a positive alternative to address the real-time need.

3.2 RTLinux-based Billing Systems Project Objectives

The two pillars of the proposed RTLinux-based Telecom Billing Systems project are, for one thing, the exploration of open source / free software development techniques combined with the promotion of using RTLinux to fulfill real-time ablilities in the replacement of previously complicated and less-opened industry real-time solutions; for another, we view this project as a brand-new catalysts program to show that RTLinux has the potential to be suitable for building up large-scale and complicated carrier-grade telecom applications.

By this means we hope to demonstrate to the industry the value of the emerging standards and of new ways of doing business. The basic premise of RTLinux-based Telecom Billing Systems project is that the telecommunications industry needs to modify its competitive behaviour in order to co-operate in developing solutions which meet the common base needs of the industry. A mechanism is required to allow the OSS supply chain to be collapsed across the organizational boundaries between blue sky researchers, equipment vendors, OSS vendors, systems integrators and service providers, which overall solve the billing-related real-time problem in a much more open-viewed and effective way.

3.3 Research Philosophy and Strategy

Telecommunications Billing Systems, rightly understood, are both a Technical and a Business discipline. Too often research in this field has been technology led to the extent that while elegant solutions have been created, they have been of limited practical value because they were developed with an incomplete understanding of the broad systems, organization and business environment. A significant factor in the success of a billing system is an appreciation of the motivation, skills and psychology of those who are being asked to use the systems to manage the all kind of billing data or other information.

The RTLinux-based Telecom Billing Systems Project is about determining whether a particular Open-Source Approach to solution development will work in the Telecommunications industry as far as billing system is concerned. Consequently the research will draw on business and engineering analyses of the problem. However the key factor in this research is that rather than simply observe the industry, we wish to use this research to actually catalyze

a change in the way the industry does business. In order to do this we will actively be seeking significant industry partners whose participation will create an opportunity for change within the industry. Finding these partners will of course be a difficult and risky challenge. The underlying research strategy for this project will therefore be that of Action Research as discussed by Saunders [21]. Action Research is a strategy whereby the researcher is not an inert observer but actively involved in a project to bring about change. In practice this means that all of the technical research undertaken must be informed by and tested against the realities of telco industry.

4 Architecture Design of RTLinux-based Billing System

4.1 Development Schedule

The RTLinux-based Telecom Billing Systems Project was initially planned with a life of 2-year beginning in Q2 2006. Currently, we have built up a so-called "NGOSS Living Lab", which is absolutely an open-sourced infrastructure to best stimulate the telecom operators' real operations environment. In the first stage, we choose Billing Systems as the first target catalyst. In the long term, the project will lead to a simple Telecoms OSS reference architecture and test-bed. In practice in the first instance, this will be implemented using OSS/J Application Programming Interfaces, or OSS/J API, on the understanding that OSS/J already represents a technology specific implementation of NGOSS [10]. The test-bed will be available to the industry as an aid to the adoption of NGOSS and OSS/J. In addition the project will produce training collateral and worked examples to assist with the uptake of the NGOSS methodology and lifecycle [23]. It is also intended that the test-bed will be available for other research activities.

4.2 High-Level Architecture

Rather than focusing on too many interfaces and open source products, the project has decided that the primary target will be the evaluation and integration of an open source Network Management solution, OpenNMS [24], using OSS/J Trouble Ticket interface [25] with another open source Billing solution, AstBill [26], using the OSS/J Billing Mediation interface [27] on RTLinux Operation System. The outline design is illustrated in Figure 1. The project sought to use as much open source or free software as possible - targeting RTLinux/GPL as the Operation System to evaluate its real-time capability, Sun Application Server PE 8 as a free standards compliant J2EE application server, OpenNMS open source solution as network monitoring tool, AstBill open source solution as Billing functional Unit. Further more, for the trial network itself, we use Nistnet [28], as an open source network emulator.

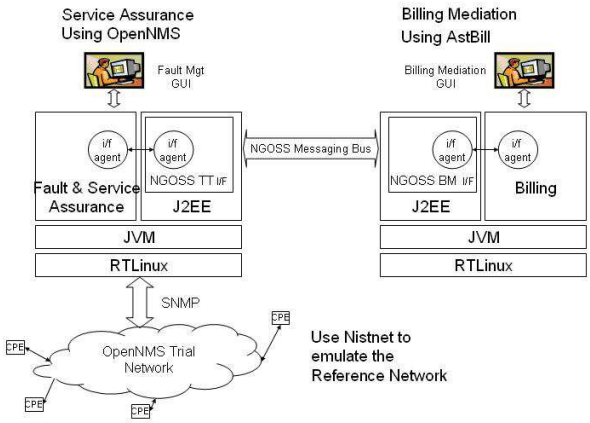


FIGURE 1: *High-level Architecture of RTLinux-based Billing System*

5 Conclusion and Future work

We have characterised how the problems within the current Telecom Billing Systems slow the innovation of next generation OSS solutions. We have described how open-source RTLinux/GPL-based Billing Solutions could realize real-time ability and make an important contribution to the rapid adoption of billing systems by academic practitioners.

The first deliverable of RTLinux-based Telecom Billing Systems will be TMF's Catalyst proof of concept, involving industrial and academic partners to be demonstrated at Telemanagement World in May 2007. The partners are currently planning the next stage of the project which will expand the community towards an open source model and finally fulfill the idea which has been expatiated in this paper. We hope our research work will, in some extent, accelerate the application development of RTLinux within the industry field in the long-run, such as Telecommunications industry as well as other Information Technology industries.

6 Acknowledgement

This work was supported by National Natural Science Foundation of China through projects: Research on Computational Chemistry E-SCIENCE and its Applications (Grant no. 90612016). And

this work is also supported by China National Technology Platform.

References

[1] C.R. Gallen, J. S. Reeve, *Investigating the Feasibility of Open Development of Operations Support Solutions*, eprints.ecs.soton.ac.uk/10378/

[2] L. Goldman 2004, *TMW Nice 2004: The Global OSS Landscape. OSS Observer*, http://www.ossobserver.com/library/library12.pdf

[3] OSS Observer 2006, *OSS Observer Report of Billing Systems*, http://www.ossobserver.com

[4] Jane M Hunter, Maud E Thiebaud & Maud Theibaud, 2003, *Telecommunications Billing Systems*,McGraw-Hill Professional, ISBN:0071408576

[5] Light Reading Report, *IMS Guide Introduction*,http://www.lightreading.com/document.asp?doc_id=70728

[6] IETF 1996, *RFC: 2026 The Internet Standards Process*,http://www.ietf.org/rfc/rfc2026.txt

[7] DMTF 2004, *DMTF WBEMsource initiative: open source enterprise management RI*,http://www.wbemsource.org/

[8] Telemanagement Forum. 2006, *NGOSS Overview*,http://www.tmforum.org/browse.aspx?catID=1912

[9] Telemanagement Forum. 2006, *OSS/J Overview*,http://www.tmforum.org/browse.aspx?catid=2896

[10] Telemanagement Forum. 2006, *OSS through Java as an Implementation of NGOSS A White Paper*,http://www.tmforum.org/browse.asp?catID=1923&linkID=29240

[11] FAN Xiao-liang & ZHANG Rui-sheng, *Introduction of eTOM Business Process Framework*,Journal of Lanzhou University (2005, Vol.41 Supp.),p268-272

[12] Juergen Haas, *Linux Benefits: For the Undecided*,http://linux.about.com/od/embedded/l/blnewbie_0toc.htm

[13] Albert Llamosi & Alfred Strohmeier, *Reliable Software Technologies*,The University of the Balearic Islands,(Ada-Europe 2004)

[14] Victor Yodaiken,Oct. 7, 1997, *An Introduction to RTLinux*,www.linuxdevice.com

[15] J. Feller & B. Fitzgerald,2002, *Understanding Open Source Software development*,Pearson Education

[16] M. Fink 2002, *the business and economics of Linux and Open Source*,Prentice Hall PTR, London

[17] K. M. Schmidt & M. Schnitzer 2003, *Public subsidies for open source*,London: Centre for Economic Policy Research

[18] Open Forum Europe. 2003, *Market Perception Analysis of Open Source Software*,http://www.openforumeurope.org

[19] A. Gavras. 2001, *Eurescom P1044 Report: A strategic study on the use of Open Source in a telecommunications operator's environment*,http://www.eurescom.de/public/projects/P1000-series/p1044/default.asp

[20] Nokia (11-10-2004), *Press Release: CO-OP Co-operative Open OSS Project*,http://press.nokia.com/PR/200410/963794_5.html

[21] J. M. Dalle & G. Rousseau. 2004, *Towards an Open Source Technology Transfer. Proc.4th Workshop on Open Source SW Eng 2004*,http://opensource.ucc.ie/icse2004/index.html,IEEE Computer Society

[22] M. Saunders, P. Lewis & A. Thornhill, 2003, *Research Methods for Business Students (3rd Edition)*, Prentice Hall, and Harlow

[23] FAN Xiao-liang & ZHANG Rui-sheng, *Brief Introduction of NGOSS lifecycle and Methodology*, Journal of Lanzhou University (2005, Vol.41 Supp.),p264-267

[24] *OpenNMS Project (2006)*, www.openNMS.org

[25] TMF OSS/J Technical program, *OSS/J Trouble Ticket API*, http://jcp.org/en/jsr/detail?id=91

[26] *AstBill Project .2006*,http://sourceforge.net/projects/astbill

[27] TMF OSS/J Technical program, *OSS/J Billing Mediation API*, http://jcp.org/en/jsr/detail?id=130

[28] *Nistnet Project*, http://www-x.antd.nist.gov/nistnet

Multilevel Tracing for Real-Time Application Interface (RTAI) Based Systems

A. Viana, O.R. Polo, P. Parra, M. Knoblauch, F. Alcojor and S.S Prieto
University of Alcala
Ctra. Madrid-Barcelona Km 33.600 - 28831 - Madrid
{avs, opolo, parra, martin, falcojor, ssp}@aut.uah.es

Abstract

Real-time systems development is a complex process. Due to this fact, the ability to trace the different system events becomes essential in order to verify the correct system behavior and implementation. The POSIX 1003.1q tracing standard provides an interface to handle event data. However, it does not fit well with multilevel software tracing, in which it is necessary to extract events information from different levels, and route it to different tracing tools that are suitable for representing the semantic of the levels.

This paper presents an implementation of a multilevel tracing mechanism over the Real Time Application Interface (RTAI). The goal of this work is to facilitate the validation and verification process of RTAI-based real-time software systems developed with a component based graphical modeling CASE tool, named EDROOM. The EDROOM services library provides tracing information letting the designer to analyze the behavior of the components during the system execution, with the advantage of using a tracing tool with the same graphical state-charts notation defined by the EDROOM formalism.

The solution also supports the tracing of the application events, which will belong to different upper levels, that are specific of each software system. The code generated by EDROOM is adapted to be executed as a kernel module over RTAI, and all the trace information is sent through a FIFO using the proper RTAI interface. A Linux user task is in charge of receiving the information, redistributing it to different pipes or files that feed different trace tools assigned to each level. An example of use of the whole multilevel tracing mechanism is also showed.

1 Introduction

Real-Time system development is a complex process that involves a lot of activities that must be carried out. One of these activities is centered on the ability to trace system events in order to verify the correct behavior of the developed system. Embedded systems can be divided into several parts or several layers or levels. In many cases those levels should be traced independently, which means to have a multilevel tracing system.

POSIX has a standard interface to provide some trace facilities. However, it is difficult to trace a multilevel system with this standard because it is necessary to extract some information about the level of each event, also routing to different tracing tools for further representation.

The work presented in this paper shows an implementation of a multilevel tracing system over the Real Time Application Interface (RTAI) [5]. The main goal of this work is to ease the validation an verification processes of any RTAI-based real-time software system developed with a component based graphical modeling CASE tool named EDROOM [1]. The EDROOM service library (ESL) generates some tracing information letting the developers analyze this information at execution time. The EDROOM tool also provides a graphical tracing tool using the same state-chart notation defined in the ROOM formalism.

The paper is organized as follows: section 2 makes a brief explanation about the EDROOM tool. Section 3 presents the POSIX 1003.q standard. A brief introduce about the RTAI system is shown in section 4. Section 5 explains the POSIX 1003.q implementation we made and the modifications over the standard to support the multilevel tracing. The multilevel tracing system implementation over RTAI is presented in the section 6. Section 7 shows a real multilevel tracing example. Finally, section 8 shows

the conclusions.

2 EDROOM Overview

EDROOM is a tool inspired on ROOM [2] and UML2 [3] methodologies. This tool provides facilities for modeling real-time systems using the object oriented paradigm. It integrates an automatic Embedded C++ code generator. EDROOM lets the designer describe the structure, communication and behavior of real-time systems using diagrams. Both the structure and behavior designs can be hierarchized in order to ease the system design. The components of the hierarchical structure are connected between them by message passing through their ports.

In figure 1 an EDROOM model example is shown.

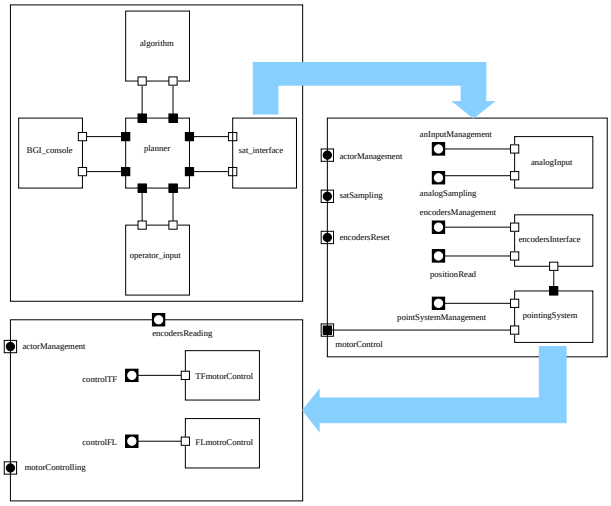


FIGURE 1: *Multilevel structure of a EDROOM model*

The behavior of each component is defined using a kind of hierarchical state chart, called ROOMCharts, based on the Statecharts introduced by Harel [4]. Received messages lead the trigger of the transitions between states. Figure 2 shows a behavior example of a component.

Both, port communication and hierarchical ROOMCharts have been adopted as new UML2 modeling elements, making the EDROOM tool also compliant with the new UML2 component based design.

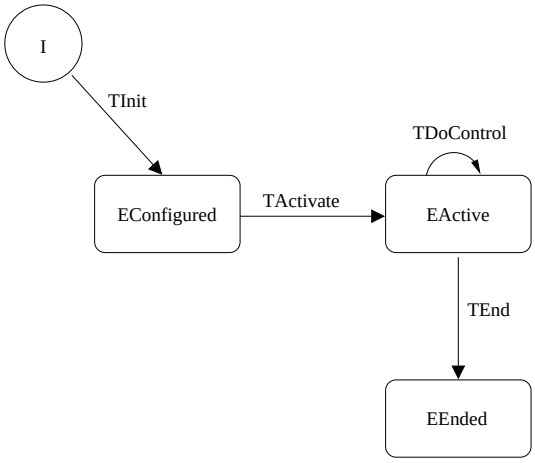


FIGURE 2: *State chart behavior of an EDROOM component.*

EDROOM defines the set of scheduling, communication, timing, interrupt and memory services provided by the EDROOM Services Library (ESL). The ESL must satisfy the requirements of the real-time software systems designed with EDROOM. The ESL is defined in two levels. The upper level is operating system independent and implements most of the code of the EDROOM services. The lower one, called Basic Primitives Layer (BPL), encapsulates the RTOS and must be re-implemented for each new ESL port. The BPL only needs support for threads, semaphores, timing and interrupt handling. Therefore it is just in the frame of the Minimal Real Time System POSIX profile PSE51. This architecture is shown in figure 3.

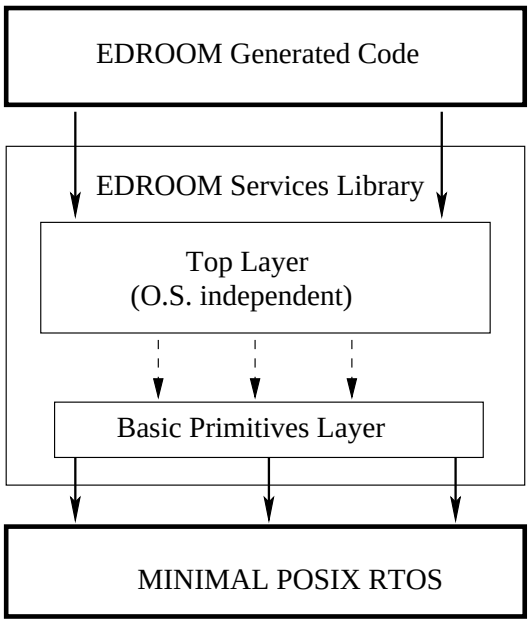


FIGURE 3: *EDROOM service architecture.*

3 POSIX 1003.q Overview

The POSIX Tracing Standard has been developed to provide tracing facilities to systems. It defines two main data types, called *events* and *trace streams*. The former includes events, occurred on the system, that must be traced. The latter is the buffer stream where the information of each event will be stored in order to be eventually analyzed.

3.1 Trace Events

When a specific application or program needs to be traced, all the required traceable events are defined. In POSIX Trace terminology, the points where the information must be generated are called *trace points*, and the information itself is called *trace events*. Each event belongs to a certain *event type* and is also associated with an *event name*. When an instrumented application wants to register a new event, it must invoke the `posix_trace_eventid_open()` routine which returns the event identifier. If the event had been previously defined, the function call returns its former event identifier. The event trace mechanism is performed by calling the `posix_trace_event()` routine. The standard specifies that the information that must be saved in every trace is the following:

- The trace event type identifier
- A time-stamp
- The process identifier of the traced process
- The thread identifier of the traced process, if the operating system supports threads
- The program address where the trace is being performed
- Any extra data associated with the event and previously defined by the user
- The extra data size

3.2 Stream Buffers

When any system application traces an event, its information is stored in the stream buffer. The POSIX Tracing Standard specifies that streams must be created by processes and that the relationship between streams and processes is many-to-many. By default, all events associated with a process are traced in all stream buffers belonging to that process. Thus, it is possible to trace events from a single process into many streams. The POSIX standard supports also event filtering. This means that it is possible to filter some specified events in order not to store them in the stream buffer. By doing this, events corresponding to one thread can be associated with a single stream. It also allows tracing the events of various processes into one single stream. This situation occurs when a process creates some buffer streams before creating the rest of the processes by using the corresponding fork system calls. In the case of one single process compounded by many threads, all events of the different threads are traced into all the streams belonging to the process.

The standard defines two types of streams: active streams and pre-recorded streams. An active stream is created to trace events during system execution. It can also be associated with a log file in order to store the information on a persistent object when a flush operation is performed. A pre-recorded stream is designed to retrieve events that have been previously recorded in a log file. They are frequently used to carry out off-line analysis of the tracing activity.

3.3 Tracing Roles

The POSIX Tracing Standard defines three types of roles called *trace processes*: the trace controller process, the traced process and the analyzer process.

Controller process. The controller process is in charge of the stream buffer creation and, in most cases, of the trace system start up. It must carry out the following operations: (1) creating the trace stream with its attributes; (2) starting and stopping the trace system; (3) filtering the events that are being traced in the corresponding streams; and (4) shutting down the stream.

Traced process. The traced process is the one that is being traced. The standard defines only two operations that must be carried out by this kind of process: (1) registering a new user event by calling the `posix_trace_eventid_open()` routine; and (2) tracing the appropriate event by calling the `posix_trace_event()` routine.

Analyzer process. The analyzer process is in charge of retrieving the traced events from the stream buffer in order to analyze the system behavior. This process can perform on-line or off-line analysis, depending on the type of the stream.

The standard also defines different layers in the implementation which can be fulfilled or not, depending on the particular system trace functionality.

4 RTAI Overview

The RTAI project began at the Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano (DIAPM) in 1996/97. It stemmed from the need of making available a tool to support a varied set of internal research activities related to advanced active controls for generic aeroservoelastic systems, including large space structures, acoustics and flexible manipulators. Its aim was to make possible their development, implementation and testing on standard 32 bits personal computers (PC) and data acquisition cards, by using high level language programming tools, so that anybody, including graduating students, could proceed to their implementation with a relative ease in building it all.

Budget constraints and the satisfaction of some DIAPM researchers in using Linux as a general purpose operating system brought the idea of adding hard real time capabilities to it. Further details on the early story of RTAI can be found in the documentation available at the RTAI home site [5] and need not to be repeated in this paper.

RTAI does not implement the POSIX Tracing Standard so we implemented it over the RTAI API with few modifications we made to provide the multilevel tracing facility but being fully compliant with the standard. In the next sections we explain some details about this implementation.

5 POSIX Multilevel Tracing Implementation

The aim was to implement a multilevel tracing system that let us tracing all the levels involved in any application. We use the EDROOM tool to model and automatically generate the application code based on the ROOM methodology and the component based paradigm. The EDROOM components reactive behavior is message passing based - each message triggers a transition between the states - so it is quite important to trace these messages in order to verify the correctness of the application functionality. Every component of the system will have the ability to register and trace all the events it considers necessary. Each event has a level identifier associated with it. This allows EDROOM applications to generate trace information at different levels or layers in order to process it later in a different way depending on the levels the information belongs.

The current POSIX standard does not fit well with the multilevel tracing approach because it does not keep the inter-level encapsulation, taking care only about the events but not about the levels. That fact breaks with the component based paradigm. We have implemented some modifications in the standard in order to perform the multilevel tracing while maintaining the compliance with the standard.

To identify a certain event belonging to a certain level, two identifiers must be known: the level and the event identifiers. POSIX routines do not take care about the level field, only the event identifier is defined as an entity. We made some modifications in those routines to consider the level identifier as well. The POSIX standard defines the event identifier as 32-bit integer number. We use the 24 higher bits for the event identifier and the 8 lower bits as the level identifier. In that way, the user passes the identifier as a parameter to the POSIX routines to register a new event with the lower 8 bits (the level) filled, and the routine fills the event bits. The routine does not suffer any modification in its definition, only differs in the implementation. Moreover, it is possible to execute applications developed over the traditional standard without make any modifications and the applications will run correctly, that means that the modifications proposed over the POSIX 1003.q maintain the backward compatibility. The only drawback of the proposal is the less number of events we can trace due the loose of the lower eight bits for the level identifiers.

6 Multilevel Tracing Over RTAI

In this section we describe the implementation of the tracing system, with multilevel trace capabilities, for the RTAI Linux Platform over the modified POSIX 1003.q explained in the previous section. As we said, EDROOM applications are compiled and executed as RTAI kernel modules. The resulting kernel module is complemented with a user level application that has several functions such as starting and stopping the module execution, as well as dealing with the information generated by the EDROOM application.

The kernel module exchanges information with the aforementioned user process by using RTAI special FIFOs. In our implementation we use two different FIFOs, the *control* FIFO and the *trace stream* FIFO.

Control FIFO. Used, as its name suggests, to control the whole execution of the kernel module, i.e. to send signals that mark the start and the end of the execution to the EDROOM tasks.

Trace stream FIFO. Through this FIFO, the EDROOM application sends all the trace information of the different levels to the user process.

The system is, thus, as shown in Figure 4.

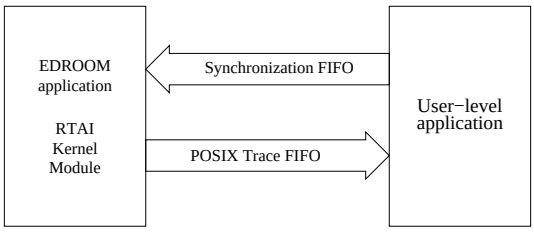


FIGURE 4: *RTAI FIFOs global scheme.*

When the module is loaded, the loader routine configures the kernel side. Setting up the FIFOs, initializing the EDROOM objects and tasks and blocking the application until the signal from the user control process is received. When the control process is running, initializes the FIFOs for the user side, sends the starting signal to the kernel module, and remains awaiting the trace information. When the applications are finished, the control process sends the corresponding signal to the module, which blocks all the pending tasks and closes the kernel side of the FIFOs, leaving the module ready to be unloaded.

The module loader routine and the user process act as the controller process, in the terminology defined by the POSIX Tracing Standard. The user process also acts as a *trace router*. It separates the raw information received into the different defined levels and re-routes them to other applications that would perform the analysis on it. This operation can be done online, through serial ports, or off-line, by saving it into different files depending on the level. The whole system follows the scheme shown in Figure 5.

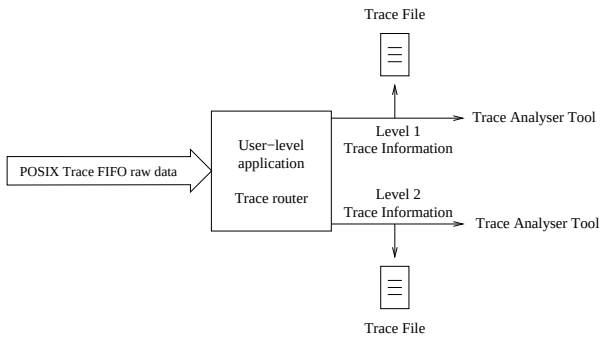


FIGURE 5: *Multilevel Routing software*

7 Multilevel Tracing Example

The described platform is already being used to develop and test the on board software of the Nanosat 1B nanosatellite of the Spanish Aerospatial Technology National Institute (INTA). Since the code generated by the EDROOM tool is platform independent, the model can be tested in the flexible framework that we are presenting in this paper. To ease the development and testing of the software, we have defined two levels of trace information: the EDROOM events level and the higher EGSE (Electrical Ground Support Equipment) trace level.

7.1 EDROOM Events Trace

The information related to this level consists in the time-tagged transitions between the states that shape the component behavior. This trace level allows to verify the system design and also the time requirements during system execution.

The trace information feeds a trace tool, called EDROOM Tracer, allowing online and off-line representation of the system behavior by showing all the transitions occurring during the application execution. An example of the EDROOM Tracer is shown in Figure 6.

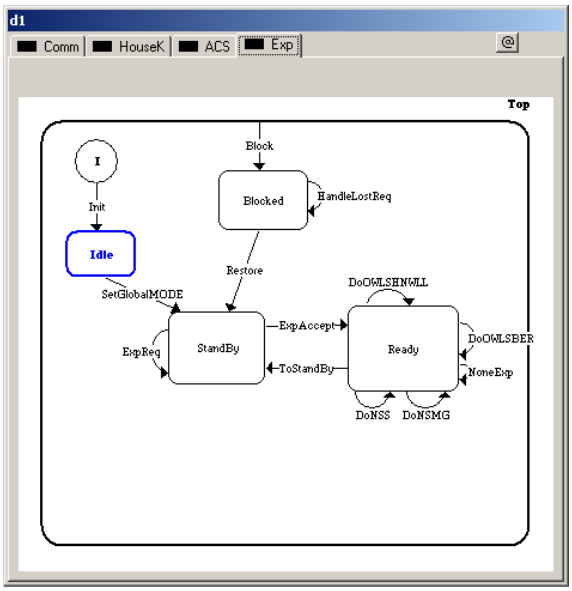


FIGURE 6: *EDROOM Events Tracer.*

7.2 Nanosat 1B EGSE Trace

This trace level corresponds to the events associated with the on-board software behavior. The information regarding these events is sent to the EGSE to carry out the monitoring of all the software functionality such as the experiments activation, communication, housekeeping etc. Figure 7 shows a snapshot of the Nanosat 1B EGSE application. Basing on the information received, the EGSE application maintains a detailed reconstruction of the on-board state.

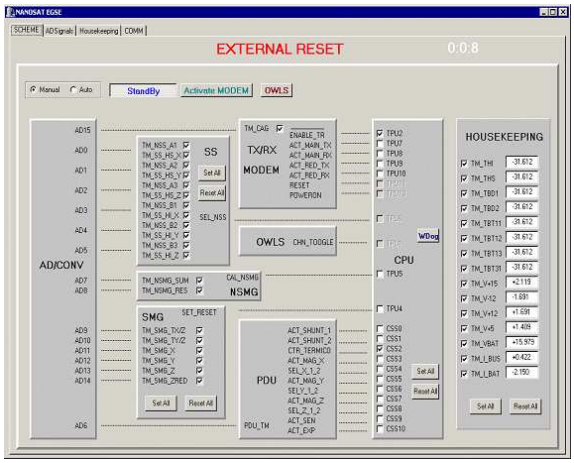


FIGURE 7: *Nanosat 1B EGSE Level Tracing.*

8 Conclusions

Nowadays, the complexity of real-time embedded systems is increasing. Due to this fact, tracing features are needed in order to ensure the correct system design and behavior. Moreover, these tracing features must not only be centered in a specific part of the system, but also must provide a full system coverage in terms of tracing, making possible the tracing at all levels such as operating system layer, application layer, end user layer and so on.

This paper has presented a multilevel tracing system based on POSIX 1003.q standard and its real implementation over the RTAI real-time kernel. To support the multilevel tracing, the POSIX 1003.q implementation made in quite different from the original one, but the backward compatibility is strictly observed.

The tracing system allows the user to trace all the embedded applications layers developed with the EDROOM modeling tool. Each layer has associated specific tracing tools, which receive and perform the online or off-line analysis of the trace information sent from the embedded system. A real example has been provided in this paper showing the utility of this approach.

References

[1] Oscar R. Polo and De la Cruz J. M. and Giron-Sierra J.M. and Esteban S., *EDROOM. Automatic C++ Code Generator for Real-Time Systems Modelled with ROOM*, NTCC2001 IFAC Conference, 2001.

[2] Selic and B. and Gulleckson and G. and Ward P.T, *Real-Time Object Oriented Modelling*, John Wiley and Sons, 1994.

[3] UML 2.0.
http://www.u2-partners.org/uml2wg.htm

[4] D. Harel, *Statecharts : A visual approach to complex systems*, Science of Computer Programming, 1987.

[5] RTAI
http://www.aero.polimi.it/rtai

Numerical Model Compiler: A Design for Generating Real-Time Numerical Simulation Code

Ivan Raikov and Robert Butera
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
{iraikov,rbutera}@ece.gatech.edu

Abstract

This paper presents the design of a domain-specific language processor that takes in a collection of first-order algebraic, differential or difference equations as input, and produces an executable that performs a real-time numerical simulation of the model described by these equations.

Domain-specific languages are a general software engineering approach that in this case allow for easy encoding of a wide variety of target APIs, and for clean encapsulation and separation of the user model from the surrounding operating system and simulation environment.

A key feature of the design is the intermediate language, which is a variant of lambda calulus augmented with semantics for memory allocation that allow the compiler to generate a static representation of the heap used by the model, and guarantee that all memory operations throughout the simulation are constant-time. The design outlined here uses type-preserving transformations that guarantee that given a syntactically correct input model, the generated output code is type correct, and by extension memory safe.

The paper concludes by showing an example of extending the user input language with support for asynchronous events, and using the example as way of anecdotal evidence for the extensibility and flexibility of the system.

1 Introduction

The Numerical Model Compiler (NMC) is a domain-specific language processor aimed at generating loadable modules for real-time simulation environments such as RTLab [3] or MRCI [11], which can simulate dynamical systems in a control loop where the state variables and outputs are dependent on both the internal model and input from experimental measurements. These simulation environments are specifically used for execution of the dynamic clamp protocol, and run under Real-Time Linux or RTAI.

The dynamic clamp is a common technique in cellular electrophysiology that uses a computer to inject simulated current driven by a measured voltage into electrically excitable cells, such as neurons and cardiac myocytes [12]. It was first introduced by Andrew Sharp et al [14], and enjoys use in a variety of applications, including endowing individual neurons with artificial conductances, creating artificial networks of real neurons [15], and coupling computational cell models to in vitro cells [7].

Our experiences with real-time dynamic clamp systems have led us to believe that 1) the domain-specific language approach can be applied to a wide variety of disciplines that involve real-time numerical simulation and control and 2) the increasing complexity and diversity of models require a flexible design that can support the evolution of modeling languages and simulation environments.

Therefore, the first goal of NMC is to support a broad family of equation-oriented languages for modeling physical or abstract processes. The architecture of NMC includes a processing stage for an abstract equation language that strives to provide flexible, general interface for multiple user-input languages that might be developed for the system.

The second goal of NMC is to support a variety of real-time execution environments, apart from the simulation systems specifically used in electrophysiology. NMC features a flexible intermediate representation and an interface for multiple backends that can support generation of different output languages and APIs. This intermediate representation

is based on Greg Morrisett's Typed Assembly Language [9], which is an instance of George Necula and Peter Lee's proof carrying code (PCC) [10]. PCC allows properties such as memory safety to be automatically verified by the compiler.

The use of typed assembly language allows NMC to generate code that is guaranteed to access resources properly, and to always call external routines with the correct types of arguments. Such verified code is suitable for loading as a Linux kernel module, or in any other context where type and memory safety are crucial. As Morrisett points out, this is similar in spirit to the SPIN project [1], where the operating system allowed the user to load verified extensions into the kernel; however, SPIN required that all extensions be written in the high-level language Modula-3, and that a special trusted compiler for that language be used, along with cryptographic signatures, in order to guarantee the safety of each extension. Morrisett argues that by comparison, a system based on a typed assembly language could support extensions written in a variety of high-level languages using a variety of compilers, as the properties of the resulting assembly code can be checked independently of the source code or the compiler.

Another consequence of using typed assembly language is that NMC can construct a table that reflects the types of all heap variables at compile time. We have found that when recursive functions are removed from the intermediate language, in effect making the language a finite state machine, the compiler can create a static heap image of the program, and eliminate the need for dynamic memory allocation.

2 Experimental Setup and Simulator Architecture

A typical experimental setup for a dynamic clamp system is illustrated in Figure 1. It consists of a neuron model being simulated in real-time, a data acquisition (DAQ) board, a signal amplifier, and one or more live neurons.

The real-time simulation engine runs the model in a loop, which consists of reading the analog inputs to the system via the analog-to-digital converters in the DAQ board, numerically evaluating the equations of the model, and outputting the state variables of the model to the digital-to-analog converters of the DAQ board.

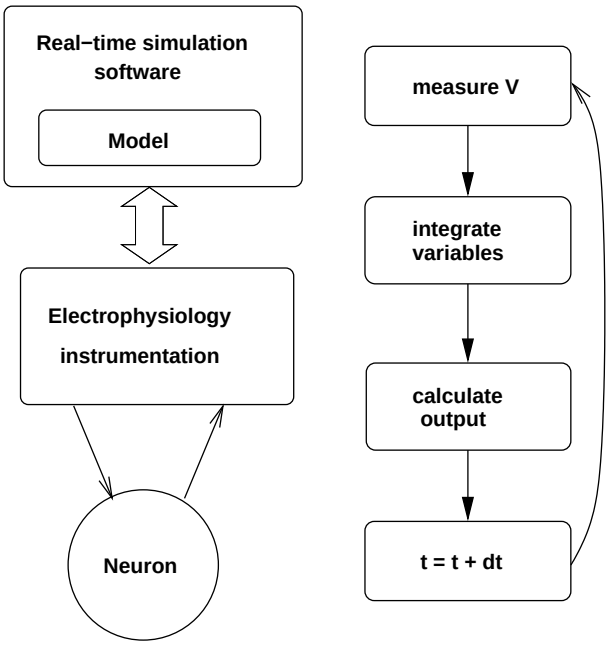


FIGURE 1: *Real-time model-experiment interactions.*

A summary of control-loop simulator architecture is presented in Figure 2. The model is typically executable code in a dynamically loadable module, and the interface used by the simulator to control the model consists of an initialization function, deinitialization function, state computation function, and an event handler. In the initialization section, all states of the model are set to their initial values and function interpolation tables are computed. The code in the initialization section is executed by the simulator upon loading of the model, prior to starting the actual computational loop.

The simulator takes care of updating the external states defined in the model. An external state is a model state that is used to interact with the data acquisition equipment, i.e. either acquiring or outputting data. Prior to each computational cycle, the simulator reads samples from the data acquisition board, and updates the external input states with the acquired data. At the end of each computational cycle, the simulator outputs the values of all external output states to the data acquisition board.

The loop is executed synchronously at periodic intervals, typically 100 μs or faster. In order to guarantee the loop frequency, both RTLab and MRCI manage the simulation process with the real-time scheduling facilities of Real-Time Linux or RTAI.

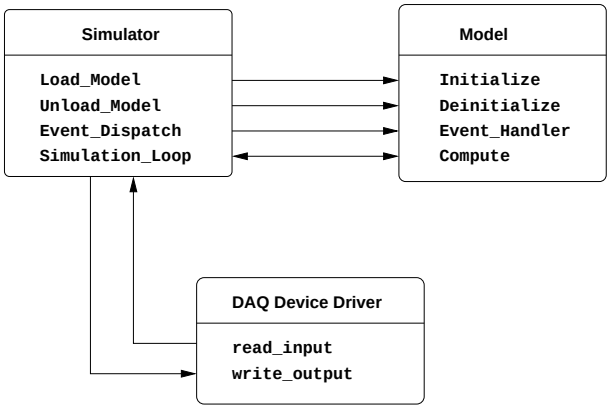


FIGURE 2: *Generalized control loop simulator design.*

The role of NMC is this context is to generate an executable model that conforms to the simulator API, given a high-level model description as input. The model description language of NMC is a domain-specific language that uses differential and algebraic equations as basic building blocks for constructing a model. The domain-specific language approach has the advantage that a model can be expressed at the level of abstraction of the problem domain. Consequently, researchers who wish to study a particular biological or physical system, can write models without the overhead of general-purpose programming languages. As Olin Shivers points out [16], the sacrifice of generality is compensated by a gain in notational compactness and clarity,

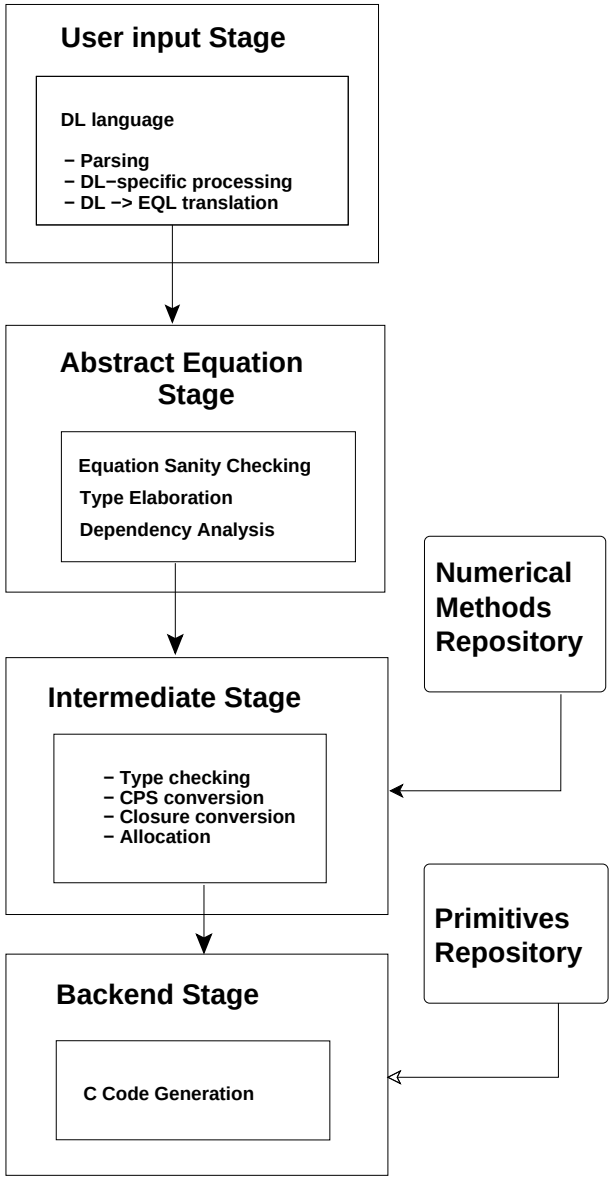


FIGURE 3: *High-level design diagram of NMC.*

3 Language Design and Translation Process

Many domain-specific languages are designed and implemented from scratch, and often include elements that are common to general-purpose programming languages, such as procedures, variables and variable scopes, datatypes and data structures. These features are designed around the domain-specific features of the language, they are often not sufficiently general, and necessitate considerable changes throughout the entire design whenever there is a need to alter the surface-level syntax and semantics.

An alternative approach is to embed the domain-specific language within a general-purpose programming language. Olin Shivers demonstrated how to embed several "little languages" in Scheme as a base language, and suggested that this approach would allow the programmer to exploit the vast wealth of design and implementation efforts put into the Scheme language and tools [16].

However, implementing features such as static memory allocation, which helps with meeting real-time execution constraints, can be relatively difficult within the framework of a general-purpose language; for example, the run-time assumptions of an existing Scheme compiler might present considerable obstacles when targeting real-time systems.

3.1 The Four Language Stages

NMC uses a four-stage architecture that combines domain-specific frontends with an intermediate representation that is a general-purpose programming language. The four stages are: a user input frontend, abstract equation language stage, which describes the pure mathematical system; intermediate language stage, which in the present design is a variant of typed lambda calculus augmented with semantics for static memory management, floating-point arithmetic, and various arithmetic functions; and a back-end, which can currently generate low-level C code.

In the first two stages, various mathematical and/or model-related transformations and optimizations can be performed on the input equations, which are then passed on to the intermediate language stage.

The intermediate language stage is where the equation language is translated into a more general representation. The intermediate language is a call-by-value variant of System F (polymorphic second-order lambda calculus), first proposed by Jean-Yves Girard [4, via [9]].

The fourth stage currently generates C code for a particular simulator API that is then compiled with a C compiler.

3.2 Translation Process

Each translation stage reads as input an XML representation of the model, converts that representation to an internal form that is more suitable for semantic processing, performs analysis and transformation operations and produces XML representation of the resulting model description. For the XML-specific parts of the translation process, NMC uses the SSAX/SXML framework, which represents XML data as S-expressions, and provides SXSLT [6], a declarative XML transformation language analogous to XSLT. The transformations between the distinct translation phases are implemented as SXSLT transformation "stylesheets" that recursively descend the SXML document, and invoke the appropriate function for each node type that is encountered.

To transform a model from its abstract equation form to intermediate representation, the corresponding SXSLT stylesheet uses another SXML document, which we call the numerical methods repository. The methods repository contains intermediate language implementations of the different numerical methods that are used to compute the equations of the model. The transformation stylesheet "weaves" together the equations from the model and the numerical method code to be used for the evaluation of each equation, and produces an executable implementation of the model in the intermediate language. This design allows for clean separation of the details of numerical methods from the language design, as well as extending the system with new methods and new types of equations.

Furthermore, it is conceivable that the domain-specific stages might need to generate additional context information that is used by a backend. SXML makes such design conceptually simple, as it is sufficient to insert the new context information to the SXML document that is passed between the different stages.

4 Intermediate Representation

The intermediate representation consists of several closely-related functional languages for computing with numeric, procedural, and compound values.

The source intermediate language, which is Girard's λ_F in the present design, is translated into several transitory intermediate languages, following the development in *From System F to Typed Assembly Language* by Greg Morrisett, et al. [9]. All transformations closely follow Morrisett's paper, except that in NMC the language has been extended with immutable record and vector datatypes, and it does not permit recursive functions. The syntax for the source intermediate language is given in table 1.

We interpret λ_F with standard call-by-value semantics, and typecheck using type judgments described in e.g. [2]. The datatypes supported are integers, reals, vectors and records, and functions. The basic operations are function creation, function application, type application for polymorphic types, conditional statement, record and vector creation and access.

Each transformation in the intermediate language stage is followed by a type check. As Morrisett points out, this provides a convenient tool for testing new transformations, and any optimizations implemented between transformation stages. In the subsequent stages, the type system provides a convenient vehicle for verifying memory safety properties of the program that represents the model being translated.

$$
\begin{array}{lll}
\textit{type} & \tau ::= & \alpha \mid \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{real} \mid \\
 & & \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \mathit{vector}(\tau) \\
 & & \mid \{\sigma_1 : \tau_1 \ldots \sigma_n : \tau_n\} \\
 & & \\
\textit{term} & e ::= & x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \\
 & & \mathsf{if}(e_1, e_2, e_3) \mid \{\sigma_1 = e_1 \ldots \sigma_n = e_n\} \mid \\
 & & \langle e_1 \ldots e_n \rangle \mid \mathrm{sel}(e, \sigma) \mid \mathsf{ref}(e, \mathit{int})
\end{array}
$$

TABLE 1: *Syntax of* λ_F

The intermediate language has atomic constructors for creating records and vectors, but in C or assembly language, space must be allocated and the structure must be filled out field by field. λ^A (allocation calculus), which is the final form of the intermediate representation, makes the processes of allocation and initialization explicit. The creation of an n-element record is separated into an allocation step and n initialization steps. Furthermore, the terms in λ^A programs are separated in declarations and statements. All memory allocation and initialization operations are declarations, which are then followed by statements.

initflag	$\varphi ::=$	$0 \mid 1$
type	$\tau ::=$	$\alpha \mid \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{real} \mid$ $\forall\vec{\alpha}.(\vec{\tau_1}) \rightarrow \mathit{unit} \mid \exists\alpha.\tau \mid$ $\mathit{vector}(\tau, \vec{\varphi}) \mid$ $\{\sigma_1 : \tau_1^{\varphi_1} \ldots \sigma_n : \tau_n^{\varphi_n}\}$
term	$e ::=$	$\mathsf{let}\ \vec{d}\ \mathsf{in}\ e \mid v(\vec{v}) \mid \mathsf{if}(v, e_1, e_2) \mid$ $\mathsf{return}[\tau]v$
decl	$d ::=$	$x = v \mid \mathsf{malloc}(\ell, \tau)$ $\mathsf{memcpy}(\ell, x, v_{dest}, v_{src}) \mid$
value	$v ::=$	$x \mid i \in \mathbf{Z} \mid r \in \mathbf{R} \mid v[\tau]$
block	$b ::=$	$\ell \Rightarrow [\vec{\alpha}](x_1 : \tau_1 \ldots \vec{\alpha}](x_n : \tau_n).e$
heapval	$\chi ::=$	$b \mid \ell \Rightarrow \mathit{vector}(\tau) \mid$ $\ell \Rightarrow \{\sigma_1 : \tau_1 \ldots \sigma_n : \tau_n\}$
heap	$X ::=$	$\langle x_1 : \tau_1 \ldots x_n : \tau_n \rangle$
program	$P ::=$	$\langle \sigma, \tau, X, e \rangle$

TABLE 2: *Syntax of λ_A*

The *malloc* declaration allocates an uninitialized record or vector and binds a heap label to a variable x0. An initialization flag of 0 on the types of the fields indicates that the fields are uninitialized, and hence typechecking will fail if any uninitialized field is used anywhere in a term.

The *memcpy* declaration updates the indicated field of the record with a value and binds the address of the record to a new variable. Now the new variable is assigned a type where the first field has initialization flag 1, meaning that this field is initialized, and hence it can be used in terms. The rest of the fields in a record are initialized in the same manner.

A program in λ^A is represented by its name, return type, heap and program expression. The heap contains information about the types of all values that are allocated on the heap; i.e. records, vectors, and procedures. Because recursion is not permitted in the intermediate languages, the run-time memory requirements of the program can be inferred from the heap structure, and all required memory can be statically allocated prior to execution.

5 Backend Code Generation

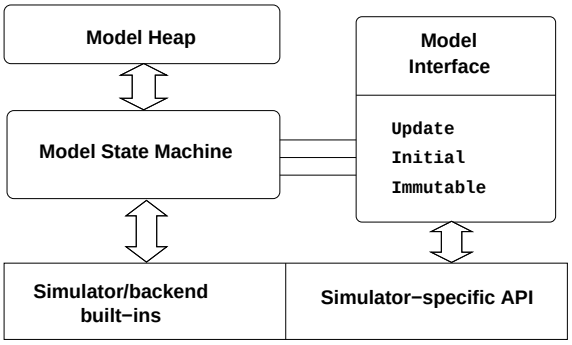


FIGURE 4: *Backend interface.*

Figure 4 presents the components of the backend interface. The procedures that form the model are non-recursive, and therefore they can be represented as a finite state machine. There are three entry points into the state machine, `initial`, `immutable`, `update`. The first two compute the initial states of the system, and the immutable states (such as interpolation tables). The update function computes the state for the next cycle, given the current state and timestep. All of the other heap values, such as records and vectors are represented as static structures in the target language of the backend.

There are also a number of simulator-specific functions that provide the interface between the simulator under which the model will be run, and the state machine representing the model. The built-in functions of the intermediate language also may have different mappings depending on the target simulator environment. Typically, those include the standard math functions, plus constructs to read and write data to and from external sources, such as data acquisition equipment, or functions for asynchronous event handling.

The translation from the intermediate language λ_A to low-level C code is relatively straightforward; the mapping of built-in functions in the intermediate languages to the primitives for a particular simulator API is again performed via an SXSLT stylesheet. Each vector and record type is translated to an equivalent C array or record type. Each procedure in the model implementation is assigned a numeric label and a corresponding block within a `switch` statement that encompasses the entire program.

A procedure call in the intermediate language causes the code generator to create temporary variables that reflect the types of the function arguments, and code that initializes those variables to the argument value, and then jumps to the beginning of the `switch` statement with the appropriate procedure label loaded in the global variable that is used as a procedure pointer.

6 Example Model

As an overview of the translation process, this section presents some of the transformation phases of a model. The example is a model of axon membrane currents that was first proposed by Alan Hodgkin and Andrew Huxley [5]. The Hodgkin-Huxley equations are generally regarded as the first example of a mathematical model of biology.

In the core of the model there are four ordinary differential equations that describe the dynamics of ionic currents flowing in and out of the cell. The total current is represented by the sum of three currents that in turn are dependent on the transmembrane potential, `V`.

Every model in the NMC input language consists of two sections. First is the declaration section, which we have omitted, and which specifies the names and initial values of all quantities in the model. Next is the equation section of the model:

```
I_Na  = g_Na * pow(m,3) * h * (V - E_Na);
I_K   = g_K * pow (n,4) * (V - E_K);
I_L   = g_L * (V - E_L);
/* Integration of the four state variables. */
d(V) = - (I_Na + I_K + I_L - I_stim) / C_m;
d(m) = alpha_m * (1 - m) - beta_m * m;
d(h) = alpha_h * (1 - h) - beta_h * h;
d(n) = alpha_n * (1 - n) - beta_n * n;
Vout1 = V;
```

The first three equations compute the values of the three ionic currents. Following is the equation for `V`, which sums the three ionic currents and an external stimulus current. The next three equations compute the values of the gate variables, which are the mathematical representation of the time dependence that exists between membrane conductance and membrane voltage.

Figure 5 shows part of the SXML tree that corresponds to the equation for the transmembrane potential `V` in abstract equation form. This is a differential equation that is numerically integrated using Euler's method. The arguments to the method consist of the expression tree that corresponds to the right-hand side of the equation in the original specification. The expression tree shown here is untyped; a subsequent stage in the abstract equation language stage will assign types to the arithmetic operations, depending on the types of the quantities involved.

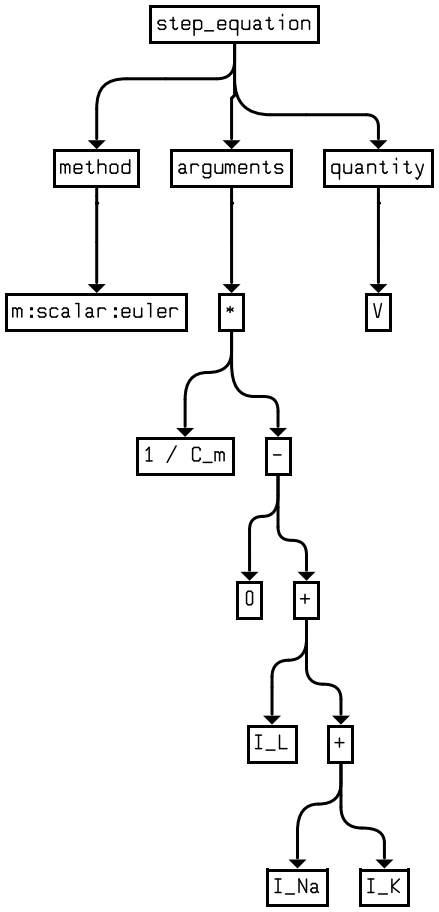


FIGURE 5: *Abstract equation language representation of the membrane potential equation of the Hodkgin-Huxley model.*

Figure 6 shows part of the SXML tree that corresponds to the intermediate language expression for the computation method function of `V`. In the intermediate language stage, there are three functions generated for each equation.

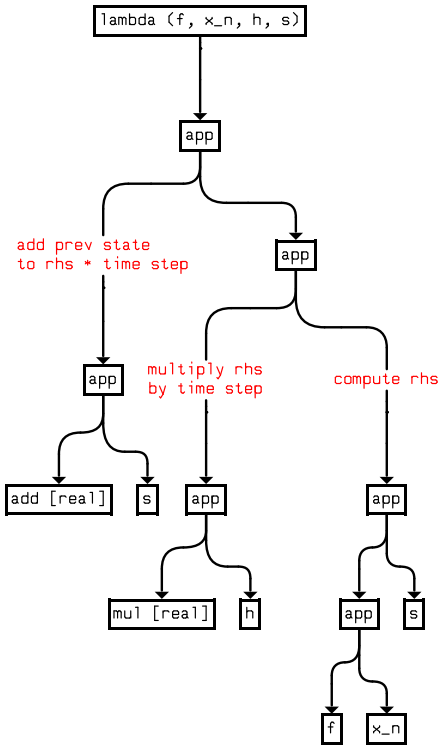


FIGURE 6: *Computation function for quantity `V` in the intermediate representation of the Hodkgin-Huxley model.*

The right-hand side function is the expression that corresponds to the right-hand side of the equation. It takes the form of a function whose arguments are a record that holds the values of the quantities that participate in the equation, and the state associated with this quantity, if any.

The computation method function is the function that implements the actual numerical method, such as Euler or interpolation.

The update function composes together the right-hand side function and the computation method function to obtain a numerical solution of the equation. The figure above shows the update function for `V`, in which the right-hand side function is bound to variable `f`. The right-hand side is computed by applying `f` to the input state vector `x_n` and previous state `s`, and then the result is multiplied by the timestep `h` and added to the previous state, according to the formula of Euler's integration method.

The arguments to the update function must correspond to the formal argument types shown in the next figure.

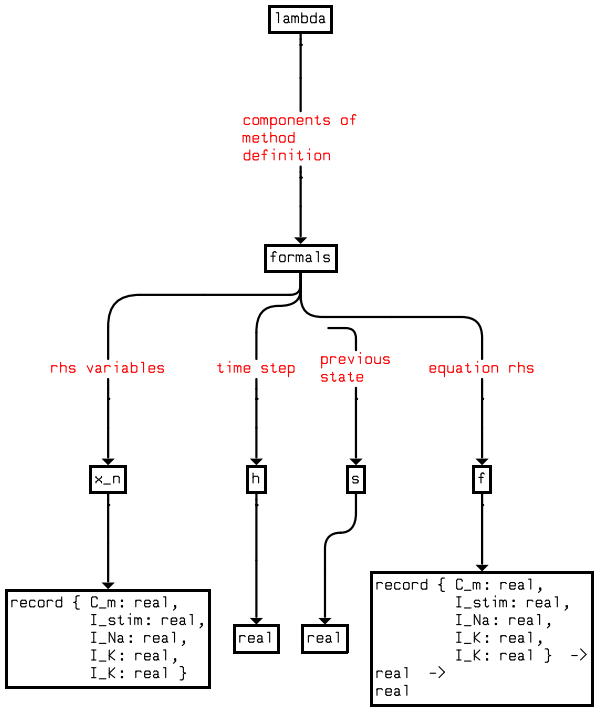


FIGURE 7: *Argument types of the computation function for quantity `V` in the intermediate representation of the Hodkgin-Huxley model.*

The intermediate language typechecker verifies that all operations in the model implementation are well-typed, and will therefore halt the translation process if the abstract equation translation process generates incorrectly typed code. In this particular example, the right-hand side function expects a particular set of arguments (that reflect the variables in the original equation), and a mismatch in the arguments passed to that function will be detected during typechecking.

An annotated sample from the generated C code is included below. The function shown corresponds to the right-hand side function for the membrane potential `V`, where all the participating variables are placed in a record on the heap, represented by the record type `t420`, and the arithmetic operations of the equation are performed in terms of the fields of that record. The result of the equation is passed on to the function that implements Euler's method. Because the intermediate representation uses continuation-passing style (CPS) semantics, the function does not return, but instead calls a receiver function with the result. The actual function call as performed as a goto statement to the beginning of the main `switch` clause.

```
case label_2163:
{
   t420 *x_env_2161;
   t5 x_353;
   t21 *x_354;

   /* read in function arguments */
   x_env_2161 = (t420*) argblock[0].Heapval;
   x_353 = argblock[1].RealNum;
   x_354 = (t21*) argblock[2].Heapval;

   /* perform computation */
   t5 cps130_82;
   cps130_82 = x_env_2161->x_349 +
               x_env_2161->x_351;

   t5 cps130_84;
   cps130_84 = cps130_82 + x_353;

   t5 cps130_86;
   cps130_86 = cps130_84 + x_env_2161->x_347;

   t5 cps130_87;
   cps130_87 = 0-cps130_86;

   t5 cps130_89;
   cps130_89 = 1.0 / x_env_2161->x_345;

   t5 cps10_90;
   cps10_90 = cps130_87 * cps130_89;

   t20 *clos3_2160;
   clos3_2160 = &x_354;

   /* set arguments for receiver function */
   argblock[0].Heapval = (& (clos3_2160->env));
   argblock[1].RealNum = cps10_90;

   /* perform function call */
   sm_mux = clos3_2160->code;
   goto sm_begin;
}
```

7 Extensions and Practice

After the initial version of NMC was completed, the necessity arose to extend the user-input language with support for asynchronous events. A real-time simulation environment that we are developing authors provides an event-oriented interface to signal the occurrence of asynchronous external events, which can be user or instrumentation input. If the event has occurred prior to the current computational cycle, a set of conditional equations will be evaluated, possibly with parameters supplied by the source of the event. The following syntactic structure was added to the input language:

```
// x is unchanged when no event has occurred
q(x) = x;
// x is assigned the value of k each time
// event x_event occurs
EVENT x_event (k)
[
     q(x) = a+k;
];
```

To accommodate this feature, the following steps were necessary:

1. Extend the parser with the appropriate syntactic constructs.
2. Augment the input XML schema with the new structures.
3. Add two new equation methods to the methods repository: one to obtain the "value" of an event, which is a boolean flag indicating whether the event has occurred; a second one to obtain the value of an event parameter.
4. Add two new primitive functions to the builtins repository used by the intermediate stage: a `sigtest` function to test for an event, and a `sigparm` function to retrieve the value of a parameter associated with an event. These functions provide a corresponding abstractions to functions in the target simulator architecture.
5. Extend the canonicalizer module for the input language to assign unique numerical values to each event and event parameter. This step was necessary because the `sigtest/sigparm` mechanism uses integer handles to identify events and their parameters. The input frontend to NMC can now assign integer handles to each event variable automatically, or a syntax extension of the language can allow the user the specify the mapping.

The only changes to the actual processing code were in the input language processing module (the first stage of NMC). The rest of the changes were additions to the methods and primitives repository. This example illustrates the flexibility of the design. The encapsulation and minimal dependence of the translation stages on one another allow individual features to be added to the user input language without altering any of the subsequent processing stages.

Intuitively, because of the high cohesion and low coupling of the stages of translation used by NMC, adding new features to the user input language affects the front-end the most, because it has to handle the specifics of user syntax; the abstract equation stage to a very small extent, provided that the new feature fits into the equation/method-oriented paradigm, and the intermediate stage the least, because it can represent any non-recursive computation.

The code generated by NMC is relatively independent of the execution environment, whether it runs in user space, or in kernel space under Real-Time Linux. A minimum of glue code is required to interface the model state machine to the API of the simulation environment. A practical issue that stands in the way of wide deployment is that the generated code tends to have a large number of memory copy operations, which are necessary for passing function environments (closures) across the chain of CPS calls. While there are many approaches to efficient closure representations in the compiler literature (i.e. [13]), our use of typed assembly language requires a careful incorporation of different function calling conventions within the type system.

However, the type system does not preclude other low-level optimizations, such as register allocation or instruction scheduling. In fact, the LLVM compiler infrastructure framework [8] was developed specifically with typed assembly language in mind, and one of the next logical steps in the development of NMC is to implement an LLVM backend that would allow the direct generation of efficient assembly code for different processor architectures.

In addition, we would also like to model networks of neuron models in real-time. A network modeling extension of the input language presents numerous challenges to the design of NMC. Modeling a network of models implies concurrent execution, possibly on several networked computers, if the models are too complex. The intermediate representation will therefore require semantics for concurrency and distribution, and lambda calculus is inherently sequential. Therefore, network modeling would require that semantics for distribution and concurrency are added to the intermediate representation. Our hope for the next step in the development of NMC is to incorporate simple calculi for distributed and concurrent computing, and to support different scheduler interfaces and real-time scheduling algorithms.

In summary, the approach taken with the initial implementation of NMC has shown promising results for developing a common platform for real-time scientific computing. The type system ensures that high-level abstractions are correctly represented at the machine-code level, and consequently the domain-specific frontend constructs can be efficiently and correctly translated to flexible machine-level representations. NMC provides a good foundation for high-performance computing in environments where code safety is an important property, and it provides a common base with general-purpose programming languages.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, 1995.

[2] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, 1997.

[3] Alan D. Dorval, David J. Christini, and John A. White. Real-Time Linux Dynamic Clamp: A fast and flexible way to construct virtual ion channels in living cells. *Annals of Biomedical Engineering*, 21:897–907, 2001.

[4] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Universite Paris VII, 1972.

[5] Alan Hodgkin and Andrew Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.

[6] Oleg Kiselyov and Shriram Krishnamurthi. SXSLT: Manipulation language for XML. In *Practical Aspects of Declarative Languages (PADL'03)*, 2003.

[7] R. Kumar, R. Wilders, R. W. Joyner, H. J. Jongsma, E. E. Verheijck, D. Golod, A. C. G. van Ginneken, and W. N. Goolsby. Experimental model for an ectopic focus coupled to ventricular cells. *Circ. Res.*, 94:833–841, 1996.

[8] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[9] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language (extended version). Technical Report TR97-1651, Cornell Univ., 1997.

[10] George C. Necula and Peter Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Jan 1997.

[11] Ivan Raikov, Amanda Preyer, and Robert J. Butera. MRCI: a flexible real-time dynamic clamp system for electrophysiology experiments. *J. Neurosci. Methods*, 132:109–123, 2004.

[12] H. P. Robinson and N. Kawai. Injection of digitally synthesized synaptic conductance transients to measure the integrative properties of neurons. *J. Neurosci. Methods*, 49(3):157–165, 1993.

[13] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *LISP and Functional Programming*, pages 150–161, 1994.

[14] A. A. Sharp, M. B. O'Neil, L. F. Abbott, and E. Marder. Dynamic clamp: computer-generated conductances in real neurons. *J. Neurophysiol.*, 69:992–995, 1993.

[15] A. A. Sharp, F. K. Skinner, and E. Marder. Mechanisms of oscillations in dynamic clamp constructed two-cell half-center circuits. *J. Neurophysiol.*, 76:868–883, 1996.

[16] Olin Shivers. A universal scripting framework or lambda: The ultimate "little language". In *Asian Computing Science Conference*, pages 254–265, 1996.

A Hardware Architecture Independant Implementation of GDB Tracepoints for Linux

Nicholas Mc Guire and Wang Baojun
SISE, Lanzhou University
Tianshui South Road 222, Lanzhou, Gansu, 7300000 P.R.China
mcguire@lzu.edu.cn, wangbj@lzu.edu.cn

Abstract

The problem of debugging applications in the value domain without halting execution due to temporal constraints imposed by components not under control of the debugged process is as only as IPC and networking. This problem does not have a simply single solution, but rather the solution depends heavily on the particular problem. Developing and thus debugging code that needs to ensure operation within well defined temporal bounds mandates tools to record the value domain without halting execution. This class of applications thus falls into the definition of real-time applications where the program's correctness depends on its time behavior. One possibility of relatively unobtrusively recording arbitrary values of an application during its execution are GDB tracepoints.

In this paper we describe our implementation and experience as well as limitations of the specification and our current implementation, followed by an outlook of future works we hope to continue.

1 Introduction

debugging of applications have two qualities to them, first one can look at functional correctness of execution, where results simply depend on the correct execution order of instructions and the correct initialization of data - this is the simple case of a standalone single process executable with restricted I/O. As soon as I/O or IPC is added things change - results no longer depend only on the correct order of execution but depend on the point in time when external values where requested or modified - we add a temporal dimension to the problem of debugging. For many simple setups IPC and I/O can "survive" with implicit synchronization or "well defined global order" - in distributed systems this well defined order starts crumbling - thats where debugging requires the ability to manage the temporal component explicitly.

Standard debuggers, like GDB, implement breakpoints, watchpoints and code/data inspection capabilities under the assumption that the program is halted at calculated addresses and can then be inspected without side-effects - the temporal behavior of applications is totally obscured by these techniques - thus debugging of distributed applications (local or networked) becomes very hard often mandating manual instrumentation to resolve cases that depend on interaction of multiple independent processes/nodes in a complex application setup.

This problem has been addressed by the GDB tracepoint [3] specification, though there has to date been no publicly available implementation, and thus the protocol has not been maintained for quite some time, resulting in multiple bugs sneaking in. Aside from those problems it turned out during the implementation phase that a number of shortcomings in the protocol specification resulting in useless commands, unclear resource conditions and sometimes simply irritating problems that require violating the current specification.

In this paper we describe our experience with implementing GDB tracepoints in a hopefully architecture independent manner, for GNU/Linux systems, and our first experiences on PPC405/440/970 as well as x86 systems.

2 Related work

The problem of temporal debugging is at least as old as the problem of multitasking and thus of synchronization it self. Typically synchronization problems stemming from temporal uncertainty - basically race conditions in a broad sense - have been traditionally

debugged with some form of instrumentation, ranging from brute force timestamping/printf to more sophisticated approaches like system level tracers [1] [2] and instrumentation. These approaches have addressed the temporal problem on a system level - since synchronization objects and protocols are provided by the operating systems in most cases, system level tracers are able to log events, and thus are potentially able to detect race conditions.

The limitation of these approaches is the inability to trace in the value domain of a particular application at arbitrary points in the code, which can be necessary as soon as one is involved in debugging protocols to implement synchronization and not system behavior related race conditions - notably the issue of network protocol debugging falls into the category as multiple systems in a non-synchronized environment are involved.

A further limitation of existing solutions is that the implementation is either very coarse-grain (i.e. LTT) or very invasive (i.e. function-instrumentation [**?**]). As long as the problem is related to "pure" software debugging, that is the races are due to the temporal behavior of CPU executed code only, the performance degradation can be acceptable in some situations as long as the degradation is evenly distributed. Uneven distribution of instrumentation is never useful for reliably detecting race conditions - the well known effect of "works with debugging enabled but fails when optimized" falls into the category of distorting instrumentation which changes temporal behavior enough to disguise race conditions.

If one now assumes well designed homogeneous performance degradation of sufficiently small amounts, one can debug temporal problems of complex software systems, but can't easily manage temporal constraints of independently operating units, may that be a local I/O subsystem which is "too smart" or a distributed system with, in the time domain, loosely coupled nodes. In this situation one generally will resort to non-invasive punctual instrumentation, i.e. printk and friends... and that is where GDB tracepoints conceptually come in.

3 Introduction to tracepoints

If one were to set a breakpoint at an arbitrary instruction in an application running under control of GDB, the application is stopped when hitting the breakpoint, by invoking a breakpoint trap handler. This is achieved by simply taking the line of code at which the application is to be halted and exchanging the machine instructions at that location by a breakpoint trap call (i.e. 0xCC for ia32 systems, or "twge r2, r2" in the case of 32bit PPC). The trap handler provided by GDB is called and control of the application is handed to the interactive terminal of the debugger - this is what breaks the temporal behavior. The human observing the systems execution will react in seconds or minutes to a hit breakpoint, thus suspending execution of this one process in an inappropriate way for many protocols needless to say for multiprocess/multi-threaded application synchronization.

So if one now would modify the breakpoint handler to not hand control to the interactive terminal of GDB but would simply execute a set of predefined actions when hitting a breakpoint and continue execution - which is basically what the engineers will do manually during a debug session - then you have the notion of a tracepoint. The question is now "simply" how do you encode the actions to be taken at the breakpoint which is going to be operated as a tracepoint

3.1 GDB tracepoint concepts

GDB tracepoints are intended for remote targets only. The idea is to have only a lightweight process, the gdbserver, or the gdb-stubs, running locally and the interactive session running off-line. GDB uses the remote protocol to communicate with the target system, and assumes the remote application will be launched under control of gdbserver (or gdb-stubs) in an halted state - allowing to configure tracepoint before launching the critical section. Further the assumption is that the application can be halted before exiting normally (i.e. just before the call to return or _exit) allowing to inspect the collected objects in the appropriate context.

First a QTDP packet is sent containing the basic tracepoint setup information which is:

```
pkt_data:          tracepoint setup data
num                tracepoint number
mem_addr_ptr       address of the tracepoint
state              tracepoint state
step               tracepoint step
pass               tracepoint passcount
actions            tracepoint have fowllowing
                   packet to speficy actions
```

This is decoded by the decode_tr_packet function in remote_utils.c and actually creates the tracepoint entry, setting up the breakpoint and preparing the data structures for the actions that describe the data collection for the given tracepoint.

Data is collected in trace-frames, each tracepoint creates a new trace-frame when the respective tracepoint is hit and stores data in this save storage area.

later request for inspection of trace data use normal syntax (that is register and memory queries - see the GDB remote protocol for details) but rather than satisfying these request via ptrace calls to the traced application the requests are answered from the traced data objects.

GDB provides three types of object to be collected and a set of commands to setup tracepoints. The three types of trace objects are:

- registers
- memory areas
- expressions

These objects are passed to the target with QTDP continuation packets, which follow the initial QTDP packet, the processing of continuation packets is coded in decode_tc_packet in remote_utils.c.

3.1.1 Tracing Registers

The tracepoint protocol specifies the "R" packet to record registers, whereby a mask is passed to identify which register to record. From the concept of tracepoints though a full set of all registers is actually required at every tracepoint any way making the "R" packet unnecessary. In our current implementation we followed the spec and thus provide the "R" packet, in future releases we think it would be better to remove it - provided it is removed from the specification as well. It does not seem to be an arguable performance advantage to pass a register mask, as processing this register mask is most likely more effort than can be saved by storing all registers - this might be false though for architectures with very many registers (i.e. ia64).

adding registers to the tracepoint actions is done by

```
main(){

   while (getpkt (own_buf) > 0) {
     switch {

      case QTDP:
          |
          `--> decode_tc_packet
               |
               `--> add_tp_register_actions
                    |
                    `----> insert_reg_list
```

At runtime the register action is processed in the tracepoint handler, which is assigned to the breakpoint as the breakpoint trap handler and coded in tp.c.

```
tracepoint_handler(){
   ...
   struct tp_regs *tr;
   ...
   list_for_each_entry(tr... ){
     dump_registers(tp, tr);
     |
     `--> malloc
           |
          registers_to_string
           |
          list_add_tail
             (...curr_frame->data_list..)
```

This data now contains the hexadecimal values of all registers stored as string (which happily eliminates endiannes problems).

Retrieval happens after the completion of the trace experiment by a call to `tfind` which sends a regular register request, but instead of satisfying this register request from the actual target register set, it looks up the appropriate register in the trace data and returns it to the host.

```
main(){

   while (getpkt (own_buf) > 0) {
     switch {

     case QTF:
          |
          `--> decode_tf_packet
               |
             find_tracepoint
               |
             tracepoint_num = tp->which
               |
               `--> collect_tp_data
                      (tracepoint_num,
                      traceframe_num)
                   goto out

out:
   struct tp_data* td
   ...
   find_tracepoint_nr
     |
   td = tp->tp_actions->curr_frame
```

At this point the traced context is set and all subsequent requests for register packets will be answered using data stored in curr_frame.

3.1.2 Memory traces

The structure for memory traces is really more or less the same - the only difference being the data collection routine called in the tracepoint handler.

```
tracepoint_handler(){
...
struct memrange *ml

  list_for_each_entry(ml... ){
    dump_memrange(tp, ml);
      |
      `--> malloc
             |
           read_inferior_memory
             |
           list_add_tail
             (...curr_frame->data_list..)
```

The actual memory data collection routine, which is provided by GDB, is read_inferior_memory, which is called to read the memory content of the ptraced process under control of gdbserver. Data retrieval is the same as shown in the previous section for registers.

3.1.3 Expressions

Expressions are a bit different - expressions are sent to the target as agent expressions and must be interpreted at runtime, this is done by the dump_bytecode function that is a 0-register machine. After resolving the appropriate expressions to memory locations these memory locations are finally retrieved with the read_inferior_memory - just like for memory objects.

The bytecode provides basic functions to read registers and memory and to operate on the read values with logical and arithmetic operations (add, left-shift etc.) finally it provides trace commands to store the results in a safe location for later retrieval.

```
tracepoint_handler(){
...
struct tp_bytecode *bc;

list_for_each_entry(bc... ){
  dump_bytecode(tp, bc);
  |
  `--> for(i=0;i<bytecode->length;){
         switch(bc[i++]){
           case BC_ADD:
           {
             push(pop() + pop());
           }
           break;
           ...
```

```
case BC_TRACE:
{
  unsigned int size = pop();
  unsigned int addr = pop();
  add_bc_list(addr, size, tp);
    |
    `--> malloc
           |
         read_inferior_memory
           |
         list_add_tail
           (...curr_frame->
                 data_list..)
}
break;

case BC_TRACE_QUICK:
{
  unsigned int size = 0;
  unsigned int addr = pop();
  size = (bc[i++] << 0);
  push(addr);
  add_bc_list(addr, size, tp);
    |
    `--> malloc
           |
         read_inferior_memory
           |
         list_add_tail
           (...curr_frame->
               data_list..)
 }
 break;
 ...
```

Retrieving of the recorded data is then done via memory request packets to the target, and these are then satisfied by the memory objects that were traced with the above shown BC_TRACE_QUICK and BC_TRACE.

The current interpreter is a pragmatic implementation, it well may be possible to optimize it - but at this point we don't see this as a real bottle neck.

3.2 Concept Limitations

As is true for most GNU projects, it is not (yet) perfect, limitations are in part implementation issues - in part of conceptual nature.

3.2.1 PTRACE

Processes being debugged must run with the PTRACE flag set, thus under control of a monitoring process - the gdbserver in the case of GDB - this is an inherent limitation of the debugging strategy and not tracepoint

related though. PTRACE is required because it enables gdbserver to change the core image of the traced process. Basically the traced process behaves normally until a signal is caught. The limitation incurred by ptrace is a performance penalty and a fairly high overhead for gdbserver to access the data/code/stack of the traced process - this is an inherent limitation of GDB which heavily builds on the ptrace system call. The heavy usage of ptrace to access the traced process context causes a substantial overhead which affects tracepoints as well as they must rely on the ptrace mechanism to modify and inspect the traced process. Note that modification is necessary to implement software breakpoints, as the instruction removed to insert the breakpoint code must be reinserted executed and then the breakpoint reinserted - thus the code segment of the traced process must be modified using ptrace system calls. A possible improvement would be a ptrace_v system call that allows to pass multiple modification requests in a single system call.

3.2.2 bytecode efficiency

Agent expression are not the most efficient way both conceptually and with respect to the factual implementation. Starting at suboptimal bytecode it is not supprising that running an interpreter at every tracepoint to collect the requested objects is not going to be the most efficient. Moving to a JIT compiler for the bytecode in addition to caching the generated statements could resolve this issue - aside from off course cleaning up some of the known inefficiencies in GDB agent expressions.

3.2.3 expression granularity

On the one side it is an advantage of tracepoints that they allow a fine grain inspection, on the other hand the fact that they mandate fine grain definition of trace objects is sometimes irritating in practice - especially larger structures can't be traced on a structure object basis but rather must be traced element for element.

3.2.4 breakpoints

Currently breakpoints and tracepoints can't coexist at the same instruction address, though this is a limitation we don't concider it dramatic as mixing tracepoints with breakpoints in the active code would defeat the whole pupose of tracepoints in the first place (no temporal guaratees with breakpoints). Currently we don't plan to lift this bug.

3.2.5 storage issues

The current protocol has no means of providing information regarding the number of tracepoints that will be hit (or some sort of maximum) and storage needs so that allocation could be done before starting a trace experiment (that is a run with armed tracepoints), thus currently one only has the two options of either allocating memory dynamically at runtime - which is very invasive - or allocate a "large" slab of memory and hope that one can satisfy the tracepoint needs with this preallocated memory region. The obvious solution is to ensure that the trace start command (tstart) would provide the necessary (and known) object size information and that some run-constraints parameters are added to tstart to allow limiting the number of traced events (thus eliminating runtime error checking as well).

3.2.6 API issues

In the current specification of GDB tracepoints we belive there are also some short commings that need to be fixed.

- `tstop` is a more or less useless command, as once the trace experiment is stoped with `tstop` recorded data is no longer available.

- stepping / while-stepping - currently stepping is specified, but using steping has such a dramatic impact on the application that it - atleast in its current form - would make absolutly no sense for applications with even mild real-time requirements.

- A parameter to ither `trace` or `tstart` which allows to specify how many tracepoints will be collected would be needed to allow proper preallocation.

- gdbserver should not terminate if a trace expreiment has been launched, i.e. by automatically setting a breakpoint at return / _exit and not requireing manual setup

- agent expression need a review with respect to the way offsets are handled currnetly a few useless add/sub instructions of 0-offsets are inserted on every trace.

None of these items listed here is though a show stopper for tracepoints - but in the current form it is not resonably posible to provide a halfway optimized implementation - which is a serious limitation to the tracepoints applicability.

4 Implementation

The implementation of tracepoints was based on the tracepoint specification and the remote protocol specification - both part of GDB. The main design decision to be taken were:

- breakpoint implementation method
- memory allocation strategy
- data organization on the target side
- encapsulation in files

For the coding it self, the painful GDB coding-style was to be applied.

Ad 1: The question is if a "optimized" breakpoint should be used or the breakpoint interface all ready available. We decided to use the available breakpoint interface with a statically defined handler because this would buy us a architecture independent implementation right from the start. The drawback at present is only that currently a breakpoint and a tracepoint can't coexist at the same location - which functionally makes little sense any way. A minor disadvantage in our believe is that the implementation could be somewhat more efficient if it were using a dedicated breakpoint implementation.

Ad 2: The discussion about the best suited memory allocation strategy focused on the issue of a pool allocator. The tracepoint is recording same-size objects at every hit tracepoint, thus lending it self to a slab/cache allocation strategy based on a preallocated memory pool. The problem with this approach turns out to be that the tracepoint protocol has no parameter to inform the target how many tracepoints will be hit - thus the initial pool-size can't be set with any reasonable heuristics at this point. For the first implementation we thus fell back to a complete runtime allocation on top of malloc and will postpone the pool allocator to after the necessary discussion with GDB folks on extending/modifying the tracepoint API.

Ad 3: Data organization was focusing on efficient access at runtime only, it is fairly irrelevant what effort is necessary to retrieve data after the trace experiment has completed - only efficient storage of trace data at runtime is relevant. All objects are organized in doubly linked lists at this point, but with allocation happening at runtime the efficiency is obviously low and dominated by the malloc calls, never the less an efficient data layout is essential assuming that the allocation problem can be resolved with a pool allocator in the future.

Ad 4: Source code maintainability was a key issue from the very beginning - it is clear that at least in the initial development phase this tracepoint implementation will be an external patch to GDB, thus the modification were split into:

- bug-fixes to existing code
  (i.e. gdb/tracepoint.c gdb/dwarf2loc.c)
- remote protocol extensions
  (i.e. gdb/gdbserver/remote-utils.c—h)
- the core tracepoint functions
  (gdb/gdbserver/tp.c—h)

this results in tp.c being a bit of a lengthy file at present, but taking the existing code for GDB we are not out of bounds for now.

The intention is to feed back the bug-fixes into the current development tree as early as possible and feed back the actual tracepoint implementation as soon as it has gotten some reasonable testing by the community.

4.1 Status

The current status is that tracepoints are functional, allowing to trace c and c++ expressions as well as registers. The current implementation is for gdb-6.3 and has been tested successfully on PPC405, PPC440, PPC970 and ia32. Whereby "tested" is to be read as "used with some example code" and not read as a thorough testing of any sort - not yet. With c/c++ expressions we mean

- simple type global and local variables
- pointers
- structure elements
- simple expressions (i.e. a + b)
- registers

4.2 Known bugs

- Not all of `tfind` is implemented - currently only `tfind start`, `tfind tracepoint #` are implemented.
- passcounts are currently not implemented yet - though this should not be too wild to do.
- With the current implementation it is mandatory to collect all registers using the `$regs` variable explicitly, if omitted nothing can be retrieved.
- floating point has not been implemented yet - thus floating point values can't be traced.
- the first call to `tfind` is ignored requiring to call `tfind` twice to retrieve the first data item.

- as there is no way to retrieve data once a process has terminated it is mandatory to set a breakpoint at the end of the application to allow data retrieval, omitting this breakpoint results in loss of all traced data (or actually the ability to access it as gdbserver terminates once the application returns or calls _exit).
- ia32 has a problem because by default the agent expression will use the stack pointer and not the base pointer - this is a GDB but though and not related to tracepoints.

As noted - this implementation is not out in the community yet - so we expect other bugs to emerge soon.

5 Usage

This is not a complete usage manual, we refer you to the GDB tracepoint manual for details, with the limitations listed below in the section Usage Limitations.

5.1 Simple Session

To explain our sample session we assume that we have a simple c file with a few global variables a structure and some local variables to be traced.

```
root@rtl22:/home/gdb-6.3-tp/gdb# ./gdb
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu".
Setting up the environment for debugging gdb.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x400007b0 in ?? ()
(gdb) symbol-file gdbserver/hello
Reading symbols from /home/gdb-6.3-tp/gdb/gdbserver/hello...done.
(gdb) list 48
43 char *msg="hello world";
44 t = (a_t *)malloc(sizeof(a_t));
45 while(i<4){
46 junk();
47 t->a=i+1;
48 printf("t->a is at %lx\n",&t->a);
49 t->b=i+1;
50 junk2(&t->a,&t->b);
51 i++;
52 printf("i is at %lx\n",&i);
(gdb) trace 52
Tracepoint 1 at 0x804851d: file hello.c, line 52.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
> collect $regs
> collect t->a
> collect t->b
> collect i
> end
(gdb) list 54
49 t->b=i+1;
50 junk2(&t->a,&t->b);
51 i++;
52 printf("i is at %lx\n",&i);
```

```
53 }
54 return 0;
55 }
(gdb) break 54
Breakpoint 1 at 0x8048533: file hello.c, line 54.
(gdb) tstart
(gdb) continue
Continuing.

Breakpoint 1, main () at hello.c:54
54 return 0;
(gdb) tfind
During symbol reading, Incomplete CFI data; unspecified registers at 0x080484f1.
54 return 0;
(gdb) tfind
0x0804851e 52 printf("i is at %lx\n",&i);
(gdb) while ($trace_frame != -1)
 >printf "TF %d, PC %x, i %d, t->a %d, t->b %d\n",\
$trace_frame, $pc, i , t->a, t->b
 >tfind
 >end
TF 1, PC 804851e, i 1, t->a 1, t->b 1
TF 2, PC 804851e, i 2, t->a 2, t->b 2
TF 3, PC 804851e, i 3, t->a 3, t->b 3
TF 4, PC 804851e, i 4, t->a 4, t->b 4
TF 2147483647, PC 8048533, i 4, t->a 4, t->b 4
invalid input (-2147483648 is less than zero)
(gdb) continue
Continuing.

Program exited normally.
(gdb) quit
root@rtl22:/home/gdb-6.3-tp/gdb# exit
```

As cam be seem the final output of the trace is not quite clean, the first `tfind` does not jump to the expected location and the termination condition of trace_frame being -1 is not hit, these are known bugs, but don't have a functional impact - this is after all - work in progress. Note also that most gdb commands can off course be abbreviated (i.e. c for continue) and they are only shown in full length for readability.

5.2 Saving and restoring Tracepoints

As with most GDB commands you can save them into files - in the case of tracepionts you simply can use `save-tracepoint FILENAME` to store the tracepoint command into a file. The format is simple enough to allow for automatic generation (i.e. for system tests or the like).

```
trace hello.c:28
  actions
    collect $regs
    collect my_variable
    end
```

to read it at the interactive prompt of GDB use `source FILENAME`.

5.3 Usage Limitations

There are a number of limitations to the current implementation we all ready listed, we will list the usage limitations here - so of which are not bugs or shortcomings of tracepoints but simply a consequence of the trace process, that mandate a slightly different approach when used.

- Use few tracepoints only - or the application runtime behavior will be too distorted to say much.
- trace exact values - you should not try to trace complete structures but rather pin point the variable of interest and only trace that object.
- you only can inspect exactly what you traced, so for instance tracing struct t and then trying to

inspect t->a will fail ! if you need t->a then you must trace t->a.

- you need a breakpoint before the application exits - in the above example at the return statement - or you will lose the context and will no longer be able to inspect traced objects.
- setting tracepoints at function names is possible - you must not use line numbers, but be sure that all variables that you want to record are actually initialized at the point where you set the tracepoint or it will fail for obvious reasons - you can't trace NULL pointers ...
- careful with optimizations, if you trace in -O2 code then it can lead to unexpected values because execution order can be changed, thus don't expect strict code order if using optimization !

There surely are more - we will add then - and hopefully fix most limitations - when usage reports start coming in.

6 Conclusion

With all its incompleteness, concept limitations of principal nature and our implementation limits, we still think this is a valuable tool for debugging of applications especially in the distributed environment. What this prototype implementation shows successfully is that the mechanism works reliable and can be implemented in a architecture independent manner.

The protocol in our opinion needs some cleanup, and some well-meant features, i.e. stepping, need to be re-thought if not simply thrown. With the proposed improvements of:

- specification of number of tracepoints (allowing preallocation)
- implementation of a pool allocator with allocation at `tstart` time
- improvements of the ptrace system call (i.e. ptrace_v) allowing to group multiple ptrace requests in a single system call
- and some minor fixes to the tracepint API

we belive that GDB tracepoints could be a very valuable tool for debugging of distributed and real-time applications aswell as protocol stacks.

7 Acknowledgment

I would like to thank Wilfried Moser from Alcatel Vienna for many helpful discussions and pointers when we got stuck in the swamps of GDB code. Further thanks to Benjamin Collar of Siemens CT SE2 for testing and help on debugging in the initial development.

8 List of Acronyms

`FDL` - Free Documentation License
`GDB` - GNU DeBugger
`GNU` - GNU Not UNIX (recursive accronym)
`GPL` - General Public License
`IPC` - Inter Process Communication
`JIT` - Just In Time (Compiler)
`KFI` - Kernel Function Instrumentation
`KGDB` - Kernel GDB
`LTT` - Linux Trace Toolkit
`POSIX` - Portable Operating System Interface (for UNIX)
`PPC` - PowerPC (Processor)
`RT` - Real Time

References

[1] [Linux Trace Toolkit] Karim Yaghmour, *LTT, LTTng*, http://ltt.polymtl.ca/, 2006

[2] [POSIX Trace] A. Terrasa, A. Garcia-Fornes, A. Espinosa, *http://www.ocera.org/download /components/WP5/ptrace-1.0-1.html*, University of Valencia, 2005

[3] [GDB Tracepoint Spec] Richard Stallman, Roland Pesch, Stan Shebs, et al. *gdb_11.html section 77*, FSF 2006

[4] CE Linux Forum, *Kernel Function Instrumentation*, `http://tree.celinuxforum.org`, (C) CE Linux Forum Member Companies, 2005.

The Study of Integrated Model Research Systems Based on GNU/Linux in Heihe River Basin

Zhang Yaonan[a,b], Luo Lihui[a,b], Gao Meirong[c], Lu Yu[a,b], Kang Jianfang[a,b], Zhang Baoshan[a,b], and Qiang Xiaoming[a,b]
[a]Cold and Arid Regions Environmental and Engineering Research Institute, CAS
320 Donggang West Road Lanzhou, Gansu 730000, China
[b]Gansu High Performance & Grid Computing Center
Lanzhou, Gansu 730000, China
[c] Institute of Mountain Hazards and Environment, CAS
#.9, Block 4 , Renmingman Road, Chengdu, 610041, China
{yaonan,luolh}@lzb.ac.cn

Abstract

Data integration environment, modeling and simulation environment, operating and manipulation environment based on GNU/Linux are important support platform of observation and integrated research of ecological-hydrological processes in Heihe river basin. Embedded Linux combined with observation sensor networks and application software, formed a data integration environment of data collection, processing and analysis. Some GNU software and development software based on Linux system constitute an operating and manipulation environment. SME(Spatial Modeling Environment) of GNU software and MMSModular Modeling System constitute a modeling and simulation environment in Heihe river basin to support collaborative model building among a large, distributed network of multi-disciplines scientists involved in creating a global-scale ecological/economic model, providing spatial modeling and integration with capacity of distributed and collaborative processing. The platform will be used primarily for graphics interface modeling tools to implement modeling, connect with high-performance computing resources, the model base and a generic object database, export visual simulation results in the form of 2D animation, 3D animation, image spreadsheet and numerical spreadsheet to support simulation and forecasts of the ecosystem and hydrological processes. A modeling and prediction experiment about Heihe river basin ecosystems has been done in this system.

1 Introduction

The Heihe River is a typical inland watershed in the northwestern China and in Central Asia in which water resources play pivotal roles in a multifaceted and mosaicked landscape. The rational use of resources is also focus in scientific community and decision-making departments, so needs to study how quantitative description of Heihe river basin eco-hydrological processes mechanisms in integrated Model level, and find mutual mechanism in the processes[1] on the basis of land surface processes research[2]. needs to support collaborative model building among a large, distributed network of multi-disciplines scientists involved in creating a global-scale ecological/economic model, providing spatial modeling and integration with capacity of distributed and collaborative processing based on accumulated mass data in distributed network environment. In support of these studies, require to build a data integration environment of observational sensor networks with embedded Linux, need to build a spatial modeling and simulation environment which made by of SME (Spatial Modeling Environment) of GNU software and MMS (Modular Modeling System) in Heihe river basin to support collaborative ecological/economic model building of multi-disciplines scientists, to provide distributed and collaborative data processing, to provide web-based modeling and simulating function and give 2D and 3D animation, graphics, image spreadsheet, numerical spreadsheet, visual and real-time simulation and

prediction results.
In recent years Linux application is gradually wide and a lot of free applications software of suitable for scientific computing supply by GNU, such as SciLab of numerical calculation software, Maxima of algebraic calculation software, it formed some conditions to build a integrated research environment for Heihe river basin.
Now we build a data integration system, modeling and simulation system, operation and manipulation system based on GNU/Linux in Dawning TC1700 cluster servers. Preliminary do the some system integration research experiments such as collection, processing, analysis, modeling, simulation, forecasting for data come from Heihe river basin. And preliminary simulate the modeling and prediction of "economic-water-ecological" which established for Heihe river basin, implement output simulation results with visual and real-time in browser in the form of 2D,3D animation, graphics, image spreadsheet, numerical spreadsheet.

2 Integrated Model Research Systems

Linux-based development software and GNU software make up operation and manipulation environment of GNU/Linux. we use embedded Linux, observational networks and analysis software to constitute a data integration environment of Heihe, and transplant, install and operate the models of VIC, SHE, TOPMODEL, SWAT, SWMM, MMS, SME and other hydrological and spatial model in this environment, formed a spatial modeling and simulation environment for Heihe river basin. The integrated models research environment formed by three environments As shown in figure 1. and the description of three environments as fellowing.

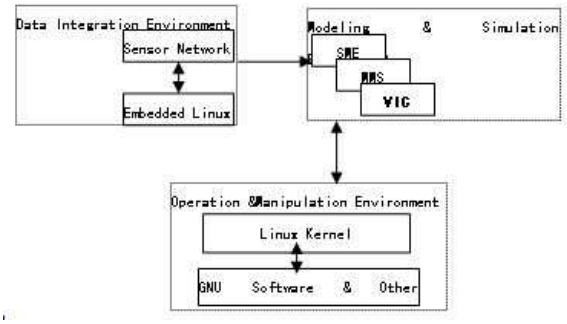


FIGURE 1: *Frame Diagram of Integrated Model Research*

2.1 Systems Data Integration Environment

An embedded Linux is taken the kernel as foundation, complete memory protection, multitask, and multithread operating system. All open source code can be modified and customized software in GPL (General Public License). Observation sensor networks and embedded Linux constituted data integration environment. These observation networks which integrated sensors technology, embedded computing technology, distributed information processing technology and wireless communication technology can coordinate and conduct real-time observation, apperceive and collect various elements of the environment, and handle date, transfer data information to users. Observation networks divided into cable sensor observation network and wireless sensor observation network. Cable sensor observation network adopt Ethernet to access[3]. Used the techniques of mobile network nodes and self-organizing, the wireless sensor observation network placed flexible and expand to easy, collect data in no man's land and cable networks could not work on the areas. Sensor observation network observed the temperature, humidity, wind speed, wind direction, atmospheric pressure and other signals of environment location region, then processed the collection data, and transmit to the main node with embedded Linux through network, an influx of data to the main node will be transmit to the main Linux server of observation through network. About 10 sensors responsible for data transmit to embedded Linux system, there are many such observation networks in the upper, middle and lower reaches of Heihe river basin. In the observation, generally requests the sensor real-time deliver the data to the main node, at the same time, according to local the network environment, the main node may act really now or delay transmission and discontinuous transmission data to the linux server.
The figure 2 shows the data integration environment, Observation sensor network, signal controllers, modems, launching and receiving modules, embedded Linux.

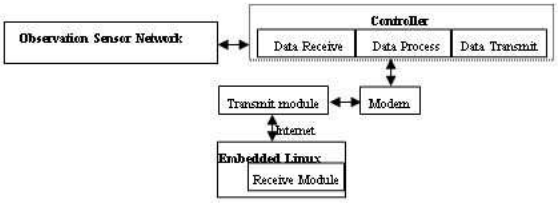


FIGURE 2: *Frame Diagram of Data Integration Environment*

2.2 Modeling and Simulation Environment

In the modeling process of Heihe river basin, using SME and MMS to simulate and predict of "economic-water-ecological" process of Heihe river. This article introduces spatial modeling and simulation environment of SME, not involve details about Stella modeling tool[4].

The SME has assumed by the U.S. University of Maryland, the SME has been partially funded by NSF Award and the NCSA Alliance. The SME is based on large-scale interdisciplinary scientist's common modeling and simulation of high-performance integrated model in spatial environment. This environment, which transparently links icon-based modeling environments with advanced computing resources, allows modelers to develop a model and operate simulation in a user-friendly, graphical environment, requiring no knowledge of computer programming, to support modeling a complex systems in spatial for researcher. The functions of SME is a mainly to build and simulate some models for economic activity, climate, population, scientific and technological, environmental changes and other economic, ecological, sociological fields. It can simulate and forecast large-scale and global ecological / economic environment, export visual simulation results in the form of 2D animation, 3D animation, image spreadsheet and numerical spreadsheet, and providing spatial modeling and integration with capacity of distributed and collaborative processing. The environment will allow users to create and share modular, reusable model components, and utilize advanced parallel computer architectures without having to invest unnecessary time in computer programming or learning new systems. The environment imposes the constraints of modularity and hierarchy in model design, and supports archiving of reusable model[5] components defined in our Modular Modeling Language (MML). The SME structure consist of four basic applications: output of modular, code generation, SME-driver and JAVA programming interface. Users can transfer these functions through command-line interface or the SME JAVA graphics interface.

The SME adopt uniform and standard MML (modular modeling language) description language which can be used immediately to construct a working spatial simulation. There are two methods for generating MML code: using a text editor to write a model in MML, or use an external modeling package and import its (exported) equations into MML. We used Stella to create models in Heihe study, and then export appropriate Stella equation file. These equation file used to describe time step change of algebra and logical equation. The SME parse mathematical expression of equation file in Linux, and then extract all kinds of constants, variables and logical relation between variables. An equations file from the STELLA unit model that will be translated into MML by the MCP (Module Constructor) application. Then, The MML objects can be archived in the Model Base to be accessed by other researchers, and/or used immediately to construct a working spatial simulation. Many MML objects can be combined hierarchically in the MML. This MML hierarchy can be converted by the Code Generator into a C++ object hierarchy in the spatial modeling environment (SME), where it can drive a spatial simulation. The SME Driver is a distributed object-oriented simulation environment which incorporates the set of code modules to perform the spatial simulation on the targeted platform. The simulation is executed within the SME, which provides numerous simulation services such as transparent distributed computing, integrated visualization and analysis tools, and integrated GIS and database access. Structure diagram of SME shown in figure 3.

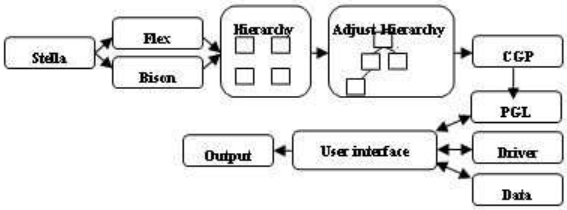


FIGURE 3: *Structure Diagram of SME*

2.3 Operation and Manipulation Environment

Linux kernel, some GNU software and Linux-based development software (this isn't discussed in the article) constitute operation and manipulation environment of GNU/Linux, provide run request functions of program compiling, data processing, data analysis, program interaction, users interaction, service requests, resultant output, etc. The main implementation of system software including: standard compiler GCC (GNU Compiler Collection), data manipulation and data analysis software HDFHierarchical data format and GRASS GIS (Geographic Resources Analysis Support System), program interaction software Tcl/TkPython, and other management software.

HDF (Hierarchical data format) is a hypertext document and hierarchical storage data format. HDF such a common data storage formats can be integrated management of 2D, 3D, vector, attribute, text and other information. Using HDF file format storage of different types of data in Heihe river basin

, including images, multidimensional array, pointer, text, and satellite data. HDF use metadata to preserve array dimension information in order to implement self-description of file structures. Users need to deploy HDF library function to read-write data objects.
Grass GIS (Geographic Resources Analysis Support System) is a GNU geographic information systems (GIS) software, which is also a mixed, modular GIS with a vector and grid function. It handles primarily spatial data and remote sensing image data in Heihe river basin.
Tcl/tk is a common scripting language, mainly for issue some commands to program interaction software such as text editor, debugger and shell. We compile Tcl/tk interface in integrated model research systems, reading SME application from user command-line interface, in order to implement Tcl/tk commands.
GNU software has a broad field in science applications, many software can replace similar business software in functions. Therefore, we preferred GNU OSS (Open-source software) to set up integrated model research systems based on GNU/Linux in Heihe river basin.

3 Experiments of Integrated Model Research Systems

SME includes water, energy, chemical and ecological processes. This system can couple hydrological models, atmospheric models and ecological models to build integrated coupled model, it take two ways data feedback during various models' coupling into account. In order to solve soil-water resources, ecology and environment how to respond to climate change and human activities, it was estimated and predicted that glacier, snow, permafrost, vegetation, soil erosion and runoff changes in upper reaches of Heihe river basin, as well as water resources transform, development and use of land and water resources and ecosystem changes in middle portion of Heihe river basin, desertification ecosystem changes in lower reaches of Heihe river basin. After conversing the DEM map to map format, adding land-use maps, vegetation cover maps, and precipitation, wind speed, wind direction, atmospheric data etc. in data integration environment, the data have transformed into time-series data and observation points in a time-series data according as respective requirements, which is related to the establishment of models for SME data and parameters ready. In the SME environment, experiment was done based on the use of water-land-air-health-man integrated model of the in Heihe river basin. Output figure 4 shows the SME model result in Heihe river basin.

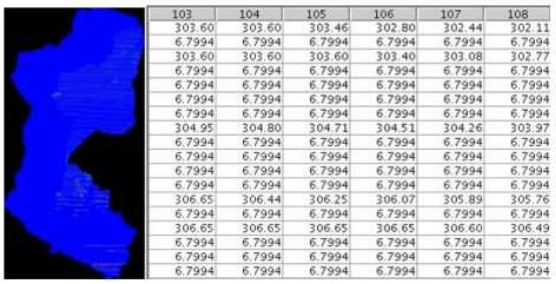


FIGURE 4: *Outcome Diagram of Simulation Output*

4 Discussion

In order to resolve the current integrated model level which study on Heihe river basin Integrated model research systems based on GNU/Linux in Heihe river basin has preliminary established and provided a viable route. The system also provides a platform for the integrated ecological, social and economic sustainable development, rational use and management of water resources and observation and research of natural processes, but, there are many issues will wait for later solution and improved.
The following work mainly focused on integrate model which is useful, any, environmental, economic and social models, then be able to obtain a water-land-atmosphere-ecological-human integrated model. Build perfect observation system in upper, middle and lower reaches of Heihe river basin, based-Linux cluster servers, construct perfect modeling and simulation environment. Based on parallel computing and VEGA Grid environment, construct Grid system in Linux servers. Using Grid to build "economic-water-ecological" modeling, simulation and forecasting environment which is needed in Heihe river basin. Above the 2D graph foundation, using Java3D and VisAD to achieve 3D landscape shape simulation and forecast in Heihe river basin. Joins the SQL interfaces in the SME, quotes Chinese Scientific Databases which including quite complete data sets, and then, it afford an effective method for numerous models.

5 Acknowledgment

This work is supported by the Incubation Foundation for Special Disciplines of NSFC (National Science Foundation of China) project (grant number: J0130085), the Knowledge Innovation Program of the Cold and Arid Regions Environmental and Engineering Research Institute, Chinese Academy

of Sciences (grant number: CACX2003102)and the Knowledge Innovation important Program of Chinese Academy of Sciences(grant number: NF105-SDB-1-21)

References

[1] C Guodong, Aug. 2002, *Study on the Sustainable Development in Heihe River Watershed from the View of Ecological Economics*, JOURNAL OF GLACIOLOGY AND GEOCRYOLOGY, Vol. 24 No. 4.

[2] L.C.Deng, H.J.Yan, Z.H.Shang, December, 1996, *Scientific computation and astronomical image processing on Pentium/Linux platform*, Publications of Purple mountain observatory,Vol.15 No.4.

[3] L.X.XiaL.R.Xia, 2006, *Design of Wireless Sensors Network*, Instrumentation and Measurement,Vol.25 No.4.

[4] Thomas Maxwell, Robert Costanza, 1997, *An Open Geographic Modeling Environment*, Simulation Journal,68(3):175-185.

[5] Tom Maxwell, Robert Costanza, 1997, *A Language for Modular Spatio-Temporal Simulation*, Ecological Modeling,103:105-113.

The Implementation of the Forest Fire Monitor System Based on Embedded Linux

Yu-Rong Sun[a], Gui Zhang[a], Ming Zhang[b],Hui-Hua Huang[a]
[a]College of science, Central South University of Forestry and Technology
Changsha 412006, PR China
[b]bThomson Broadband R&D(Beijing) Co.,Ltd
Beijing 100085, PR China
sunyurong@163.com

Abstract

The forest fires are caused not only inevitably but also accidentally. It is useful to implement dynamically monitoring for forest fire for effectively controlling firing disaster and making use of forest fire. Embedded system has been used widely in remote monitor fields in recent years. The basic monitoring system discussed in this article is composed of ColdFire kernel processor with high-capability and low-consumption, and the open source embedded Linux. By extending RS232 interface and modules, the system builds the fire auto-monitoring platform and provides the system structure and software implementation. The article also discussed the key technologies are given.

1 Introduction

Forest is one of the indispensability resources for human survival and social development. But because of some uncontrolled behaviors in social activity and abnormal natural factor, forest fires are caused time to time. This has caused huge damage to our life property, earth resources and zoology environment. Forest fire auto monitoring and commanding system is designed for two main problem: one is how to locate the fire position earlier, the other is how to realize the management quickly and efficiently to fight the fire in time [1]. Obviously the method of monitoring the fire is very critical. With the development of science and technology, the monitoring method is renovated continually: from manual work to automation, and network remote monitor by using the intelligent equipments, which improve the level of fire-monitoring continuously. The embedded system is focused on application, based on computer technology, it can be tailored in HW & SW, and it is suitable for the special requirement, reliability, cost, cubage and consumption of an application system. Importing the OS into embedded system, it is easy to implement different functions by programming. And it is helpful to strengthen the agility and control of the system.. Furthermore embedded system can link those serial devices without Ethernet interface and other intelligent equipment conveniently into the network. It can also realize the data collection and the real-time monitoring of scene, the equipment configuration and the upgrade of the system software. Embedded Linux is becoming a widely useful embedded OS because of its abundant utility supporting software and free of charge. uClinux is an excellent network OS designed for embedded system [2]. The system's purpose discussed in this article is to realize the auto-monitor and alarm of forest fire, based on uClinux.

2 The brief introduction of embedded uClinux

uClinux is a branch of standard Linux. It contains common API, kernel smaller than 512K, and some relative tools. It is designed to use in micro-control field. It has strong ability of porting, tailoring, and stable performance. It also has some real-time management abilities. No MMU, supporting many file systems such as NFS (Network File System), ext2, MS-DOS and FAT16/32, and so on. Also including a complete TCP/IP protocol stack, and supporting many other network protocols, convenient to us-

ing network and developing network application programs. It provides a common application lib which is useful to shorten the development periods and improve the porting ability.

3 System design

3.1 system design target

This design target is to make the management person find out the scene of forest, high up the management level, and ensure the security. The main function is to monitor the forest and relative peripheral equipment 24hours continuously and give alarms. This requires a high reliable system.

3.2 system architecture

This system adopts the mode of browser/server (Figure.1). It uses the embedded system which is set in the monitored environment monitoring as a server, and the computer that run monitor software as client. They can communicate and duplex transmit monitoring signals through network (LAN or WAN).

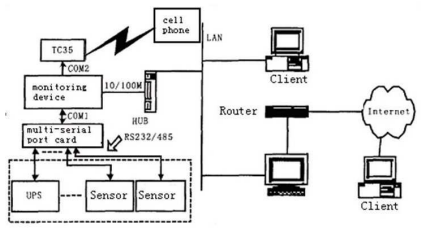


FIGURE 1: *system framework*

Embedded system adopts some light-duty equipments network (such as RS232, RS485, CAN Bus and so on) and link the peripheral equipment and some other intelligent equipments such as smoke-sensor, temperature-sensor, and humidity-sensor etc. together.. Also includes some signal collection modules like audio and video signals, then link them to client and database server through network equipments (router, switch). The client visit the monitoring center's Web page by browser software, and enter the monitoring page of Web server system, get the running state of relative equipments and set the parameter of these devices, also send the command to the embedded system(monitor-end) by this software. The monitoring terminal is placed in the monitored environment. The processor converts all of the collected signals into data in memory, and then converts information into TCP/IP protocol data package, then sent them to the user's computer via network, therefore it realized the transmission of remote information. It has solved the problems of embedded system linking to network. Also it manages several embedded systems in one time, therefore it enhances the integration of network framework, intelligence, information resource's browsing.

3.3 The hardware architecture of the embedded system

This system adopts the ColdFire embedded processor MCF5272 chip produced by Motorola which integrate ample common modules in SIM unit inside. Such as10/100M fast Ethernet controller, USB1.1 interface, 2 synchronal serial interface, 4 16-bit timers, 3 channels PMW modules and can connect to some common peripheral equipments (such as SDRAM,ISDN transceiver) perfectly, consequently predigest the design of periphery circuit, decrease the products' cost, cubage, consumption, and the difficulty of design at the same time. The system memory consists of three parts. It integrated 4K SRAM inside and extended FLASH and SDRAM. The capability of FLASH is 16bit x 1M, AMD29LV is adopted, its chip-select signal is CS0 and it is used to save vector table, uClinux kernel image and ROMFS file system. SDRAM is adopted 2 pieces 16bit x 4M of HY series, together formed 16M ,32bit system main storage , SDRAM's Chip-select signal is CS7.This processors support background debug mode(BDM).It provides the dispatching of hardware's low level. Here we use the 26-pin bnc to meet the BDM emulator connector recommended by Motorola, and to realize the code downloading and debugging [3] and the formed HW architecture graph is [4]

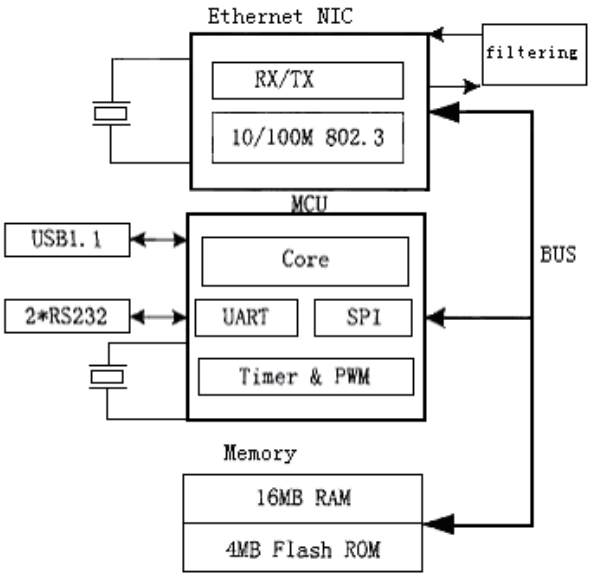


FIGURE 2: *Hardware architecture*

3.4 The system software design

3.4.1 The client software

This system adopts B/S mode to realize fire monitoring, install the browser software on the client only, the system monitoring can be achieved by internet/intranet.

3.4.2 The monitor-end software design

(1)the porting of the embedded uClinux system
The system adopts Linux2.4.20 kernel, cut out the corresponding kernel according to the needs, and develop corresponding equipment driver programs. Such as USB interface driver, it is mainly used to transmit the real time image information of the monitored environment. If there are higher requirements to real time performance, you need to load real time kernel RTAIReal-Time Application Interfacemodule[4, 5, 6]. Because the peripheral equipments and intelligent equipments are linked through serial interface with embedded system, a serial communication control module [2] should be added when the compiled kernel is transplanted.

(2)the network communication module design
Use socket to realize the network communication. This method of data transmission is a kind of special I/O. Socket is a file descriptor which is generated by calling the system function. Through this descriptor you can build the network connection and realize data transmission. The key design of this module is to use the select method to solve the blocking problem of Communication, and effectively save the system resource to make the system more effective. [7]

(3)Implementing CGI in C language
This system is a monitor system based on B/S mode. Because the Web browser can't understand the inside operation of embedded gateway system. So the CGI technology is used to realize the communication between server and client. when a user sent a request through browser to Web server, the server guard process start up the corresponding CGI program, convert the request into a format that server can identify, in the end convert the result into format Web that browser can identify, and sent them back to the client as the HTTP reply.

(4)The implementation of Web Server [8, 9]
The server sends, receives and deals with the data through serial interface, then sends the data to Web Server child-process. The child process displays the data on webpage and download it to customer. The child process also receives the command of setting parameters, then upload the settings to remote client (Figure.3).

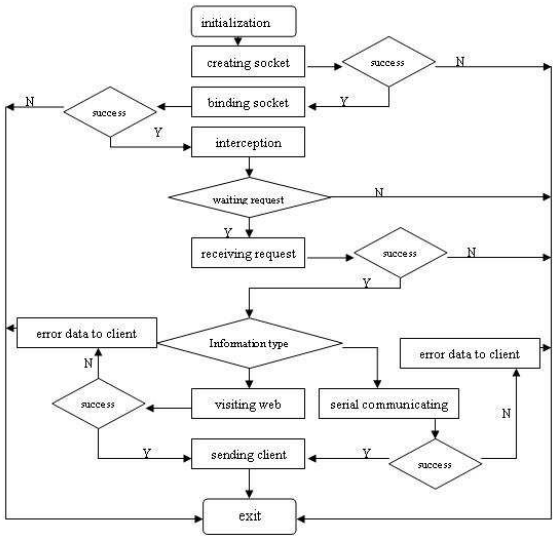


FIGURE 3: *The software implements flow*

Because of integrated the embedded web function, Monitoring device collects the data sent by I/O modules through RS232 bus. After processed inside, the user can visit these data through Internet/Intranet. By contrast, user can configure and control the output to I/O module through network. The basic function of the completed embedded system:
(1)Data communication, collection. Transmit the control command. Configuration information downloads.
(2)Manage the real time/history data. Check the alarm condition. Store the user configuration information
(3)Provide the Web visit function .Query the equipment data. Control the scenes equipments. Function ordered.

4 Key technology in system design

- This system is used for monitoring fire, so the working environment may be very bad. Such as the high-temperature, low-temperature, and humidity. Need to select the logical hardware and corresponding protection measures . Furthermore, the uClinux OS have not the MMU module, Adopts real-memory management policy, so it can not protect the memory space.

The memory management is dispatched by developer manually. Any address error will cause the program Corruption while running, even more then hang up of the system. Only repeat testing could ensure the system's .haleness and safety.

- The software divided the task into real time task such as data collection and non-real time task such as data storage. They are administered and dispatched by OS separately. The communication between them is realized by sharing memory.
- In embedded monitoring system, there are frequent inputs and outputs. In order to improve the performance and throughput. Multithread technology can be used to deal with multi-I/O operations. uCLibc and Posix Thread Support is selected when compiling the kernel of uClinux. By this way the application program can use corresponding threadlib. Because the running space is shared by many threads, the costs of the system can be decreased and the efficiency can be improved.
- Use DMA to realize the high-speed data transmission [10].

5 Conclusions

This system has a high agility, and it is easy to extend, it is suitable for difference situation and requirements by configuring the system's HW&SW. and this system has been used in Guangzhou forest fire prevention system, the effect is very good.

References

[1] Gui Zhang, 2004, Study *on forest fire dynamic monitoring in GuangZhou City[D]*, Changsha: Central South University of Forestry and Technology

[2] Si-Yi Zhou, 2002, *The design & application of embedded Linux [M]*, Beijing: Tsinghua University Press

[3] 2002.3. *Motorola.MCF5272 ColdFire Integrated Microprocessor User's Manual.*

[4] Jia-Fu Wan, 2003, *The porting of embedded Linux and its application research in network measurement & control. [D]*, Guangzhou: Guangdong University of Technology

[5] Zhen-Rong Liu, 2004, *The detail explanation of embedded Linux application's development [M]*, Beijing: China Machine Press

[6] Xian-Ying Huang, Chao-Hui Xiao, Yuan Chen, 2004, *Software Development for Embedded System and its Applications [J]*, Techniques of Automation and Applications

[7] Jian-Guo Zhou, Pu-Liu Yan, Cheng-Cheng Guo, 2002, *The Study and Implementation of Client/Server's Asynchronous Communication on Linux [J]*, Application Research of Computers

[8] SSV Software Systems, 2002, *Web server for embedded webservers[EB/OL]*, http://www.Dilnetpc.com

[9] ZiLOG Inc, 2002, *Introduction to embedded webservers [EB/OL]*, http://www.Zilog.com/docs/ez80

[10] Wei-Guo Wang, 2004, *The application research of embedded Linux in high speed data collection system[D]*, Xi'an: Northwestern Polytechnical University

RTLinux Ethernet Device Drivers

Florian Bruckner
OpenTech EDV-Research GmbH [1]
Lichtenstein Str 31, 2130 Mistelbach, AUSTRIA
florian.bruckner@aon.at

Abstract

This paper is about the current development of REDD, the Realtime Ethernet Device Drivers. First of all the previous state of the original REDD driver will be described, which was the base for our work. Then we will explain the most important changes we did during our development, like a common POSIX interface, multidevice support, `/proc` interface, and the error handling mechanism. The next step will be to introduce some examples that have been developed concurrently with the drivers. Our intention is to offer some easy to use tools to test the drivers functionality, like a ping request/reply or a simple benchmark tool. The last part will deal with future development of the driver itself, and of cause with the possibilities it offers. Examples are global time, redundant links, distributed shared memory.

1 Introduction

1.1 Brief History

REDD is a project started in 2002 and maintained by Sergio Prez Alcaiz [2]. The initial development of real-time ethernet for RTLinux/GPL was derived from the lw-IP [3] project with drivers merged from etherboot [4] and was released as RTL-LW-IP in the framework of the OCERA project [5]. Aside from the etherboot drivers not targeting real-time and there being some serious problems with lw-IP, it seemed preferable to depend future development on the main stream Linux kernel drivers. One of the main problems with lw-IP was its memory management, which was resolved in REDD by utilizing TLSF [6]. Secondly the code-base for lw-IP was large, most of which was not suitable for RT-applications, and thus neither well maintainable nor well integratable into the RTLinux/GPL effort. Departing from the shortcommings of lw-IP required a complete reimplementation which was REDD.

1.2 What is REDD

REDD consists of a set of ethernet device drivers suitable for RTLinux and a common lowlevel interface to access them in an RT-safe way. Currently rtl8139, e100 and 3c59x cards are supported and a VIA Rhine drive is being developed. The drivers are originally based on the standard Linux drivers, modified to fit the RT-constraints, i.e. some non-deterministic performance-improvement features are turned off in order to guarantee realtime behaviour. Examples of such features are

- early interrupts - which are raised before the whole packet is actually transfered to the input buffer.
- multiple events handled per interrupt - Modern NICs also allow to handle more than one packet within one interrupt handler, which would dramatically decrease realtime performance.

Another important difference to Linux drivers is error handling. Realtime applications cannot easily retransmit a packet without introducing inaceptably high worst case latency or ignore a link-state-change. Thus there must be a possibility to detect these errors in a timely manner and allow a clean shutdown of the system to reach a safe state. In order to make ethernet realtime capable there are also some restrictions imposed on the topology of the network. The simplest case, preventing collisions, is to use point-to-point connections only - which can be achived by the use of cross-over cables or hubs. Another solution would be to use a Time Triggert approach in a multinode network. A variant of which, namely TDM, is implemented in RT-Net [7].

2 Current State of Development

Currently a layered module set is used, offering a common interface via the `redd.o` module to the RT-application layer based on a POSIX open/read/write/etc calls. `redd.o` registers an REDD major device, which allows access via standard device mechanisms. Additionally it offers a register function for the device drivers. The device driver has to offer some common access functions like a send, receive function. During the registration of the REDD device a device structure is created. The minor number of the device allows to distinguish all REDD devices.

This design allows the usage of multiple devices including a loopback device to test your applications. Currently loopback only supports point-to-point connections, but it is planned to implement a switch/hub mechanism to simulate a virtual network. It should be possible to configure a logical network as it will be used in real world (e.g. 3 nodes connected by a hub). Of course the timing of the real network cannot be simulated, but it can be very helpful for testing. Additionally there should be an error injection mechanism to simulate some hardware errors like link-change, wrong checksums, etc . This should allow better testing of your application under some special error conditions which would otherwise be difficult to create and support device validation efforts similar to netemu.

Another possibility that this design offers is to implement drivers that need access to more than one NIC. Examples for such driver would be redundent links, bonding drivers, routing threads, etc.

3 Problems during development

One problem that surfaced during the development of REDD is the error handling strategy. Nearly all errors that can be detected by the NIC are serious errors for realtime applications and will lead to a shutdown (in case of single link systems). Exceptions are errors that can be recovered from within the driver code (e.g. if a transmit FIFO underrun occurs, the packet can be correctly transfered but then the tx-threshold is set to a higher level) or from within the protocol layer.

The main problem are the class of connection independant or general errors that do not belong to a specfic packet/connection and thus show a bad detectability (i.e. won't be noticed until the device requests a transmission or receive), usage of heart-beat mechanisms can resolve these issues with an acceptable overhead. Additionally it is problematic that during the, de-facto mandatory non-blocking send procedure, error information becomes available only when that packet has been sent by the NIC and not when the write call returns, thus delaying recovery or safe shutdown. The current implementations uses a global (per device) errno variable that stores the current error values. Every POSIX call checks this variable and returns the errors code if something went wrong. The error may not correspond with the actual packet. One possibility to directly read the error values would be to use IOCTLs.

4 Examples and benchmarks

During the development of the driver we also created some example applications for testing reasons. First example was a simple ping reply and ping request. The ping request is able to create a ARP reply for address resolution and if it gets an ICMP request it simple swaps sender and receiver address and changes the ICMP code to ICMP-reply. The ping request first hast to send an ARP request and then creates ICMP-request packets of different sizes. Additionally the roundtrip time of the packet is measured.

Another example is the test_bandwidth example that can be used to measure the roundtrip time as well as the throughput of packets of different size. It simply sends packets with a sequence number and a certain size to the receiver. For one measurement `NUM_ITERATIONS` packets are sent with the sequence number incremented after each packet. If the receiver get a packet with sequence number equal zero is replies an acknowledge packet. The receiver also checks that no sequence number is missing. The sender repeats this procedure `NUM_MEASUREMENTS` times and always takes timestamps before the first send and after the reception of the acknowledge. If `NUM_ITERATIONS` is set to 1 the difference between these timestamps is the roundtrip time of the packet, if it is set to a very high value you get the inverse of the maximum throughput.

First benchmarks with two equivalent 8139too 100Mbit/s NICS connected by a crossover cable show that the roundtrip time start at 50us for small packets (60 Bytes) and rises up to 200us for MTU-sized packets (1500 Bytes). The throughput measurement shows a nearly linear connection between packet size and time. For MTU-sized packets it takes about 120us with gives us a bandwidth of about 95 MBits/s which is a very good values. These a values of without any systemload. If the system running lmbench or netpipe during the benchmarks the

roundtrip times got worse value but the throughput values almost stayed the same. A big problem was that lmbench and netpipe do not create the same load over time, which makes it hard to find some real worst case times for each packet size.

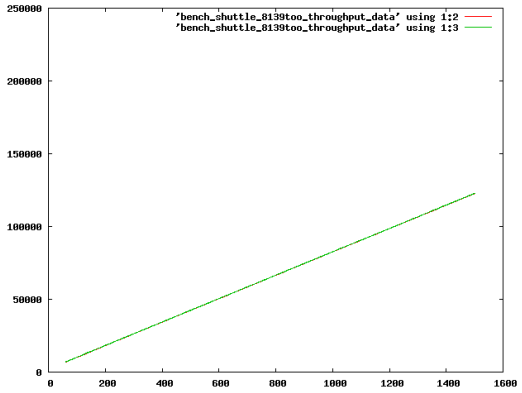


FIGURE 1: *Throughput*

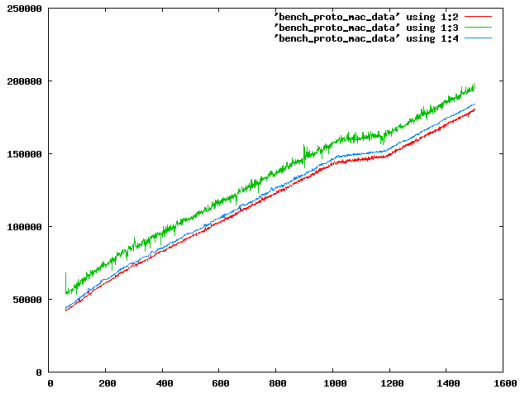


FIGURE 2: *Round-Trip Time*

5 Conclusion and possibilities for future development

Currently REDD offers a low level interface to some ethernet cards. It uses a read/write interface instead of a heavy socket interface. This keeps the interface overhead at the minimum but leads to some restrictions in usage. In contrast to a general purpose network stack it is not possible to layer many different protocols on top of this interface. But for realtime behaviour this would not be a good idea anyway. The REDD interface will allow different protocols but the access to the NIC will be restricted to one protocol at a time.

Examples for such protocols are distributed shared memory implementations, global time, time triggered protocols, MPI RT, and so on. The next steps in REDD development will be to automatically add a MAC header with the local source address and a multicast destination address. Additionally an interface should be added that allows to bind several physical links to one logical link and provide redundancy.

References

[1] Opentech, *OpenTech - RTLinux Support in Europe*, 2006-09-19, `http://www.opentech.at/`

[2] Sourceforge, *REDD: RTLinux Ethernet Device Drivers*, 2006-09-19, `http://redd.sourceforge.net/`

[3] Sourceforge, *RTLinux Lightweight TCP/IP Stack*, 2006-09-19, `http://rtl-lwip.sourceforge.net/`

[4] Sourceforge, *EtherBoot/GPXE*, 2006-09-19, `http://etherboot.sourceforge.net/`

[5] Ocera, *Open Components for Embedded Real-time Applications*, 2006-09-19, `http://www.ocera.org/`

[6] TLSF, *Real-Time Dynamic Memory Allocation*, 2006-09-19, `http://rtportal.upv.es/rtmalloc/`

[7] RTnet, *Hard Real-Time Networking for Real-Time Linux*, 2006-09-19, `http://www.rtnet.org/`

Operating System Modifications for User-Oriented Addressing Model

Dong Zhou and Taoyu Li
Department of Electronic Engineering, Tsinghua University
Tsinghua University, Beijing, 100084, P R China
zhoud@mails.tsinghua.edu.cn, overnitary@gmail.com

Maoke Chen and Xing Li
Network Research Center, Tsinghua University
Tsinghua University, Beijing, 100084, P R China
{maoke,xing}@cernet.edu.cn

Abstract

In both IPv4 and IPv6, IP address is assigned to interface and shared by different users on a same host. This model of addressing is not suitable for future Internet-based computing, where user-oriented accounting, behavior tracing, multi-homing and independent port usage must be supported. Because IPv6 provides a big enough space that everyone in the world can own several addresses, it is believed that a new, user-oriented IP addressing model will appear in the near future.

On the other hand, network protocol stacks in current operating systems are designed in the framework of interface-oriented addressing model. To specify a dedicated address to each individual user, the kernel must be modified so that one user can use only its own IP address, either for source or destination of a packet. Secondly, both user and IP address administration tools must contain a map from user ID to a group of address that the user having permission to apply them. Furthermore, the behavior of application programming interface must be modified in order that each user-mode network program binds address properly. Issues of compatibility and security are also discussed.

1 Introduction

Current Internet addressing model is host- or interface-oriented. Both IPv4 and IPv6 protocol specifications assume that IP addresses are assigned to network interfaces [1] [2] [3]. One host may have one or more interfaces and each of them can be assigned with several different addresses. When a user in a system is sending a packet to a certain destination, any interface address of the destination can be employed, and the source address is specified with the first one (in IPv4) or following a certain set of conventions (in IPv6, as described in RFC3484 [4]).

New applications and techniques raise the problem of changing the addressing model of the Internet. The Source Address Validation Architecture (SAVA) [5] aims at a new solution of global sender identification, for the purpose of both security and accounting, and accordingly requires that IP addresses can identify end users instead of interfaces that shared by several persons. The Site Multi-homing mechanism (shim6)[6] is trying to separate the two roles of the IP address: identifier and locator, in order to get the flexibility of changing service providers on demand of users. The PlanetLab system [7] opens a new computation model that multiple users share multiple personal computers over the Internet, and interface-based addressing model doesn't meet the requirement of independent port utilization for each user.

Therefore, a new addressing model is needed to support user-oriented accounting, behavior tracing, multi-homing and independent port space. These needs can be easily satisfied if every user has his own address and this is possible in IPv6, which provides enough addressing capability. In this paper, focusing on the implementation of the user-oriented IPv6 addressing model, we discuss the modifications that

are needed in the Linux operating system. In the modified operating system, a user can only use the address assigned to it as either sender or receiver of IPv6 packets. Address assigned to other users can be applied only by the super-user (root) of the system.

There are several strategies for realizing the modifications. It is possible to put protocol stack into user mode based on a micro-kernel, or to modify the protocol stack within kernel itself, or to utilize some hook mechanisms like netfilter or libpcap [8]. In this paper, we take the second way as the solution and design a new kernel with augmented data structures and updated network stack routines.

The rest part of paper is organized as follows. The user-address mapping facilities and corresponding administration tools are designed in Section 2. Kernel and API modifications are presented in Section 3. Then, in section 4, issues of compatibility and security are also discussed, followed by the concluding remarks.

2 User-Address Mapping

The mapping from user-id space to the space of IP address is the key component in the implementation of the user-oriented addressing model. A dedicated data structure for the map must be defined. Throughout this paper, we discuss either IPv4 or IPv6 case but it is suggested that the data structures and functions are separately implemented for IPv4 and IPv6. Commonly we don't specify the "v4" or "v6" after the word "IP" unless it is really necessary.

2.1 Data Structure of User-IP Mapping Table

Because it is not useful in practice to get all the users that an IP address is assigned to, the mapping from user ID to IP address is the only data structure that is necessary. The user-address mapping table contains all the IP addresses accessible for each user. In the operating system, this table is stored in a plain-text configuration file under the */etc* directory.

The configuration file is written in the ASCII text mode, where a set of available IP address is defined for each user ID. A typical file */etc/uamap6.conf* may be written as follows.

```
......
401;2001:da8:200:9002:250:4ff:fe98:6291,\
    2001:da8:200:9002::a;
402;;
403;2001:da8:200:9002:250:4ff:fe98:3001;
1001;2001:250:1214:80::8ee8; ......
```

In each line of the file, the first term before the semicolon indicates the uid of the user, while the term after it contains all the IP addresses the user has privilege to access, separated with comma. The sequence of the lines are sorted by the uid. A uid of an inexistent user or a user who doesn't need network access can be noted with a blank line or just excluded from the file.

When the kernel starts, this configuration file is loaded into memory and stays there until the system halts. The data structure in the memory where user-IP table is stored is described in FIGURE 1.

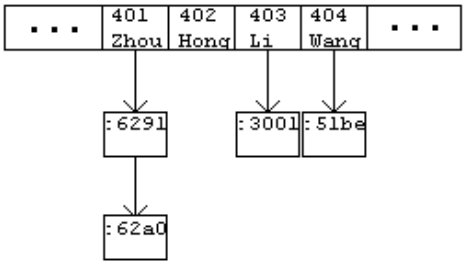


FIGURE 1: *User-IP Table In Memory*

As showed in FIGURE 1, the mapping table is implemented with an array of chain lists. Each chain list contains all IP addresses assigned to an individual user. Each uid corresponds to a certain element in the array. The data structure can be described by the following codes:

```
struct clip
{
    in6_addr ip;
    clip* n;
}
clip* uatable [MAXUID];
```

A set of functions are defined for manipulating the global variable of *uatable* and the configuration file */etc/uamap6.conf*. Their declarations are as follows.

```
int addip(unsigned int uid, in6_addr* ip);
int delip(unsigned int uid, in6_addr* ip);
int chkip(unsigned int uid, in6_addr* ip);
clip* listip(unsigned int uid);
int rmusr(unsigned int uid);
int tb2f();
int f2tb();
```

addip() **and** *delip()*

These two functions carry out the function that add or delete a certain IP address to the mapping of a certain user. They both take the uid of the user and the IP address to be added or deleted as the parameter and return 0 if success or other integer if error occur.

chkip()

chkip() checks if a certain IP address and a certain user have been mapped in the table. The uid of the user and the IP address are transferred as parameter. Returns 1 if the user and the address have been mapped, 0 if not, and negatives on errors.

listip()

listip() lists out a chain list containing all the IP address mapping with a certain user. The only parameter indicates the uid of the user. Returns the head pointer of the chain list if success, or NULL if failed.

rmusr()

The function *rmusr*() is usually called when a user is deleted from the system. This function removes all the address mapping to a certain user indicated by the parameter as uid. Returns 0 if success or negatives on errors.

tb2f() **and** *f2tb()*

These two functions preserve the synchronism between configuration file */etc/uamap6.conf* and the data structure in the memory,. *f2tb()* reads the mapping table from configuration file out to the memory, and*tb2f()* writes the table back to the file. Usually, *f2tb()* only needs to be called once while the system starts. However, *tb2f()* must be called after each time when the data structure in the memory is updated by *addip()*, *delip()*, or *rmusr()*. Returns 0 if success or negatives on errors.

With this data structure, the former functions can be preformed in a most rapid way. Most functions first locate the user's chain list of mapped IP addresses by taking uid as the tag of the array, and then carry out the work within a simple chain array. This design assures the efficiency of the system.

2.2 User-IP Mapping Table Administration

Since new configuration file has been applied, new administration programs are in demand. Some of the existing administration programs, such as *userdel*, must also be modified. The user-IP mapping table will be modified in several conditions, such as the first time a user want to connect to network, the time a new IP address is assigned to a user, or the time a user is removed from the system. At these time, there must be certain administration tools to maintain the mapping table. It must be assured that only the super user can modify the mapping table. When modifying user-IP mapping table, first the data structure in memory is renewed, then the new data is written back into the configuration file.

To reduce the change of system administration tools, the general tools to add a new user, such as *useradd* program, are not modified. Instead, a specialized tool , *uamap* , is developed to maintain the mapping table. It can add, list, or delete each user's IP address. That means, when a new user account is created, it does not has the privilege to access any of the IP addresses until *uamap* is executed manually. However, when the user is deleted, its IP address table must be cleared, or the mapping table will result into chaos after new users are added. So the tool *userdel* must be modified to call *uamap* automatically before actually delete the user. Besides, when adding or deleting users, the administrator can also add or delete term from the mapping table using *uamap* at any necessary time.

3 Kernel and API Modification

To specify a dedicated address to each individual user, a checking approach must to be added to check if a user has the privilege to send or receive data from an address. If this address has been assigned to this user, the operation will be allowed, otherwise the request will be refused. Furthermore, if the user use IN6ADDR_ANY_INIT instead of specifying a certain address, we should limit the user only to use the addresses assigned to it instead of all available addresses. According to the discussion in Section 2, our data structure is efficient enough to do package-level checking without obviously raising time-cost.

3.1 Location of Modifications

Ideally, the checking can be placed at any step between the application(application layer) and the device(physical layer). But actually, only some special

steps make sense, while the most others are not suitable.

Since we want to do an IP address checking, it is impossible to place the checking under IP layer. Furthermore, in order to do a user-address mapping checking, it is necessary to know the user-id. Therefore it is unadvisable to place the checking in the TCP/IP protocol stack because in the stack it is not convenient, and sometimes impossible to get user information. For example, when an interface receives a UDP packet, the protocol stack will deal with this packet automatically, and finally put this packet in socket receive queue, waiting for socket-level receive. In this whole process, protocol stack knows only which address this packet should be sent to, but it has no information about which user will read this packet. Therefore it is impossible to insert the checking into this process. So the checking should be higher than TCP/IP stack(transport layer).

On the other hand, the checking should be under applications since we can not modify the user applications. Therefore the checking is supposed to be placed between transport layer and applications, that is, socket layer. Socket 'is the interface between the transport layer protocols in the TCP/IP stack and all protocols above. ... All data and control functions for the TCP/IP stack pass through the socket interface'. In Linux, 'the socket interface is the only way that applications make use of the TCP/IP suite of protocols'.[9] Therefore the socket layer is an ideal place to set the modification, as in FIGURE 2.

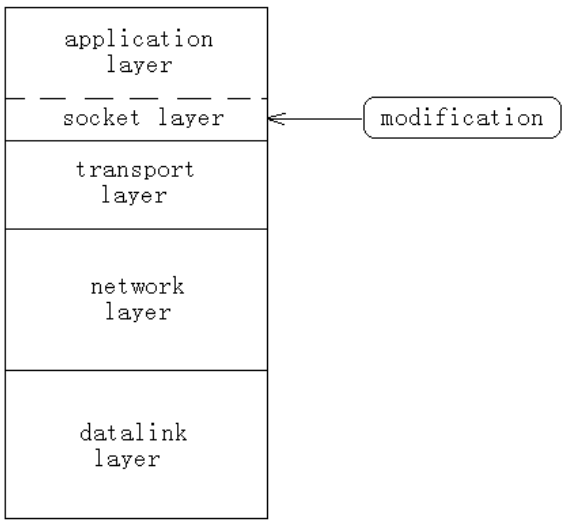


FIGURE 2: *Location of Modifications*

3.2 Some Modification Details

After locating the checking at socket layer, we must still consider where of the socket layer should we do this check. Since users may do different operations on a socket, to check all these operations or just part of them is a problem to consider.

All possible operation on a socket is defined in the data structure of *socket->proto_ops* as in Table 1.

Type	Operations	Modification Needed
int *	release	
int *	bind	√
int *	connect	√
int *	socketpair	
int *	accept	
int *	getname	
unsigned int *	poll	
int *	ioctl	
int *	listen	
int *	shutdown	
int *	setsockopt	
int *	getsockopt	
int *	sendmsg	√
int *	recvmsg	√
int *	mmap	
ssize_t *	sendpage	

TABLE 1: *Socket Operations*

Among these 16 operations, only those related to local address, including *bind*, *connect*, *recvmsg* and *sendmsg*, should be checked. The others does nothing to do with the address, so there is no need to check them.

3.2.1 *recvmsg*

When the application calls any of the read functions on an open socket, the socket layer calls the function pointed to by the *recvmsg* field in the prot structure. For example, for the SOCK_DGRAM type sockets, the receiving function is *udp_recvmsg*, in the file linux/net/ipv4/udp.c. This function dequeues packets from the socket's receive queue by calling the generic datagram receive queue function, *skb_recv_datagram*, and then copies packets from kernel space to user space. Therefore we can insert the checking call before the *skb_recv_datagram*, in order to prevent the user getting the packets which do not belong to him. Same modification should be made to *tcp_rcvmsg* for SOCK_STREAM type sockets.

3.2.2 *bind*

To prevent the user binding the socket to an address which does not belong to it, the checking should be appended into the *bind* function. This modification does not only serve for the *bind* operation, but also infect the *sendmsg* and *connect* operation through *autobind* mechanism.

On the other hand, if the user tries to bind to IN6ADDR_ANY, the kernel will read from the mapping table to get the list of the addresses accessible for the user, then try to bind to these addresses instead of to all the addresses available in the host.

3.2.3 *sendmsg*&*connect*

When user tries to send messege or do connect operation while current socket has not been bound to a special address, the kernel will execute *autobind* to bind the socket to IN6ADDR_ANY. This means we can reuse the checking in *bind* and don't need to modify the two operations.

Therefore, we need to modify the *recvmsg* and *bind* function to add the checking module. These two function are both system calls in the kernel. The API is not modified.

4 Compatibility & Security

Since we just modify functions in the Linux kernel and don't change the API, most applications can work well with new kernel without any modification or re-compilation. Therefore the proposed scheme of implementation has good compatibility. Calling *listip*() will help an application to get the addresses assigned to a certain user, if it need these addresses as running parameters.

We supposed to do checking in functions of socket layer. Network programs without involving socket communication may escape the checking mechanism and get the privilege of using addresses not assigned to it but belonging to another. In such case the checking mechanism is bypassed. This security issue is still in research.

5 Conclusions

User-oriented IPv6 address architecture will be a new revolution for the next generation Internet. Implementing such an addressing model in a operating system is challenged by the issues of efficiency, compatibility and security. The proposed configuration file and kernel data structure for the user-oriented addressing model as well as the routines manipulating them are working in a way as efficient as possible. Issues of compatibility and security are briefly discussed. It is believed that realization in Linux operating systems will encourage the popularity of the user-oriented addressing architecture.

References

[1] DARPA Internet Program, 1981, *Internet Protocol* , IETF RFC 791.

[2] S. Deering, and R. Hinden, 1998, *Internet Protocol, Version 6 (IPv6) Specification*, IETF RFC 2460.

[3] R. Hinden, 1998, *IP Version 6 Addressing Architecture*, IETF RFC 2373.

[4] R. Draves, 2003, *Default Address Selection for Internet Protocol version 6 (IPv6)*, IETF RFC 3484.

[5] Tsinghua University, 2003, *Source Address Verification Architecture Framework, draft-sava-framework-00*, Internet-draft, Work in progress.

[6] J. Abley, B. Black, and V. Gill, 2003, *Goals for IPv6 Site-Multihoming Architectures*, IETF RFC 3582.

[7] Larry Peterson, Steve Muir, Timothy Roscoe, and Aaron Klingaman, 2006, *PlanetLab Architecture: An Overview*, http://www.planet-lab.org/PDN/PDN-06-031.

[8] V. Jacobson and S. McCanne, *libpcap: Packet capture library*, ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[9] Thomas F Herbert, 2004, *The Linux TCP/IP Stack: Networking for embedded systems*, Charles River Media, ISBN:1584502843.

Real-Time P2P Interactive Game Playing Application: Under Linux Based on XCAST6

Dwijendra Kumar Das
Dept of computer Science, Asian Institute of Technology
Pathumthani,Bangkok,Thailand
dwijendra.kumar@gmail.com

Long Jiaoyan
Dept.of Computer Science, Lanzhou University of Technology
Gansu, Lanzhou, China
jylng@lut.cn

Abstract

Building real-time interactive P2P game playing applications in Linux posses many challenges and opens a wide area of research. There have been many implementations of remote interactive servers for game playing in some universities or companies providing collaborative access to the remote services. Most current systems that provide a collaborative remote game environment either use the (multi) Unicast technique or the Multicast technique to transmit data packets to the participants of the experiment group in the network. Both of these schemes have disadvantages and advantages. In this paper we implement architecture under Linux for real-time multiplayer game application based on XCAST especially for the cases where there exist numerous small collaborative groups. We had provided extension to the Linux kernel and XCAST and show that formation of numerous simultaneous groups, where each group would collaborate for a separate game, is possible. The system proposed is robust in consistently providing group formation and collaboration activities in real time, back-up route or priority queuing, and on time packet delivery with minimum delay in network in the presence of continuous node arrival and departure in the entire game playing procedure. We also show that for data packets of low size, the use of XCAST in the network layer decreases the stress at the sender in each group whereas due to the increased header size of the XCAST packets. Our implementation has shown significant improvements to meet the demands of some real-time game applications.

1 Introduction

Peer-to-peer (P2P) networks are a logical choice for supporting distributed multi-user virtual worlds as expensive servers are not required. We are currently focusing on Massively Multi-player Online Gaming (MMOG) applications. P2P architecture offers several advantages over centralized architectures for various applications, including MMOG. Decentralization with high scalability and no single point of failure forms the key advantage. We first looked at the common properties of P2P systems and for ways that these properties might be used to inform the design of a game-world, such that the design of the game-world will reflect the state of the nodes and network links which compose the peer-to-peer network [1], [2].

A MMOG application can be modeled as a set of interacting entities in a virtual world, with a large population of players. Players run an application on their PCs that allow them to interact with the entities and each other. Static entities, e.g., a picture hanging on a wall, can be cached at all users PCs and displayed whenever required. Dynamic entities, e.g., a rocket propelled grenade launched by a user, need to be continually updated on all relevant PCs. Dynamic entities can be user controlled, changing state as the result of user input, or as the result of pre-programmed game logic. Based upon these observations, we have developed P2P implementation architecture under Linux by integrating XCAST API [4], [5] and providing a P2P application support on

the network layer.

2 Background

Relevant work exists in the fields of distributed multiplayer gaming. This is discussed in sections 2.1 respectively.

2.1 P2P Game Support

The majority of approaches to supporting distributed, multiplayer gaming on P2P networks, such as and focus upon providing support for real-time communication and consistency management by compensating for the heterogeneous network technologies and local platforms used in the underlying P2P network [3]. Real Tournament [6] aims to support real-time gaming through the implementation of content filtering schemes on the underlying P2P network, similar to those implemented in the CAN overlay network [6]. In this model, filters of interest ensure that communication of game-state between nodes is bandwidth efficient and furthermore, that game-space is intelligently partitioned between nodes based upon their position in the game world. Solipsis [7] uses a P2P network to maintain a virtual world that is capable of handling very large numbers of players and game world objects. Solipsis does this using a P2P network whose architecture corresponds to a Gnutella-like network structure [8]. Solipsis optimizes this network by collocating peers which are close in the game-world on the P2P network, thus reducing the distance that messages must travel on the underlying P2P network. All the above approaches have the underlying disadvantage of scalability of the group and also as all the above explained architectures attempt to compensate for the heterogeneous network technologies and local platforms used in the underlying P2P network, the additional support for that introduces traffic which in turn increases the delay. Our XCAST6 [5] based architecture seeks to create a novel gaming experience as it creates multiple simultaneous groups of small size and integrate them under common application layer API.

3 Design of the Framework

3.1 P2P game architecture

The P2P game architecture is aimed towards the development of complex P2P multiplayer game applications where there are numerous groups collaborating for resources. All nodes desiring to be part of the group join a common P2P network as peers and search for other peers that have matching interest by announcing their respective interests to the network. There are various such resource discovery and object location techniques CAN, Chord, Pastry, and Tapestry. Pastry allows a large number of peers to be integrated into the P2P system with less routing state information per node and guarantees the location of an existing object in a logarithmic order of hops. All nodes desiring to be a part of the integrated group enter the Pastry ring (P2P network) as peers. Each peer uses the Secure Hash Algorithm (SHA)-hash of the group name it wants to join to calculate the key value for the group name. It then uses this key to tag a Register Message that it sends to the ring. The Pastry ring then automatically routes the Register Message to the node that is responsible for the key. As shown in Figure 2, the key K for group named Control System when tagged to the Register Message, the message gets routed to node X, which is currently responsible for key K. This node is also called the Rendezvous Point (RP) for the group. When two or more nodes send the Register Message for the same group, each of the Register messages gets routed to the same node that is the RP for the group. The RP in turn can inform all registering nodes about other nodes that have registered for the group. As a result, a fully connected mesh of members in the group is formed.

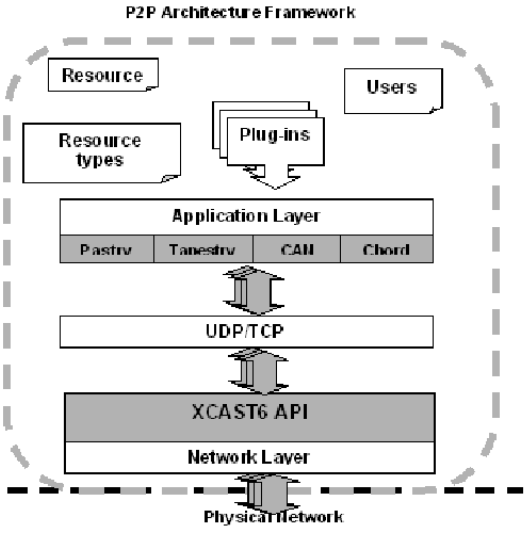


FIGURE 1: *Figure The Architecture of the Framework*

The following are the sequence of steps that occur for the formation of a single group:

Step 1: New node desiring to join a particular group sends a Register message to the ring which gets routed to the RP for that group. The Register message contains node specific details: IP address, port, role (Resource Peer, Resource Requester), and username.

Step 2: RP sends the New Member Arrival Information Message to the existing members of the group (if any). Hence, the existing members get information of the new node arrival.
Step 3: RP sends the Existing Members Information Message (if any other node has already registered for the group) to the new node. Hence, the new node discovers the existing members for the group.
The Pastry technique supports simultaneous routing of multiple messages in the ring. Therefore, when Register messages corresponding to different groups are simultaneously sent to the ring by different nodes, the Pastry API (Application Programming Interface) is capable of routing all the messages to respective RPs for the groups.When the registration for different groups and routing of the Register messages occur simultaneously in the Pastry ring, formation of multiple simultaneous groups is possible.

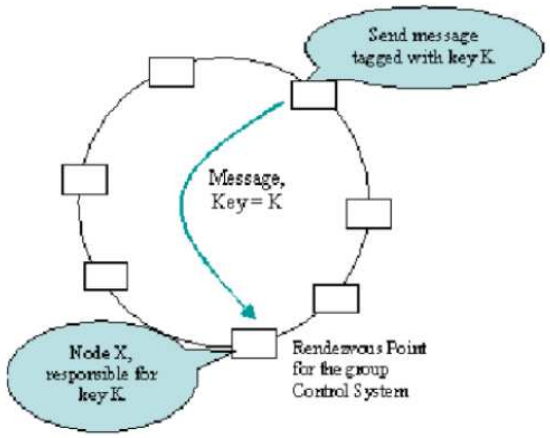


FIGURE 2: *Figure Routing of Register Message to RP*

3.2 Access Right Assignment

n each of the multiple simultaneous groups formed, any member of the group can have one of the following roles:
Resource Provider This is the node in the group that has the game specific file(s) or resource(s) interfaced to it. An analogy with the LINUX operating system would be considering the Resource Peer as the file system for LINUX. Just as the LINUX file system provides the access to users to one or more files, the Resource Peer provides access to one or more resources.
Resource Requester Players join the group for the purpose of playing games. Each player joining the group can issue requests for files and resources that the Resource Peer provides. A player can also voluntarily release the access right assignment he or she has been granted. An analogy with the LINUX operating system would be considering the player as a normal user of the LINUX file system.
The following are the sequence of steps that occur during the access right assignment to a member for any resource:
Step 1: The new member joining the group requests the resource list details from the Resource Peer.
Step 2: The Resource Peer sends the resource specific details to the requesting node using a Resource List Message. The details include the names of resource file, the possible access rights for each, and the current assignment of access rights to different users (if any) for each file.
Step 3: The member node desiring to gain access right for a file sends the Access Right Assignment Request to the Resource Peer for a specific resource. The request message contains the ID for the file and the access right the member desires to have.
Step 4: The sender side checks the validity of the request by comparing the request with the current access right details received from the Resource Peer. A Control Access Right Assignment Request is valid if there is no controller already for the resource file. An Observe Access Right Assignment Request (for a resource is valid if the member requesting the access is not already an observer for the resource. Provided that the request is valid, the request is sent to the Resource Peer.
Step 5: The Resource Peer again checks the validity of the request. This is necessary for the case when two or more members almost concurrently send the Control Access Right Assignment Request message for the same resource. In this case, the member whose request reaches the Resource Peer first will be granted the access right and the other member(s) will be denied Control access right. The Resource Peer updates its Access Control List (ACL) for the particular resource.
Step 6: All the existing members of the group are informed about the changes in the access right assignment, via an Access Right Change Information Message, for the resource such that the changes get reflected in every other member. Each receiving member then updates its information about the access right assignment details for the particular resource.

3.3 Robustness of the System

The following two points should be considered for achieving robustness in the P2P network:
Persistent Connectivity in the Ring on Node Arrivals and Departures: The persistent connectivity in the Pastry ring is maintained by the API by using periodic Keep Alive messages such that all the Pastry ring specific data like the routing tables,

neighborhood set, leaf set are updated with node arrivals and departures.

Persistence of Application Specific Data on Node Arrivals and Departures: For achieving the persistence in group specific data, the application specific data in each of the Rendezvous Points should be available on departure of the RP from the ring. This is achieved by the transfer of group registration information from the departing RP to the node that would be the next RP for the groups after the current RP departs. This solution works for the case of graceful departure of the RP, that is, when the RP informs the P2P network when departing. For the case of silent departure of the RP, for example, in case of electric surge, the solution adopted does not work. For this case, the robustness can be achieved by storing group register information for each group in multiple nodes. Specifically, they should be stored in the K-closest nodes for the key corresponding to the group name. In such a case, if it is assumed that all the K-closest nodes for any group key do not fail simultaneously, then the group registration information for the group remains persistent even with the silent departure of the current RP for the group.

4 Linux Implementation

The system has been implemented in Fedora core 4.0 platforms [9]. It uses FreePastry API (FreePastry 1.4.2) [11] that implement the Pastry technique in the application layer. In the network layer, the XCAST6 API (XCAST Release 0.2.1) [4] that implements the XCAST protocol is used. XCAST is used in the underlying layer in situations when a sender has to send the same information to more than one fs receiver. The FreePastry API is implemented in Java. Hence, Java (Java Version 1.4.2-p7) [9] has been used for the import of API functions into the application system. However, the XCAST6 API is implemented in C. Hence, the system uses the Java Native Interface to call C (gcc version 3.4.2) [5] functions from Java whenever XCAST has to be used in the network layer. Furthermore, the FreePastry API supports only IPv6 addresses currently and the XCAST6 API supports IPv6 addresses only. Hence, in the current implementation of the system uses both IPv4 and IPv6 addresses.

4.1 Multiple simultaneous groups formation based on interest

For the implementation of multiple simultaneous groups formation, the FreePastry API has been used. The API provides classes and functions to connect to the BootStrap node, thereby allowing the new node to join the existing node, or to start a new ring if the BootStrap node does not exist. In either case, the new node can successfully initialize its state in the virtual Pastry ring. The API also provides functions to calculate the SHA-hash of any group name that is provided, and to support automatic routing of any message tagged with a key to the node currently responsible for the key. For group formation, XCAST6 API is used in Step 2 when the RP has to send the New Member Arrival Information Message to more than one existing members of the group.

4.2 Role based access right assignment

In this phase, XCAST6 API is used in the network layer when the Resource Peer sends the updates of the access right assignment of a resource to all the group members.

4.3 Collaboration among group members

The simulation of possible collaboration among group members by sending text messages has been implemented using XCAST6 API for one to many communication and Unicast for one to one communication.

4.4 Exit from the group

This phase uses XCAST6 API for sending BYE messages to remaining group members when multiple group members are present and Unicast when there is only one other group member remaining other than the member that is leaving. The FreePastry API is again used to route the BYE Message to the RP when the member leaves. In case of the RP departing from the group, the functions provided by the API are used to find out the new node that would be responsible for the group information that the departing node is currently holding and to transfer the RP member table details for each such group.Each node in the system uses a linked list to store the details of the members that have registered for the group names for which it is the Rendezvous Point. Each element of the liked list contains details of the members for a single group name. The following table describes the format of each element of the RP Table:

gams ID	*Node ID List*	*IPv6 Address List*	*PortList*	*Role Name List*	*User Name List*

FIGURE 3: *Figure Rendezvous Point Information Table*

where, NodeIDList is a linked list containing the NodeIDs of all the nodes that have registered for the game with ID gameID.
IPv6AddressList is a linked list containing the IPv6 Addresses of all the nodes that have registered for the game with gameID.
PortList is a linked list containing the ports of the corresponding nodes that have registered for the gameID.
RoleNameList is a linked list containing the status values (Resource Peer, Resource Requester) corresponding members that have registered for the gameID.
UserNameList is a linked list containing the usernames of the corresponding members that have registered for the gameID.

The following are the formats of the messages that the system uses for various operations provided by the system:

ADDRESS 8 Bytes	*User Name 11 Bytes*	*IPv6 Address 41 Bytes*	*Port 6 Bytes*	*Role Name 8 Bytes*

FIGURE 4: *Figure Register Message*

Group Name Key (VAR)	*User Name (10Bytes)*	*IPv6 Address (40Bytes)*	*Port (5Bytes)*	*Role Name (8Bytes)*

FIGURE 5: *Figure New Member Arrival Information Message*

ADDNEW	User Name 1	IPv6 Address 1	Port 1	Role Name
	User Name 2	IPv6 Address 2	Port 2	Role Name
	User Name 3	IPv6 Address 3	Port 3	Role Name

FIGURE 6: *Figure Existing Member Information Message*

Resource Name 1	*Operation 1*	*User List 1*
	Operation 2	*User List 2*
Resource Name 2	*Operation 1*	*User List 1*

FIGURE 7: *Figure Resource List Message*

```
Operation = {Cache, Share}
```

4.5 XCAST6 Header Size

In the following, we derive an analytical formula to calculate the size of the XCAST header as a function of the number of destinations. The XCAST6 API that we use for implementation has the following packet format:

Payload
UDP
XCASTRouting Ext + Dest Opt
Inner IP src =A, Dst=All_X_, Protocol= XCAST
XCAST SP-tunnel, Hop-by-hop
outer IPsrc=A, dst=B, prot=IP

FIGURE 8: *Figure XCAST6 Header Format*

If you need to quote a book reference [**?**] you can do it like this. A paper by Author(s) [1] in a conference proceedings or journal can be quoted in a similar manner. Therefore, the header for the XCAST6 API can be decomposed into the following parts:

XCAST Header = IPv6 Header (outer) + Hop-by-Hop Header + IPv6 Header (inner) + Routing Extension Header (variable) + Destination Option Header (variable) + UDP Header

Payload
UDP
XCASTRouting Ext + Dest Opt
Inner IP src =A, Dst=All_X_, Protocol= XCAST
XCAST SP-tunnel, Hop-by-hop
outer IPsrc=A, dst=B, prot=IP

FIGURE 9: *Figure XCAST6 Header Size*

where, n is the number of destinations in the XCAST packet
Note that the size of the Destination Option Header depends on the number of destinations, such that it has a value equal to 8 bytes for the first 2 destinations and increases by 8 bytes for every 4 more destinations in the destination list. Hence, for 3, 4, 5, and 6 destinations a length of 16 bytes, for 7, 8, 9, and 10 destinations, it has a length of 24 bytes, and so on. For the calculation of the Destination Option Header size, the following algorithm can be used:

destinationHeaderLength = 8
for n=3 to Number of Destinations do
destOpt=1
count=1
while count¡=4 do
*variablePart=destOpt*8*
destinationHeaderLength=
destinationHeaderLength+variablePart
destOpt=0
count ++
Using the above formula, the size of the XCAST for different number of destinations can be calculated.

Number of Destinations	*XCAST Header Size*
1	*152*
2	*168*
3	*192*
4	*208*
5	*224*
6	*240*

FIGURE 10: *Figure XCAST6 Header Size for various Numbers of Destinations*

5 System Evaluation

5.1 Criteria for Evaluation

One of the factors that define the efficiency of the system is the time required by the system for providing various functionalities it offers. Hence, we consider delay as one of the parameters for the system evaluation. The results show the delay results in milliseconds. Furthermore, since the implementation of the system used XCAST in the network layer, which is an experimental protocol, it becomes necessary to evaluate the performance our system with other systems that would use the same design but use other technology like Multicast and Unicast. Therefore, technology in the underlying layer is another criterion for system evaluation. Finally, the ability of the system to perform equally well with large number of nodes is one of the most important criterion for evaluation of any system. Scalability of the system in terms of the maximum number of simultaneous groups possible, maximum number of resources possible in each group, and maximum number of members possible in each group are the parameters of the system that would be used for the evaluation of system.

5.2 Test Bed Configuration

For the real time measurement of the delay in the system for various functionalities of the system, we used a test bed consisting of 2 routers and 5 normal nodes. The topologies similar to the following topology 1 with one or two router are the network configurations that were used.For the comparison of performance of the system with other systems based on the traffic flow in the entire network, the following three different network topologies (topology 1, 2, 3) Fig 9 are considered.

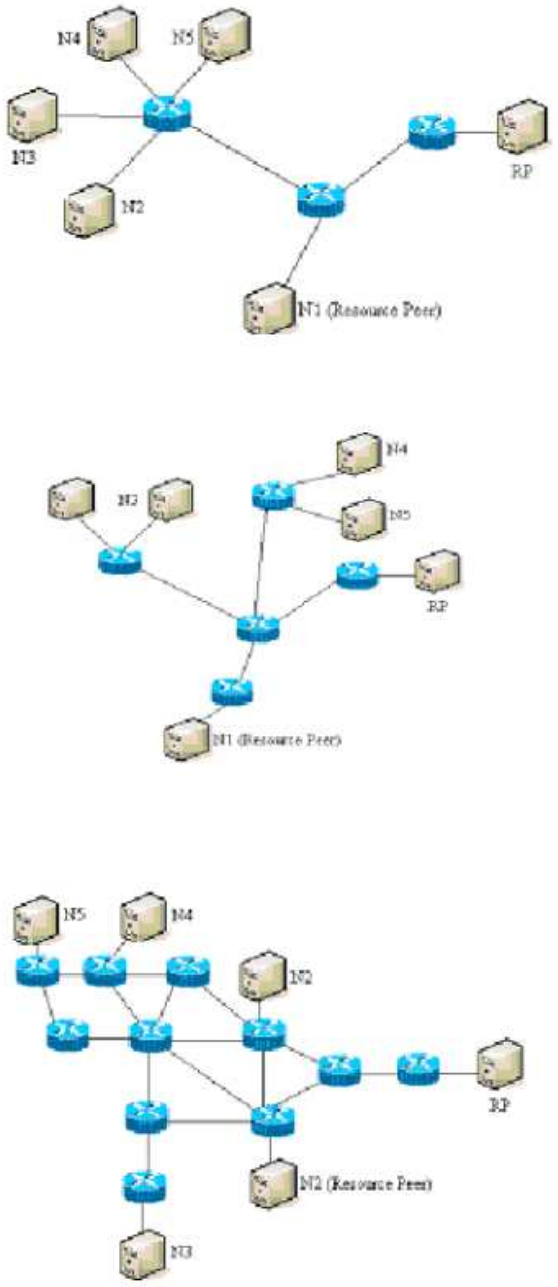


FIGURE 11: *Figure Topology Considered for Evaluation*

5.3 Delay Measurements

Group Join Time
The Group Join Time was measured when the RP is one router and two routers behind in some scenario like topology 1. The variation of Group Join Time was observed by changing the number of existing members in the group. Fig 10 shows the variation of the group join time with respect to the existing number of members in the group.

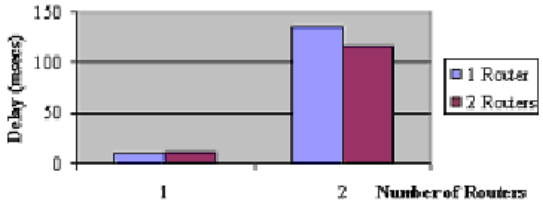


FIGURE 12: *Figure Node Join Delay*

The Group Join Time varied from 10 milliseconds to 135 milliseconds in case of the RP lying 1 Router behind. In the case of the RP lying 2 routers behind, it varied from 12 milliseconds to 117 milliseconds. This difference in Group Join Time is due to the implementation issue of the FreePastry API. For the maintenance of the Pastry ring, the API sends periodic messages (like routing table updates, keep-alive messages, etc) to other nodes in the ring. Hence, whenever a group join request is to be sent, the request message might have to stay in the message queue of sender, the intermediate nodes it travels before getting delivered to the RP, and the RP itself, in case if any of these nodes is already busy processing the periodic API messages. It can also be observed that the maximum delay value observed for 1 router is higher than the maximum delay value observed for 2 routers. This supports our explanation of the delay not being related to the size of the network but the length of the message queue at the nodes it passes through before getting delivered to the RP.

Average New Node Arrival Notification Delay
The variation of Average New Member Arrival Notification Delay was observed by changing the number of existing members in the group. The following graph shows the variation of the Average Member Arrival Notification Delay with respect to the group size

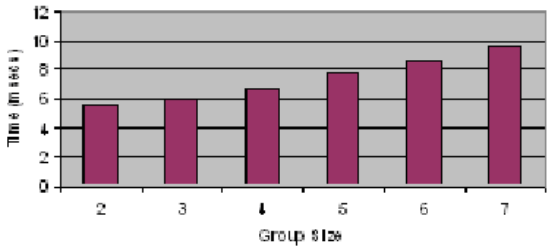


FIGURE 13: *Figure Average New Member Arrival Notification Delay*

It was observed that the Average New Member Arrival Notification Delay increases with the increase in the number of existing members. This is because as there are more existing members to inform about the new member, the size of the XCAST packet becomes higher due to increase in header size. The time the receiver has to spend constructing the header and the time the router has to spend processing the packet increases; hence in effect, the Average Member Arrival Notification Delay increases.

5.4 Technology

For the evaluation based on technology, the performance of the system developed (using XCAST) is compared two other hypothetical systems that would provide the same functionality but using Multicast and Unicast in the underlying layers. The following parameters are used for comparison:

Routing Entries in the Intermediate Routers
This parameter compares the performance with respect to the number of additional routing entries required in the intermediate routers for the system to work.

Traffic Flow
This parameter compares the performance with respect to the traffic flow in the system for various operations of the system. We focus on the following points for the comparison of traffic flow

Stress on Rendezvous Point
Stress on RP refers to the stress in terms of the traffic flow in the RP whenever a new node joins the group. We observe the variation in the stress with respect to the group size.

Stress in RP = Register message traffic from new node to RP + New Member Arrival message(s) traffic from RP to existing group members + Existing Group Members message traffic from RP to new node

Since the Register Message traffic from new node to RP is the same independent of the technology being used in the lower layer (since the message is Unicast in any case because the number of receiver for the message is one), we consider only the New Member

Arrival message traffic and Existing Group Members message traffic for the comparison of the stress in RP.

Stress on Exiting Node

Stress on Exiting Node refers to the stress in terms of the traffic that flows through the interface of the node that is exiting from the group. We observe the variation in the stress with respect to the group size.

Stress on Exiting Node = BYE message traffic from exiting node to member no + BYE message traffic from exiting node to RP

Since the BYE message traffic from exiting node to RP is the same independent of the technology being used in the lower layers, we consider only the BYE message traffic from exiting node to other members of the group

Traffic Flow in Entire Network

For the comparison of performance of the system with other systems based on the traffic flow in the entire network, the following three different network topologies are considered: **New Member Join Traffic**

New Member Join Traffic is the traffic that flows in the system when a new member arrives to the group. This traffic can be calculated as follows:

New Member Join Traffic = Register message from new node to RP + New Member Arrival message(s) from RP to existing group members + Existing Group Members message from RP to new node

Group Leave Traffic

Group Leave Traffic is the total traffic flow (in bytes) in the system when a node leaves the group.

Group Leave Traffic = BYE message traffic from exiting node to remaining members of group + BYE message traffic from exiting node to RP

Since the BYE message traffic from exiting node to RP is the same independent of the technology being used in the lower layers, we consider only the BYE message traffic from exiting node to other members of the group.

Access Right Update Traffic

Access Right Update Traffic is the total traffic flow (in bytes) in the system for sending the updates of the changes in the access right assignment for resources to all the members of the group (from the Resource Peer of the group).

Access Right Update traffic = Access Right Change Information message traffic from Resource Peer to other members of the group

5.5 Scalability

Scalability refers to the capability of the system to handle increasing number of nodes or objects that are part of the system. In this regard, the following parameters for the system are evaluated: Maximum number of members in each group Maximum number of simultaneous groups

6 Results and Analysis

6.1 Technology

In this section, we compare the performance of our system using XCAST with two other hypothetical systems that would use Multicast and Unicast respectively in the network layer for achieving the same functionality that the current system provides.

6.2 Multicast

A system using multicast technology in the network layer for the purpose of multiple simultaneous group formation and collaboration would require prior arrangement of multicast address for each group. The nodes interested in same lgroup should know the respective multicast address for the group and join the group. Group formation would not require topic based searching; rather it will require nodes to join multicast group address corresponding to the group they are interested in. If there is a large number of such groups, which is very plausible if there are numerous playing groups in the arena, using Multicast technique would require pre-allocating Multicast address for each of the groups.

Routing Entries in the Intermediate Routers

For each source in each group, there should be a separate entry in the routing tables of the intermediate routers in the network. This highly wastes router memory. For n such groups and m members in each group, the routing entries in intermediate routers is O(n*m). On the other hand, for the system implemented using XCAST, the already existing Unicast routing tables are sufficient for the system to work and there is no need to add additional information in the intermediate routers in the Internet. Hence, from the perspective of router memory, the performance of the system we developed using XCAST in the underlying layer is better than that of the other system that would provide the same functionality using Multicast.

Traffic Flow

From the perspective of the traffic that flows through the sender and receiver interfaces, the system using Multicast would have comparatively less traffic flow than our system that uses Xcast. This is because

Multicast uses normal IPv6 packets whereas Xcast packets have higher size due to the additional header information to be added to each Xcast packet. But, if we consider the bandwidth flow in the whole network, our Xcast based system is more efficient since using Xcast avoids the control overhead that occurs in a Multicast based system. In Multicast, considerable amount of traffic overhead occurs in the whole network for the periodic construction of the Multicast trees .

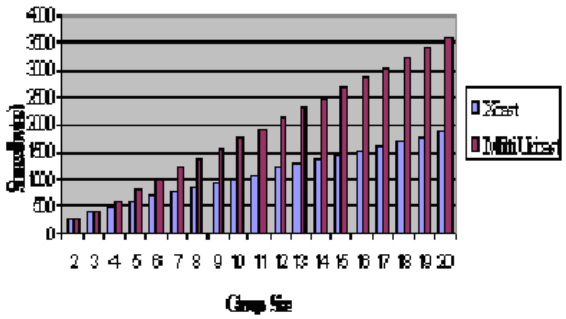


FIGURE 14: *Figure Variation of Stress on ERP w.r.t Group Size*

6.3 Unicast

Routing Entries in the Intermediate Routers
The system using Unicast would have to make no prior arrangements of address allocations unlike the case of Multicast. Similarly, there is no necessity for extra entries in the intermediate routers per source per group. In this regard, our system would have same performance as compared to the Unicast based system.

Traffic Flow
Theoretically, the performance of the system that has been developed should be better than the Unicast based system in terms of the traffic flow. This is because the XCAST based system requires any sender to send one packet instead of n packets to n receivers, unlike the case of Unicast based system where n identical packets have to be sent to n receivers.

Stress on RP
The following graph shows the comparison between XCAST and Unicast based systems from the perspective of stress on RP. It also shows the variation of the stress with respect to the number of members in the group. The graph shows that the stress on RP for the XCAST system developed is less than that for the Unicast based system for all group sizes greater than 3. This is because for the XCAST based system the RP has to send only one packet instead of sending n packets for n destinations. It also shows that stress on RP increases with the increment in the number of existing members in the group for both XCAST and Unicast based systems. But, the rate of increment of stress for XCAST based system is lower than that for Unicast based system. This is because the increment in n, the number of receivers, increases the gain in terms of the bytes saved by transmitting only 1 packet instead of n packets. In this regard, the performance of the system developed is better than the one using Unicast.

Stress on Exiting Node
The following graph shows the comparison between XCAST and Unicast based systems from the perspective of stress on a node exiting from a group. It also shows the variation of the stress with respect to the number of members in the group. The graph indicates that the stress on the exiting node is lower in our system compared to that in Unicast based system; the stress increases with the increment in the group size but the rate of increment is much lower for XCAST based system compared to Unicast based system. This is because for the XCAST based system the RP has to send only one packet instead of sending n packets for n destinations. Therefore, from the perspective of stress on exiting node, the XCAST based system performs better.

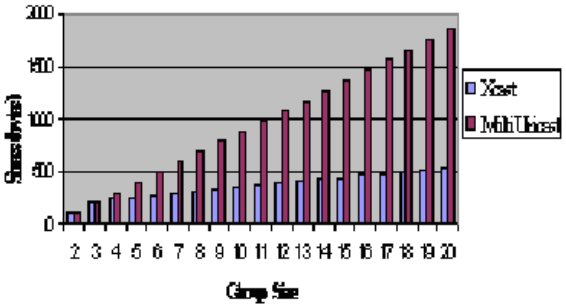


FIGURE 15: *Figure Variation of Stress on Exiting Node w.r.t Group Size*

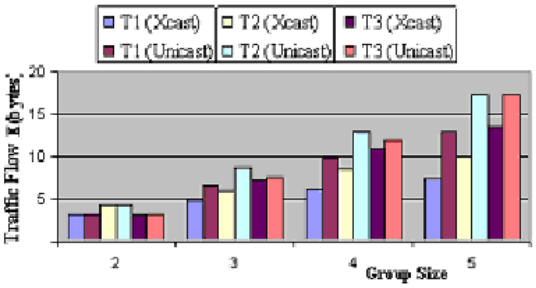


FIGURE 16: *Figure Traffic Flows for Various Topologies and Group Sizes (1024 Bytes of data)*

The above graph shows the traffic flow in the network when the sender has to send data size of higher value, for example, video data of 1024 bytes from the Resource Peer to all the other nodes in the group . It is evident that in this case, the traffic flow in our system is less than in the system using Unicast. This is precisely because the gain obtained in the sender side is greater than the total payment to be made in all receivers in terms of higher header size of XCAST. When the later phases of this system will be implemented, there will be situations when large sized data should be transmitted to more than one receiver. In such a situation, the system will show good performance than the system using Unicast.

6.4 Scalability

Scalability refers to the capability of the system to handle increasing number of nodes or objects that are part of the system. In this regard, the following parameters for the system are evaluated: Maximum number of members in each group Maximum number of simultaneous groups

Maximum number of members in each group The maximum number of members in each group is limited because of the limited packet size of an XCAST packet. The current XCAST6 API does not support packet fragmentation. Hence, any packet that exceeds the Maximum transmission Unit (MTU) of 1500 bytes will not be automatically fragmented by the API. There has to be some kind of application layer means of measuring the packet size of the packet being sent and then applying application layer fragmentation in the data. Currently, the application does not support that. During the departure of a node, it sends a BYE message to all other members using an XCAST packet. Hence, the size of the packet increases with the increment in the existing number of members. The size of the XCAST packet for various values of existing group size can be calculated analytically (since the header size of XCAST packet for different numbers of destinations can be calculated). The variation of the BYE message packet size with remaining group size is depicted in the graph below. It is clear that the packet size exceeds the MTU of 1500 bytes when group size reaches 75. Hence, the maximum group size is 74.The limit on group size due to the size of XCAST packet for sending access role changes information to group members (by the Resource Peer) is also the same because the data size for this operation is the same as BYE message data size.

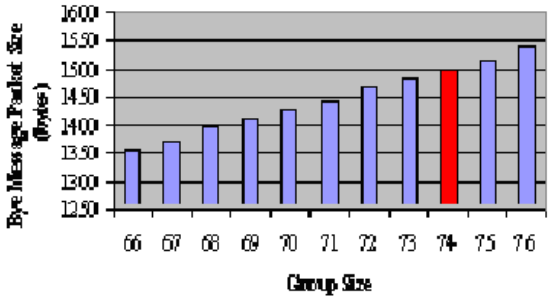


FIGURE 17: *Figure Limit in the Group Size due to Bye Message*

During the new member arrival phase, the RP sends the new member information to all existing members using an XCAST packet. The following graph shows the variation in the packet size with increment in existing group size:

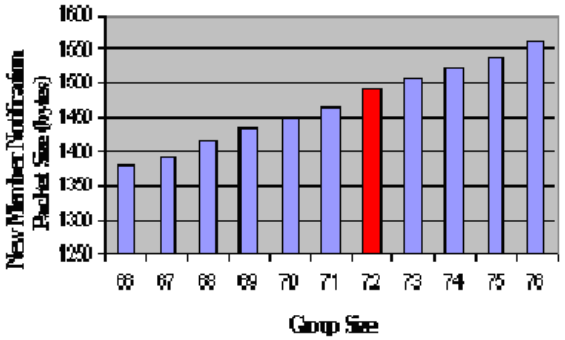


FIGURE 18: *Figure Limit in the Group Size due to New Member Notification Message*

It is obvious from the above graph that the packet size exceeds 1500 bytes for group size of 73. Hence, in effect, the maximum group size when a new node joins the group is 72. Taking the lower bound between 72 and 74 (as obtained from the two graphs above), the maximum group size is 72.

7 Conclusion and Future Work

We also proposed an architecture that utilizes the layers machines to allow gaming companies to scalably reach out to larger audiences without a huge investment in the infrastructure. The solution proposed uses a structured P2P technology based approach to form multiple simultaneous groups of nodes (using XCAST6) such that nodes with interests in the same game are brought together in a single group. Inside each group, members are allowed to have access rights assigned to control and observe the resources for the game. The robustness of the design lies in the capability of the system to continuously provide group formation and collaboration

in presence of continuous arrival and departure of nodes. Our design is efficient in the sense that the use of P2P technology prevents failure of any one node from affecting process of group formation and collaboration in the entire network. The research also evaluates the system using various criteria delay, traffic, and scalability. The delay of the system for various operations is in the order of milliseconds, which is reasonable for test bed in a Local Area Network and it has been found that the system performs and scales much better then systems using Multicast or Unicast traffic. The immediate future extension to the system would be to implement the design proposed to provide robustness in the presence of silent departure of nodes from the system. The current system requires the members to know the exact name of the group game they wish to join. The system design can be extended to provide key word based searching for the group such that it obviates the need for any member to know the exact group name.

References

[1] Ross, K.W., 2005, *Tutorial on Peer-to-Peer Internet.* Sixteenth Asian School on Computer Science, AIT, Thailand.

[2] Bryan, D. et. al., *A P2P Approach to SIP Registration and Resource Location.*, Internet Engineering Task Force (IETF) website: http://www.ietf.org/internet-drafts/draft-bryan-sipping-p2p-02.txt

[3] Gong, Y., 2005, *Identifying P2P users using traffic analysis.* SecurityFocus.website: http://www.securityfocus.com/infocus/1843/2

[4] Imai. Y. and Kishimoto. H., 2003.,*Xcast6 : eXplicit Multicast on IPv6.*, Information Processing Society of Japan website: http://www.ipsj.or.jp/english/saint/2003/log/wk4-ipv6/xcast6.pdf

[5] Ooms. D.,2005, *Small Group Multicast (Explicit Multicast) Overview.* Internet Engineering Task Force

[6] McCaffrey D. Finney J., 2004, *The need for real-time consistency management in P2P mobile gaming environments* Proceedings of the First ACM International Conference on Advances in Computer Entertainment Technology (ACE 2004) in co-operation with SIGCHI.

[7] Keller J. and Simon G., 2002, *Toward a Peer-to-peer Shared Virtual Reality.* In 22nd International Conference on Distributed Computing Systems (ICDCS02) pages 595 -601. IEEE Computer Society.

[8] *The Gnutella Protocol Specification,* , http://dss.clip2.com/GnutellaProtocol04.pdf

[9] Wang H. et al, 2005, *To Unify Structured and Unstructured P2P Systems.*,In the proceedings of the 19 th IEEE International Parallel and Distributed Processing Symposium (IPDPS05), 1530–2075

[10] Jiang S. et al., 2003, *LightFlood: an Efficient Flooding Scheme for File Search in Unstructured Peer-to-Peer Systems.*, In the proceedings of the International Conference on Parallel Processing (ICPP03), 0190–3918.

[11] *The FreePastry homepage*, (2005). Ffrom: http://freepastry.org/FreePastry/

Rapid Controller Prototyping at the SUPSI Laboratory

Roberto Bucher
University of Applied Sciences of Southern Switzerland
Galleria 2, Lugano-Manno 6928, C H
roberto.bucher@supsi.ch

Abstract

The Scuola Universitaria Professionale della Svizzera Italiana (SUPSI) is a university of applied sciences which offers students the possibility to get a specialization in mechatronic in the third study year. One of the most important parts of this study is represented by the mechatronic laboratory, where students can put all the tasks connected to the different lectures into practice. Rapid Control Prototyping methods (RCP) play a key rule to achieve these objectives.

Since 2003, the SUPSI has based his control laboratory on Linux RTAI. The RTAI-Lab project allows to directly integrate the code generated from the commercial suite Matlab/Simulink/RTW or from the open source suite Scilab/Scicos/RTAICodeGen into Linux RTAI.

Different blocks and functions which facilitate the plant identification and the control design have been created for both environments.

1 Introduction

The Scuola Universitaria Professionale della Svizzera Italiana (SUPSI) is a university of applied sciences which offers students the possibility to get a specialization in mechatronic in the third study year. One of the most important parts of this study is represented by the mechatronic laboratory, where students can put all the tasks connected to the courses of *Modelling and Identification*, *Control Design*, *Mechatronic systems* and *Sensors and actuators* into practice. Rapid Control Prototyping methods (RCP) are used to achieve these objectives.

Section 2 gives an overview about the mechatronic laboratory at SUPSI. Section 3 presents a state feedback control of a simple servo DC motor. The identification of a voice coil motor for nanopositioning is described in section 4. The classical inverted pendulum is described in section 5, where the PC is connected to the plant using a CAN bus. Some conclusions about the experiences collected in the past 4 years are reported in section 6.

2 The SUPSI laboratory

2.1 Equipment

Each labor place is composed of a PC (Pentium 4, 2.8 GHz) with Linux RTAI as real-time operating system, a CAN bus interface (Peaks PCAN or a self built EPP CAN Dongle) and an embedded system (Compact PCI) as supplementary RT target for distributed control. Some PCs are equipped with AD/DA cards (Computer Measurements PCI-1200 DAQ cards).

2.2 Linux RTAI

The RTAI extension was created as an environment for implementing low cost data acquisition and digital controller systems ([1], [2]) and is intended as an open source replacement of other hard real-time OSs such as QNX or VxWorks. This extension adds hard real-time capabilities to Linux, allowing sample frequencies up to several thousands of cycles per second with a jitter of just a few microseconds. Real-time processes can run both in the kernel and in the user area. The low latencies guaranteed by Linux RTAI allows us to implement controllers for systems with a bandwith up to a few hundred hertz.

2.3 RTAI-Lab

RTAI-Lab is an open source project which integrates the code generated by a CACSD (Computer Aided Control System Design) environment into a Linux RTAI hard real-time task. At present, three CACSD suites are supported: the commercial

MATLAB/Simulink/RealTime-Workshop (RTW) suite, the commercial EicasLab suite and the open source SCILAB/Scicos suite. The system has been widely described in [3], [4], [5], [6] and [7]. The document [8] describes in detail how to install the full RTAI-Lab toolchain. RTAI-Lab provides a GUI application for remote monitoring and controlling of hard real-time generated tasks.

3 Control of a DC servo motor

3.1 Plant description

The plant is represented by a servo DC motor from Siemens AG (Fig. 1)

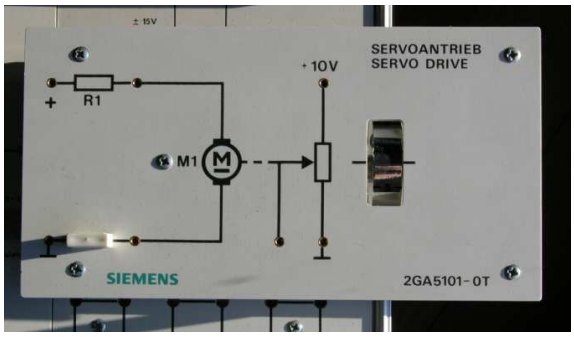


FIGURE 1: *Servo DC motor*

This plant is part of an Universal Experimenter Didactic System, which allows the investigation of PID controllers. In our case, the controller is implemented as a digital controller in the Linux RTAI OS and only the current amplifier and the plant are considered. The interface between the PC and the plant is implemeted by a AD/DA card of Computer Measurements (PCI-1200) using the COMEDI interface([9]) which sends the control signal to the motor and reads the position using a potentiometer.

3.2 Identification

A simplified model of the servo DC motor is represented by

$$G(s) = \frac{K}{s^2 + \alpha \cdot s} \qquad (1)$$

The Scicos block diagram used for the identification is represented in Fig. 2.

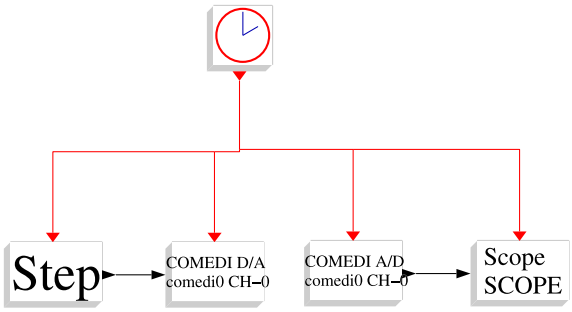


FIGURE 2: *Block diagram for the identification*

The motor can get control signals between $-10V$ and $10V$; a step signal of $8V$ is sent to the real plant and the response is collected using the *xrtailab* application. A simple script (see 3.6) performs the identification using the *leastq* command of the Scilab environment. The measured data and the step response of the identified plant are shown in Fig. 3.

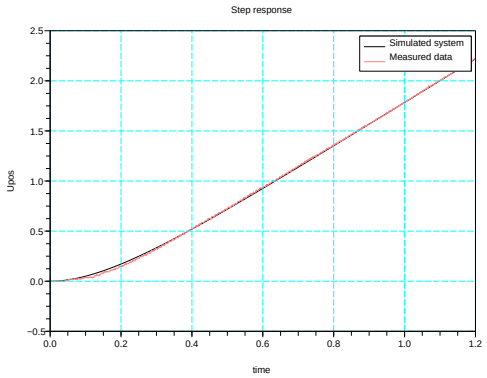


FIGURE 3: *Measured and simulated step response*

3.3 Controller design

The state feedback controller has been implemented using the pole placement method. The 3 poles of the controlled system are placed near to the position of the plant pole. The script which calculates the controller is the same used for the identification (see subsection 3.6). A discrete reduced order observer is used to generate the unknown state of the plant (see A.1).

3.4 Simulation

The same Scicos block diagram has been used for simulation and realtime control; the only difference between the two block diagrams is represented by the superblock *plant* (see Fig. 4 and Fig. 5) and by the *scope* block.

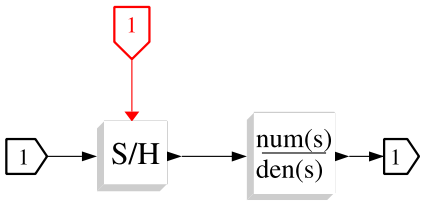


FIGURE 4: *Scicos blocks for simulation*

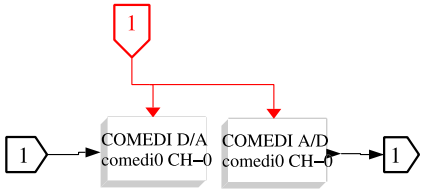


FIGURE 5: *Scicos blocks for real-time code generation*

3.5 Real-time controller

In the case of the real-time control, the mathematical representation of the plant is substituted by two COMEDI blocks, in order to send and collect data from the AD/DA card (see Fig. 5).

The controller has been realized under Scicos (see Fig. 7). It contains a discrete reduced order observer and an integral part with anti-windup to eliminate the steady state error.

Fig. 6 shows the plot of the simulation vs. the plot of the real plant.

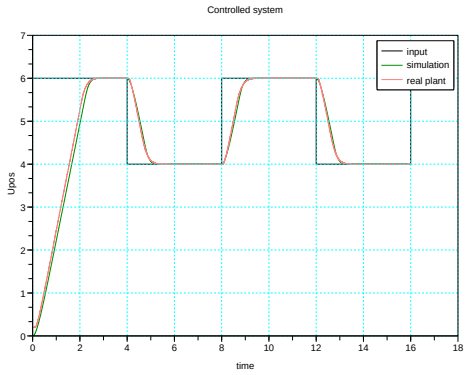


FIGURE 6: *Controlled system - Response of the simulation and of the real plant*

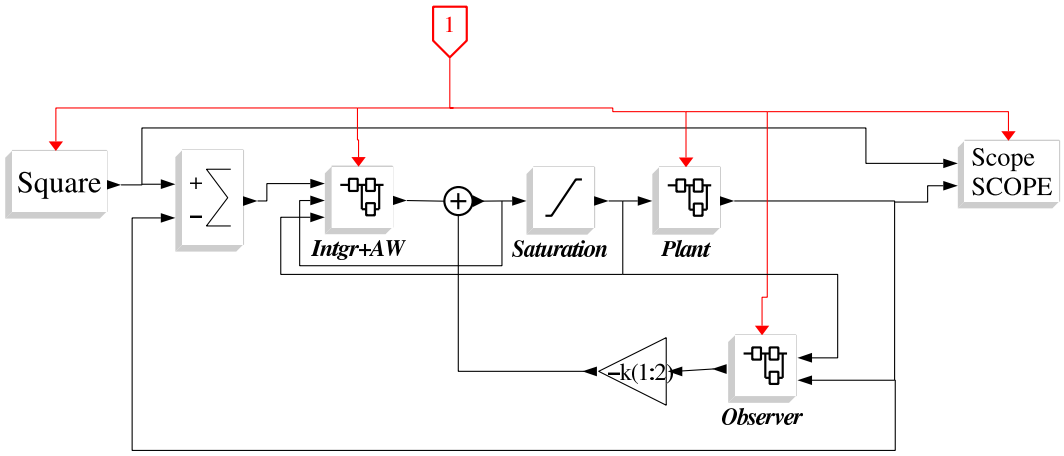


FIGURE 7: *Block diagram for the controller (simulation*

3.6 Scilab script

```
// == Servo DC: Identification and Control design ==
x=read('SCOPE',-1,2);
t=x(:,1);
y=-x(:,2);
y=y-y(1);
t=t(1:4001);
y=y(1:4001);
xbasc()
plot(t,y)

ts=1e-3;

function z=fun(p,t,y)
z=y+p(1)/p(2)^2-p(1)/p(2)*t-p(1)/p(2)^2*exp(-p(2)*t);
endfunction
```

```
Uo=8;
p0=[1,4];
p02=[1,4,1];

[ff,p]=leastsq(list(fun,t,y/8),p0);

g=syslin('c',p(1)/(s*(s+p(2))));
xbasc()
bb_step(Uo*g,4)
plot(t,y)

num=g.num;
den=g.den;

g_ss=tf2ss(g);
p_g=roots(den);
```

```
g_dss=dscr(g_ss,ts);
p_max=max(abs(p_g));

p_c=[-p_max,
     -p_max*cos(pi/6)+%i*p_max*sin(pi/6),
      -p_max*cos(pi/6)-%i*p_max*sin(pi/6)];
k_poles=1.2;
p_c=k_poles*p_c;

p_d=exp(p_c*ts);
[Phi,G,C,D]=abcd(g_dss);

[m1,n1]=size(Phi);
[m2,n2]=size(G);
Phi_f=[Phi,zeros(m1,1);-C(1,:)*ts,1];
G_f=[G;zeros(1,n2)];

k=ppol(Phi_f,G_f,p_d);
k_sat=k(3)/30;

p_oc=[-5*p_max];
p_od=exp(p_oc*ts);

T=[0,1];
[Ao,Bo,Co,Do]=redobs(Phi,G,C,D,T,p_od);
```

4 Identification of a voice coil motor

4.1 Plant description

One of the most difficult tasks in control system design is represented by the identification of the plant model. This section describes how Scilab/Scicos/RTAI has been used to find the non-parametric and the parametric identification of a voice coil motor. SUSPI has great experience in controlling high precision systems ([10]). The plant to be identified is a voice coil motor (Fig. 8) used for high precision positioning. It can perform deplacements up to $1mm$ with a precision of about $50nm$.

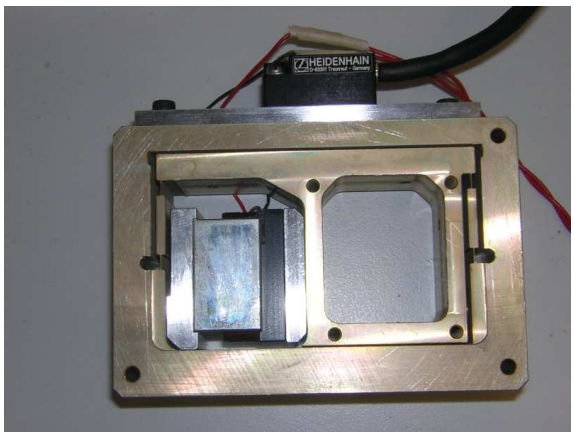


FIGURE 8: *Voice coil motor*

The controller is implemented in a Compact PCI system, where a 10MB Linux-RTAI with full network support is stored into a Flash Card. The two cards to interface the real plant (a DA analog card and a sincos encoder card) have been designed and built at SUPSI. The hard realtime drivers have been also created.

The controller is designed and simulated on a second PC. The generated code is then downloaded and executed into the Compact PCI. The *xrtailab* monitor application can be started on each PC in the LAN to monitor the hard real-time task.

4.2 Identification

The scicos block diagram for the identification is shown in Fig. 9.

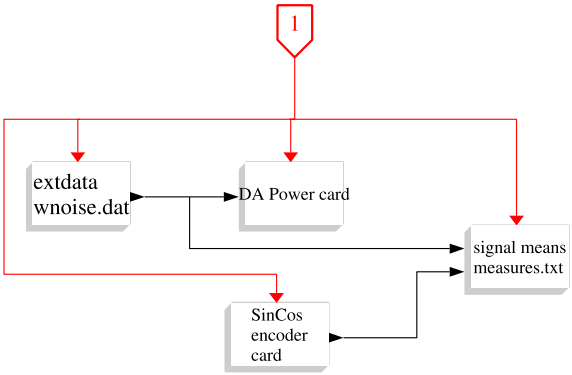


FIGURE 9: *Block diagram for the identification*

The plant is excited by a white noise signal of $10s$ which is sent repeatedly to the plant. This signal and the sensor values are collected by the block *signal means*. This block collects the input data and creates a cumulative sum of packet of $10s$. When the generated hard realtime task is stopped, this block creates a mean of the collected signals in order to reduce the effect of sensor noise, and store it in the file *measures.txt* . The script in subsection 4.3 performs the identification of the plant. The core of the script is the function *xspectrum* (see A.4) which performs the non-parametric identification of the system (represented in Fig. 10); the parametric identification of the plant is calculated using the scilab function *frep2tf* (the 2. order term in the numerator has been eliminated). The non parametric model and the gainplot of the parametric model are represented in Fig. 10.

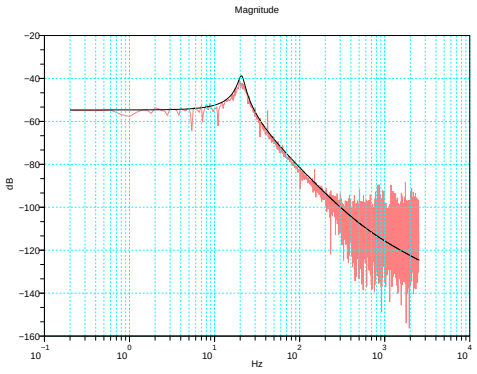


FIGURE 10: *Plot of the measured and identified system*

4.3 Identification script

```
data=read('measdata.dat',-1,3);
x=data(:,2);
y=data(:,3)/12;
ts=1/5000;

P=xspectrum(x,y);
frq=0:1/ts/2/length(P):(1/ts/2-1/ts/2/length(P));
xbasc()
gainplot(frq(2:$),P(2:$)')

N=2835;
h=frep2tf(frq(2:N),P(2:N),2);
h=ss2tf(tf2ss(h));
num=h.num
den=h.den
coeff_n=coeff(num)
coeff_n=coeff_n(1:2)
num=poly(coeff_n,'s','coeff')
sys=syslin('c',num/den)
repf=repfreq(sys,frq(2:$));
gainplot(frq(2:$),repf)
```

5 Control of the inverted pendulum

5.1 Plant description

The inverted pendulum is a classic application in a student laboratory. Our system is connected to the controller (a normal PC with Linux RTAI) through a CAN Bus. The CAN interfaces under Scicos have been realized in a three layer hierarchy: a middle layer has been introduced as a *generic CAN interface* to connect the different CAN devices to the different CAN cards (as shown in Fig. 11). In the SUPSI laboratory we have different CAN cards: for most of them we rewrote the driver in order to use them in a hard real-time environment. In particular, we have real-time drivers for CAN cards of Peak System ([11]) and for a self-built parallel port dongle.

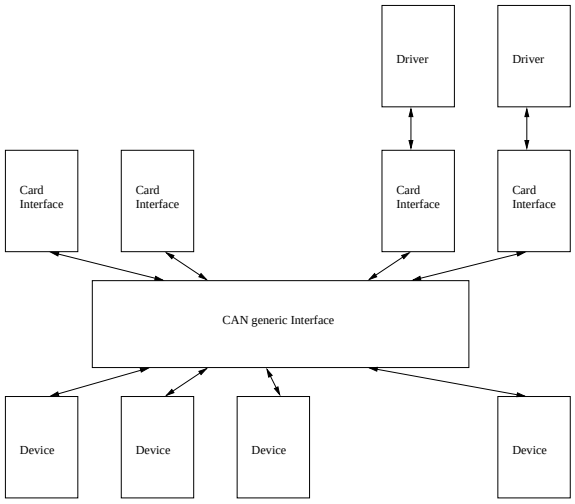


FIGURE 11: *Three layer CAN implementation*

The actuator is a DC motor from Maxon (RE 40) which is controlled by current through an Epos driver (EPOS P 24/5) ([12]). The same driver is used to count the impulses of the incremental encoder of the motor to get its position. The angle of the pole is measured using a second incremental encoder, which sends via RF the impulses to a receiver. The power supply to this encoder and to the transmitter has been realized by current induction.

5.2 Identification

The linear state-space model of the inverted pendulum is

$$\begin{bmatrix} \dot{\varphi} \\ \dot{\omega} \\ \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{m\cdot r\cdot(M+m)\cdot g}{\Theta\cdot(M+m)-m^2\cdot r^2} & 0 & 0 & \frac{-m\cdot r\cdot d}{\Theta\cdot(M+m)-m^2\cdot r^2} \\ 0 & 0 & 0 & 1 \\ \frac{m^2\cdot r^2\cdot g}{\Theta\cdot(M+m)-m^2\cdot r^2} & 0 & 0 & \frac{-\Theta\cdot d}{\Theta\cdot(M+m)-m^2\cdot r^2} \end{bmatrix} \cdot \begin{bmatrix} \varphi \\ \omega \\ x \\ v \end{bmatrix} \quad (2)$$

$$+ \begin{bmatrix} 0 \\ \frac{m\cdot r}{\Theta\cdot(M+m)-m^2\cdot r^2} \\ 0 \\ \frac{\Theta}{\Theta\cdot(M+m)-m^2\cdot r^2} \end{bmatrix} \cdot F \quad (3)$$

where M is the mass of the car, m is the mass of the pole, r is the distance to the center of mass of the pole, Θ is the inertia of the pole and d is the friction constant of the car.

The force F is obtained by the torque given by the motor and it is proportional to the motor current:

$$F = K_t \cdot \frac{I_{mot}}{r_p} \quad (4)$$

where r_p represents the radius of the motor gear.

The distance r to the pole center of mass is calculated from the geometry of the pole (different volumes V_a and V_m of the pole parts)

$$r = \frac{V_a \cdot \frac{r_a}{2} + V_m \cdot (r_a + \frac{r_m}{2})}{V_a + V_m} \quad (5)$$

The value of Θ can be identified by measuring the swing frequency ($\rightarrow$ the oscillation time T_{osc}) of the pole as

$$\Theta = m \cdot g \cdot r \cdot \frac{T_{osc}^2}{(4 * \pi^2)} \qquad (6)$$

5.3 Simulation

By the simulation the inverted pendulum has been implemented using a *C-Block2* block in Scicos. The C-code of this block contains the non linear description of the inverse pendulum ([13], pages 245-247).

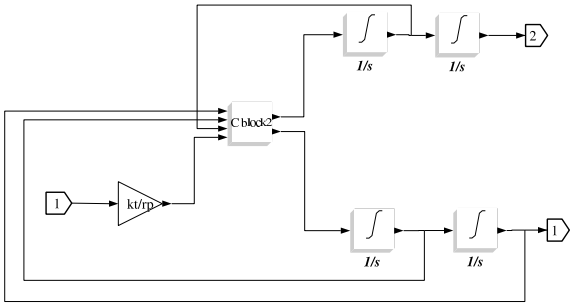


FIGURE 12: *Inverted pendulum - Block used for simulation*

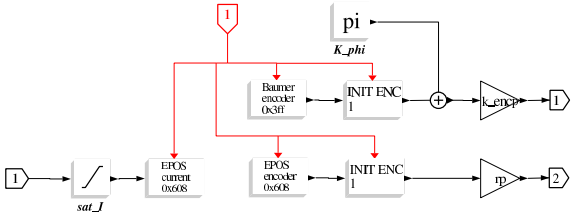


FIGURE 13: *Inverted pendulum - Block used for realtime controller*

Fig. 12 shows the superblock of the pendulum used for the simulation. The subsection 5.5 shows the code of the *C-Block2* block.

5.4 Real-time controller

Fig. 14 shows the scicos block diagram of the implemented controller.

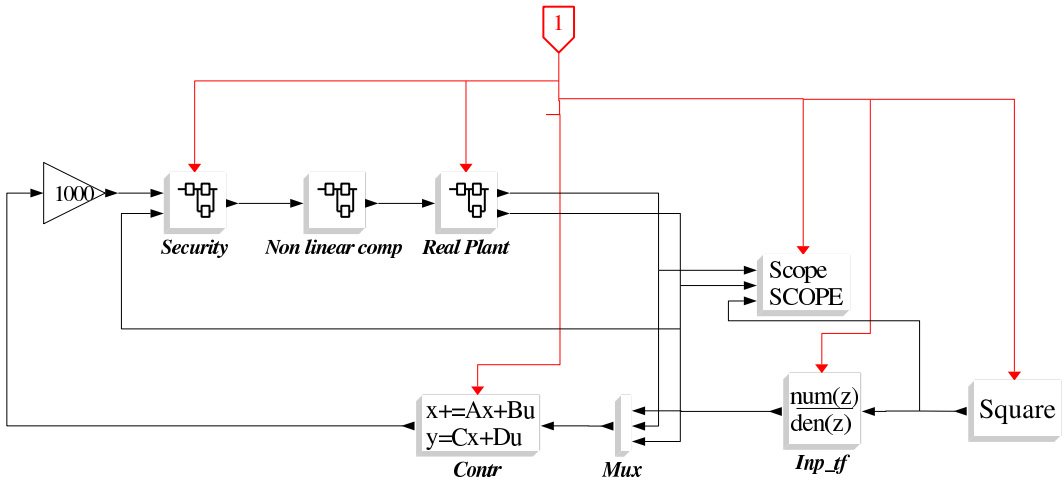


FIGURE 14: *Controller of the inverted pendulum*

All the scilab commands needed to implement the controller are shown in the subsection 5.6. The state feedback controller has been implementing using a LQR approach. The function *redobs.sci* creates the 4 matrices for the reduced order observer. As a last step, the reduced order observer, the integral part and the state feedback gains are concentrated in a single state space system, in order to avoid any possible algebraic loop in the scicos block diagram. The block of the pendulum for the real controller is represented in Fig. 13. The *Init encoder* block fixes the position of the encoder after 1*s* as *0* position.

A special block is represented by the *Security* block (see Fig. 15).

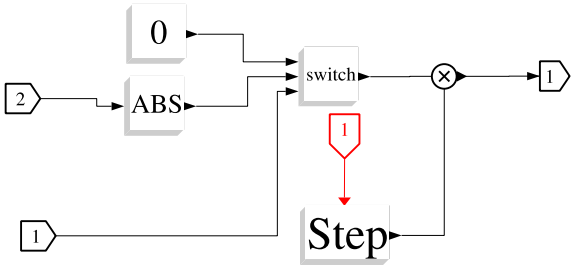


FIGURE 15: *Inverted pendulum - Security block*

The "switch" block stops the controller if the car position exits from a given interval. The *step* block is used to start the controller after an initialization time.

5.5 C-code of the non linear model

```
/* Inverted pendulum - Nonlinear model */
#include <math.h>
#include <stdlib.h>
#include <scicos/scicos_block.h>
void inv_pend(scicos_block *block,int flag)
{
  double M,m,g,Theta,r,d,th,thd,xd,u,delta,thdd,zdd;

  M=block->rpar[0];
  m=block->rpar[1];
  r=block->rpar[2];
  Theta=block->rpar[3];
  d=1.2571;
  g=9.81;
  th=block->inptr[0][0];
  thd=block->inptr[1][0];
  xd=block->inptr[2][0];
  u=block->inptr[3][0];

  delta=Theta*(M+m)-m*m*r*r*pow(cos(th),2);
  thdd=m*r*((M+m)*g*sin(th)-m*r*thd*thd*sin(th)*cos(th)+
       (u-d*xd)*cos(th))/delta;
  zdd=(m*m*r*r*g*sin(th)*cos(th)-Theta*m*r*thd*thd*sin(th)+
       Theta*(u-d*xd))/delta;
  block->outptr[0][0]=zdd;
  block->outptr[1][0]=thdd;
}
```

5.6 Control design of the inverted pendulum

```
// Inverted pendulum - State feedback control design
M=0.62336
m=0.08377+0.10834;
g=9.81;
Tosc=29/21;

r=0.026+.297;
Theta=m*g*r*Tosc^2/(4*pi^2);
d=1.2165
kt=60.3e-3;
rp=0.6/20.91044;
k_encp = -1;

Ts=5e-3;

Imin=0;  // For nonlinear compensation

A=[0, 1, 0, 0;
 m*r*(M+m)*g/(Theta*(M+m)-m*m*r*r),0,0,-m*r*d/
       (Theta*(M+m)-m*m*r*r);
 0, 0, 0, 1;
 m*m*r*r*g/(Theta*(M+m)-m*m*r*r),0,0,-Theta*d/
       (Theta*(M+m)-m*m*r*r)];
B=[0;
 m*r*kt/(rp*(Theta*(M+m)-m*m*r*r));
 0;
 kt*Theta/(rp*(Theta*(M+m)-m*m*r*r))]';

C=[1 0 0 0; 0 0 1 0];
D=[0;0];

sys=syslin('c',A,B,C,D);

sysd=dscr(sys,Ts);
[Ad,Bd,Cd,Dd]=abcd(sysd);

// LQR discrete controller
// weights
Q=diag([250,1,200,1,250]);
R=[1];

[m1,n1]=size(Ad);
[m2,n2]=size(Bd);
[m3,n3]=size(Cd);

// Add suppl. state (integral) to SS form
Ad_f=[Ad,zeros(m1,1);-Cd(2,:)*Ts,1];
Bd_f=[Bd;zeros(1,n2)];
Cd_f=[Cd,zeros(m3,1);zeros(1,n3),1];
Dd_f=[Dd;zeros(1,n2)];

k_lqr=bb_dlqr(Ad_f,Bd_f,Q,R);

preg=max(abs(spec(A)));

// Reduced order observer

poli_oss=exp([-preg,-preg]*10*Ts);
T=[0,0,0,1;0,1,0,0];
[Ao,Bo,Co,Do]=redobs(Ad,Bd,Cd,Dd,T,poli_oss);

// Compact form
Contr=comp_form_i(Ao,Bo,Co,Do,Ts,k_lqr,[0,1]);

// Filetr for input square signal
g=syslin('c',10/(s+10));
gz=ss2tf(dscr(tf2ss(g),Ts));
Ainp=0.3
```

6 Conclusion

This paper presented three practical examples explaining some basic activities connected with control system design. The Scilab/Scicos/RTAI suite has demonstrated to have reached enough maturity and to be an interesting alternative to other expensive commercial suites. This tool can be used for data acquisition, analysis, identification, control design and simulation. Just with a few steps, the same scicos block diagram used for simulation can be tranformed into a real-time controller running on a standard PC.

References

[1] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous, "RTAI: real time applications interface," *Linux Journal*, April 2000.

[2] E. Bianchi and L. Dozio, "Some experience in fast hard real-time control in user space with RTAI-LXRT," in *Real Time Linux Workshop*, Orlando, 2000.

[3] R. Bucher and L. Dozio, "CACSD with Linux RTAI and RTAI-Lab," in *Real Time Linux Workshop*, Valencia, 2003.

[4] R. Bucher, "Interfacing Linux RTAI with Scilab/Scicos," in *Real Time Linux Workshop*, Singapore, 2004.

[5] R. Bucher, L. Dozio, and P. Mantegazza, "Rapid Control Prototyping with Scilab/Scicos

and Linux RTAI," in *International Scilab Conference*, Paris, 2004.

[6] R. Bucher and S. Balemi, "Scilab/Scicos and Linux RTAI - A unified approach," in *IEEE conference on Control Applications*, Toronto, 2005.

[7] R. Bucher, "Targeting the Scicos Code Generator - The Linux RTAI Example," in *SCILAB Research, Development and Applications*, Wuhan, China, 2005.

[8] R. Bucher, S. Mannori, and T. Netter. (2006) RTAI-Lab tutorial: Scilab, Comedi and real-time control. [Online]. Available: https://www.rtai.org/RTAILAB/RTAI-Lab-tutorial.pdf

[9] COMEDI. Linux Control and Measurement Device Interface. [Online]. Available: http://www.comedi.org

[10] S. Balemi, J. Moerschell, J.-M. Breguet, D. Braendlin, S. Bottinelli, and I. Beltrami, "Surface Inspection System for Industrial Applications," in *Conf. on Robotics and Mechatronics*, Aachen, Germany, Sept. 2004, pp. 1597–1602.

[11] Peaks System. PCAN Linux website. [Online]. Available: http://www.peak-system.com/linux/index.htm

[12] Maxon motor. Maxon motor website. [Online]. Available: http://www.maxonmotor.com/

[13] S. Campbell, J.-P. Chancelier, and R. Nikoukhah, *Modeling and Simulation in Scilab/Scicos*. Springer, 2006.

A Some useful functions

A.1 Function *redobs.sci*

```
function [A_redobs,B_redobs,C_redobs,D_redobs]=redobs(A,B,C,D,T,poles)
P=[C;T]
invP=inv([C;T])

AA=P*A*invP

ny=size(C,1)
nx=size(A,1)
nu=size(B,2)

A11=AA(1:ny,1:ny)
A12=AA(1:ny,ny+1:nx)
A21=AA(ny+1:nx,1:ny)
A22=AA(ny+1:nx,ny+1:nx)

L1=ppol(A22',A12',poles)';

nn=nx-ny;

A_redobs=[-L1 eye(nn,nn)]*P*A*invP*[zeros(ny,nn); eye(nn,nn)];
B_redobs=[-L1 eye(nn,nn)]*[P*B P*A*invP*[eye(ny,ny);L1]]*[eye(nu,nu) zeros(nu,ny); -D, eye(ny,ny)];
C_redobs=invP*[zeros(ny,nx-ny);eye(nn,nn)];
D_redobs=invP*[zeros(ny,nu) eye(ny,ny);zeros(nx-ny,nu) L1]*[eye(nu,nu) zeros(nu,ny); -D, eye(ny,ny)];
```

A.2 Function *compform.sci*

```
function [Contr]=comp_form(A,B,C,D,Ts,K)
  // Create the compact form of the Observer ABCD and the
  // gain K,
  //
  // A,B,C,D: Observer matrices
  // Ts: sampling time
  // K: stte feedback gains

  ss_sys=syslin('d',A,B,C,D);
  ss_sys(7)=Ts;
  g_sys=ss2tf(ss_sys);

  gu=g_sys('num')(:,1)./g_sys('den')(:,1);
  gy=g_sys('num')(:,2:$)./g_sys('den')(:,2:$);

  Greg=[1/(1+K*gu),-K*gy/(1+K*gu)];
  Contr=tf2ss(Greg);
```

A.3 Function *compform_i.sci*

```
function [Contr]=comp_form_i(A,B,C,D,Ts,K,Cy)
// Create the compact form of the Observer ABCD and the
// gain K, using an integrator at the input
// to eliminate the steady state error
//
// A,B,C,D: Observer matrices
// Ts: sampling time
// K: stte feedback gains
// Cy: matrix to extract the output for the steady
// state feedback

  [larg,rarg]=argn(0);

  if rarg ~= 7 then
    Cy = [1]; // only 1 output
  end

  ss_sys=syslin('d',A,B,C,D);
  ss_sys(7)=Ts;
  g_sys=ss2tf(ss_sys);

  g_int=syslin('d',Ts/(%z-1));
  g_int(7)=Ts;

  gu=g_sys('num')(:,1)./g_sys('den')(:,1);
  gy=g_sys('num')(:,2:$)./g_sys('den')(:,2:$);

  nn=size(K,2);

  Ke = K(1,nn);
  K  = K(1,1:nn-1);

  Greg=[-Ke*g_int/(1+K*gu),(Ke*Cy*g_int-K*gy)/(1+K*gu)];
  Contr=tf2ss(Greg);
```

A.4 Function *xspectrum.sci*

```
function Txy=xspectrum(x,y)

  nfft=int(size(x,'*')/2);
  wind=window('hn',nfft)';
  n=size(x,'*');
  nwind=size(wind,'*');
  index=1:nwind;
  k=fix(n/nwind);

  Pxx=zeros(nfft,1);
  Pxy2=Pxx;Pxy=Pxx;

  for i=1:k
    xw=wind.*detrend(x(index));
    yw=wind.*detrend(y(index));
    index=index+nwind;
    Xx=fft(xw(1:nfft));
    Yy=fft(yw(1:nfft));
    Xx2=abs(Xx).^2;
    Xy=Yy .* conj(Xx);
    Pxx=Pxx+Xx2;
    Pxy=Pxy+Xy;
  end

  if modulo(nfft,2)==1 then
    selct=(nfft+1)/2;
  else
    selct=nfft/2+1;
  end

  Pxx=Pxx(1:selct);
  Pxy=Pxy(1:selct);

  Txy = Pxy./Pxx;
```

RTL-IO: An Extension of RTLinux I/O

Lei Wang
School of Computer, Beijing University of Aeronautics and Astronautics
No. 37, XueYuan Road, HaiDian District, Beijing City, China
wanglei@buaa.edu.cn

Chen Yang
School of Computer, Beijing University of Aeronautics and Astronautics
No. 37, XueYuan Road, HaiDian District, Beijing City, China
yangchen_buaa@sina.com

Xin Wang
School of Computer, Beijing University of Aeronautics and Astronautics
No. 37, XueYuan Road, HaiDian District, Beijing City, China
wangxin@buaa.edu.cn

Abstract

As Linux becomes more and more popular in many fields of science and industry, RTLinux is also widely used. But RTLinux only provides basic services, it does not support dynamic memory management, and its I/O support is not good enough either. There is a facility called RTFIFO in RTLinux. This method is inconvenient because developers have to provide a user space program for every real-time thread. In this paper, we studied dynamic memory management algorithms, and implemented the dynamic memory management module based on TLSF algorithm in RTLinux. Then we analyzed I/O mechanism used in RTLinux and designed a new I/O mechanism, RTL-IO. We have already used RTL-IO into Cyber data acquisition simulation system.

1 Introduction

Nowadays various real time operating systems based on Linux are developed to run the real-time application, For instance Hard-hat Linux [1] by Monta Vista company, Kurt-Linux [2] by Kansas University and RTLinux [3] by New Technical University of Mexico, etc. RTLinux is developed based on Linux operating system, because its sources code is opened and interfaces conform to POSIX, Nowadays RTlinux is more and more widely used.

RTLinux improves the real-time performance of Linux by adding a real-time subsystem to the Linux kernel and making the module support the hard real-time application. It provides an I/O method called RTFIFO [4]. Through this way, the control logic can be very simple, but the programmers have to implement corresponding user-mode program for every real time application that needs disk or net I/O, which complicates the development.

Not too many projects dedicate to I/O method of RTLinux so far, RTLFS (Real-time Linux File System) [5] is sponsored to solve the problem that RTLinux can not support real-time disk I/O, which is a sub-project of OCERA. With its help Linux can share the same hard disk with the real-time application. However, Linux cannot share the same partition with the real-time application.

We developed a new solution called RTL-IO to make RTLinux more convenient to handle the real-time I/O. The principle of RTL-IO is to utilize the resources and process ability of Linux to the largest extent. The real-time threads of program just make an I/O request via the interfaces provided, and then the request is encapsulated and saved within RTL-IO, Linux will process these I/O requests at proper

time. RTL-IO is the abbreviation of Real Time Linux Input and Output.

The remainder of this paper is organized as follows: Section 2 introduces the design of RTL-IO. In section 3, we present how we have implemented RTL-IO system. Section 4 shows our experimental results, and finally, section 5 summarizes the paper.

2 Design of RTL-IO

The basic structure of RTL-IO is illustrated in Figure 1:

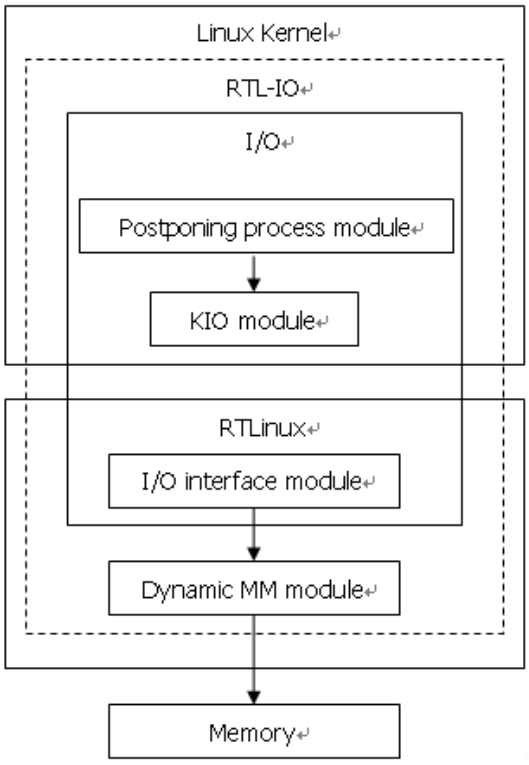


FIGURE 1: *RTL-IO*

As we see from Figure 1, RTL-IO consists of 2 parts: the I/O module and the dynamic memory management module. The former will provide the I/O interfaces and process all the I/O requests, and the latter is in charge of dynamically allocating memory to a program and retrieving it.

I/O module is separated into several parts, the I/O interface module provides I/O interfaces and the KIO module encapsulates all the I/O services provided by Linux and postponing process module handles all the requests using the postponing process function provided by the Linux kernel.

2.1 Design of dynamic memory management module

We use the TLSF [6] algorithm to develop the dynamic memory management module, whose time complexity is O(1) for both allocating and retrieving memory, even in the worst case. Its real-time performance is much better than other algorithms.

TLSF uses two levers separating arrays to save the available memory block list information. 1st lever array separates the memory space into several address range according to its FLI(First Lever Index), and 2nd lever separating array divided the address range which have been divided by the 1st lever array into some smaller ranges. Suppose f is the index of 1st lever array (called 1st lever index) while s (called 2nd lever index) is the index of the 2nd lever array, the memory size ranges corresponding to f and s can be calculated with Formula (1) and (2), respectively:

$$[2^f, 2^{f+1} - 1](1)$$

$$[2^f + s \times 2^{f-SLI}, 2^f + (s+1) \times 2^{f-SLI} - 1](2)$$

The defect of TLSF algorithm is that the lower limit of the address range calculated is greater than the requested block size, but a list may possible exist whose upper limit is greater than the requested size while lower limit less. That list may have an adequate memory block for allocation, but the allocation may still fail even if a memory block that can be allocated exists.

2.2 Design of I/O module

As we have mentioned, the I/O module consists of 3 parts: the I/O interface module, the postponing process module and the KIO module.

When it is necessary, the Real-time thread makes an I/O request, the I/O interface module encapsulates the request and inserts it into the corresponding list, then calls the Linux postponing process module to handle the request and makes the real-time thread sleep. The Postponing process module calls Linux kernel, which get I/O request from the queue and make the device complete I/O operation. Linux awakens the real-time thread, and the real-time thread resumes to work. The result is returned to the real-time thread by the I/O interface module. Figure 2 shows how these threads cooperate.

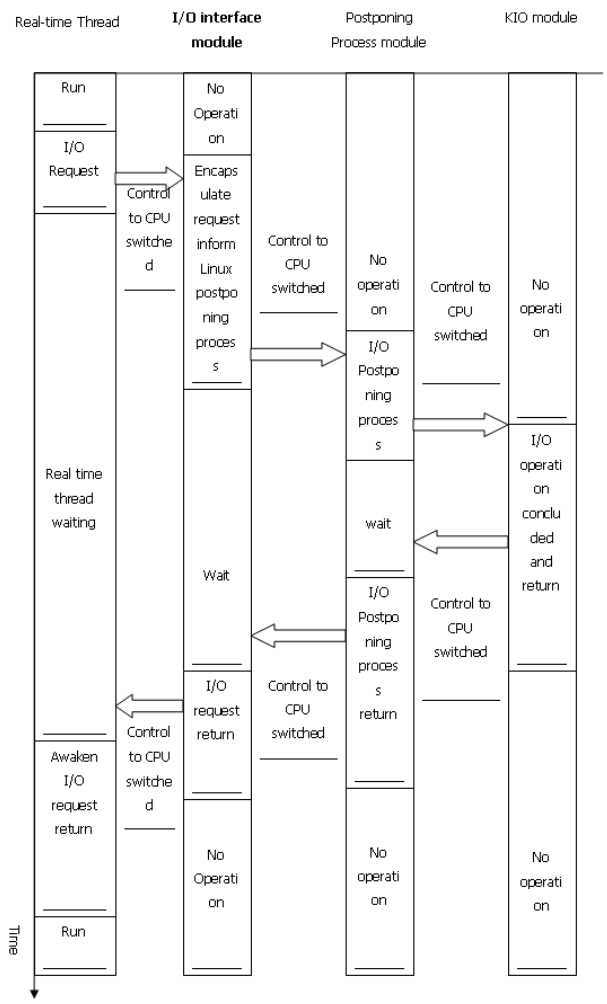


FIGURE 2: *Thread Cooperation*

The interfaces provided by I/O module are presented bellow:

int rtlio_creat(char *pathname, mode_t mode)

rtlio_creat is used to create a new file.

int rtlio_open(char *pathname, int flags, mode_t mode)

rtlio_open is used to open or create a file. It will return a file descriptor if it is successfully called.

int rtlio_close(unsigned int fd)

rtlio_close is used to close a file. *fd* is the file descriptor.

ssize_t rtlio_read(unsigned int fd, void *buf, size_t count)

rtlio_read is used to read a file. The data will be read from the file and copied to the buffer pointed by *buf*. *count* is the length of the read data. If it is successfully called, the byte number will be returned, otherwise 0.

ssize_t rtlio_write(unsigned int fd, const void *buf, size_t count)

rtlio_write is used to write date to a file. Its very similar to the function *trlio_read*.

loff_t rtlio_lseek(unsigned int fd, off_t offset, int whence)

rtlio_lseek is used to set the file position indicator for the next I/O operation. The position is *offset* bytes from *whence*. The value of *whence* can be SEEK_SET, SEEK_CUR or SEEK_END , which correspond to the beginning of the file, the current position in the file, or the end of the file, respectively.

int rtlio_ioctl(unsigned int fd, unsigned int cmd, void *arg)

rtlio_ioctl is used to send command to a device. The *fd* is the file descriptor of the device file.

int rtlio_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)

rtlio_fcntl is similar to the function *rtlio_ctl*.

Currently, the thread which calls any function above will be blocked.

3 Implementation of RTL-IO

As we have mentioned, the TLSF algorithm is used to develop the dynamic memory management module. Apart from what have been introduced, there is still something to be accomplished: First of all, we must implement the initiation and destruct function of the memory pool, and it is necessary to develop some basic bit logic functions to make the calculation of indexes of the arrays of 2 levers more efficient.

We use *bigphysarea* patch, which is not included in the standard Linux kernel. It preserves some memory when system boots and users can use the interface provided to get the preserved memory to initiate the memory pool.

Four bit logic function are implemented as follows: *ffs* searches the first bit that is 1, while *fls* searches the last one. *ffz* searches the first bit that is 0. They all process from the most significant bit to the least significant one. *set_bit* and *clear_bit* are used to set and clear a bit, respectively.

The index of first lever separating array f and the index of second lever array s can be figured out using Formula (3) and (4):

$$f = \mathrm{fls}(size) \quad (3)$$

$$s = ((size \wedge (1 << f)) >> (f - SLI)) \quad (4)$$

We use the *Boundary Tag* [7] technique introduced by Knuth to implement the memory block dividing function. *Boundary Tag* is some bytes to save the boundary information of the memory block, the programmer can divide or merge the memory blocks conveniently using these tags. The pre-condition is

that the block must be bigger than the tag. Its drawback is the tag will consume part of the memory, generally 16 bytes. It is quite wasteful when requested block is smaller than 16 bytes.

The I/O request queue consists of a pointer array and a bitmap. There are 32 elements in the pointer array. Every element corresponds to a real-time thread priority and points to a list, which includes the I/O requests of the corresponding priority. We added a bitmap consists of 32 bits to indicate whether an element of the pointer array is in use.

The postponing module processes the I/O request by using the postponing mechanism provided by the Linux kernel. We use the schedule queue or the kernel thread to implement the postponing module. The process of both the schedule queue and the kernel thread is similar: First of all, call wait_rtlio_request to wait for the I/O requests, then call *ffs* to find out the request who has the highest priority and delete it from the list, at last call the corresponding functions in the KIO module according to the type of the I/O request, after the operation exits, make the thread who is waiting the result of the I/O operation awaken.

We use the virtual file system lever services to implement the KIO module. All the I/O services provided by virtual file system can be called via the file_operation structure. The problem is that all these functions verify the address space to make sure user-mode processes can not access the kernel space. We bypass the verification through the following code:

```
old_fs = get_fs();
set_fs(KERNEL_DS);
ret = filp− > f_op− > read(filp,
buf, count, &filp− > f_pos);
set_fs(old_fs);
```

4 Experiment Results

We use a computer as the testing platform whose CPU is a PIII at 450MHz, memory capacity is 256MB, hard disk 40GB. The Linux kernel version is 2.4.20, The RTLinux version is 3.2-pre2. First we do an experiment to test its I/O transfer performance on the condition that there is only one thread running, and then multithreads running.

The computer was made to read 64MB in order from a hard disk or write to the hard disk to test the performance on the condition of single thread. We tested RTFIFO, the task queue implemented RTL-IO and the kernel thread implemented RTL-IO, for every method we tested 5 times and then calculate the average value of I/O read/write speed. Table 1 lists all the data of performance.

	RTFIFO	RTL-IO (task queue)	RTL-IO (kernel thread)
I/O write speed (MB/s)	27.4	28	28.8
I/O read speed (MB/s)	29.3	30	30.7

TABLE 1: *Single Thread*

As we can see from Table 1, performance of these 3 methods is at the same lever. RTFIFO needs several times of duplication of the I/O data and context switch between the user mode and the kernel mode when processes I/O request. RTL-IO executes completely in kernel mode, as a result the data can be directly accessed by device driver, and the data duplication is needed only once. Nevertheless, for every process of RTFIFO, context switch is not needed, thus to some extent counteract the benefits from less duplication of the data. Consequently the difference between the performance of RTFIFO and RTL-IO is not obvious.

The task queue is shared by all the module in the Linux kernel and all the tasks in the queue are processed serially. As a result, the performance of RTL-IO may probably be affected when several tasks are being processed. Compared with the task queue, if we implement RTL-IO with the kernel thread, the affect from other tasks will be weaker, but the implementation is more complicated.

We made several real-time threads to read data from or write data to the disk at the same time to test the performance of RTL-IO on the condition of multi-thread. The calculation of the data is similar to the up-mentioned one. The speed data of RTL-IO using the task queue and RTL-IO using kernel thread is almost identical, so we did not list the data of the 2 methods respectively.

	2threads	3threads	4threads
RTFIFO speed (MB/s)	32	37	36
RTL-IO speed (MB/s)	41	43	43

TABLE 2: *Reading speed of RTFIFO and RTL-IO*

	2threads	3threads	4threads
RTFIFO speed (MB/s)	29	28	21
RTL-IO speed (MB/s)	42	41	41

TABLE 3: *Writing speed of RTFIFO and RTL-IO*

As is shown in Table 2 and Table 3, RTL-IO gains an advantage over RTFIFO on condition of multi-threads, especially when a lot of threads are writing data to the disk. This is because the real-time thread must cooperate with the user mode program in RTFIFO to process I/O request, but RTLinux provides no service for them that can be used to communicate with each other. As a result, user mode program can not inform real-time thread its status while the latter has to read the buffer to see whether the requested data has been provided by the user mode program and the busy waiting is probably to occur. Thus RTFIFO is made less efficient. Moreover, the more thread running, the less efficient RTFIFO will be. In RTL-IO, the real-time thread and the kernel thread or the task queue are running synchronized, consequently busy waiting will never occur. RTL-IO is more efficient than RTFIFO.

5 Summary

In this paper we describe a newly implemented I/O method for RTLinux. The main difference compared with previous related works is that we present an I/O method supporting the dynamic memory management and the synchronized concurrency of the real-time and non-real-time threads. Our experiment results demonstrate that our I/O system is more efficient than RTFIFO.We have successfully applied RTL-IO in the Cyber data acquisition simulation system.Through this way, the majority of the code of Cyber data acquisition system need not to be modified, the code that need to be modified is just calling the interface provided by RTL-IO. The development is obviously simplified and the cost of the maintenance also decreases.

References

[1] Kevin Morgan, 2001, *Preemptible Linux: A Reality Check*, MontaVista White Paper.

[2] Yu-Chung Wang, Kwei-Jay Lin, 1999, *Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel*, Proceedings of the 20th IEEE Real-Time Systems Symposium, pp246–255.

[3] Michael Barabanov, 1997, *A Linux-based Real-Time Operating System*, A Master Thesis. New Mexico Institute of Mining and Technology, Socorro, New Mexico.

[4] Cort Dougan Matt Sherer, 2003, *RTLinux POSIX API for IO on Real-time FIFOs and Shared Memory*, Finite State Machine Labs.

[5] Alejandro Lucero, Vicente Esteve, Ismael Ripoll and Alfons Crespo, 2004, *IDE Driver and RTLFS system for RTLinux*, Sixth Real-Time Linux Workshop, pp65–73.

[6] Miguel Masmano Tello, Ismael Ripoll, Alfons Crespo, Jorge Real, 2004, *TLSF: A New Dynamic Memory Allocator for Real-Time Systems*, 16th Euromicro Conference on Real-Time Systems, pp79–88.

[7] D.E. Knuth,1973, *The Art of Computer Programming, volume1: Fundamental Algorithms*, Addison-Wesley Reading, Massachusetts, USA.

Real Time Management for P2P Resources on Mobile Devices

Xiaoyun Chen, Lei Wang, Jingchun Zhang, Yu Zhou, Yong Tang, and Minghui Mo
School of Information Science and Engineering
Lanzhou University, Lanzhou, P.R China
lwang@lzu.edu.cn

Abstract

According to 3G standard, the communicating bandwidth of mobile devices is 2Mbps, which makes it possible to share P2P resources among mobile devices. BitTorrent use pure P2P structure to share resources among PC. Howerer, mobile devices have very low computing and memory capabilities. In this paper, a new method designed as Real Time Estate management Server (RTES) is found to realize sharing resources among mobile devices. Based on the operating system of RedHat Linux AS3 Update4, the RTES collects real time on-line estates of the mobile devices by an on-line estate management module. It employs an improved algorithm named Mobile Devices Resources Selecting (MDRS) to realize pieces and addresses selecting of mobile devices' downloading. The RTES manages the process of mobile devices' resources downloading. The simulating results indicate that by adjusting the weight of valuables of MDRS algorithm, in general situation, the average response time to the mobile peers' request is within 90ms, the situation of the mobile peers less than 3000. This proves the system is real time and effective.

1 Introduction

P2P resource sharing system has been more and more popular network system[1]. The resource sharing system based on P2P always exists among PCs, it is different from the file dispensing system based on FTP or HTTP. P2P resource sharing system has been popular accepted because of its multiple-peers downloading, rich resources, service stability, such has come more famous systems: BitTorrent, Napster, Gnutella, Kaza and Maze[2][3][4]. P2P system can be divided into two kinds according to its topology structure: mixed and pure P2P, the mixed P2P system adopts the index server to find the peers, for example: Napster; while the pure P2P system adopts the message broadcasting among peers, for example: BitTorrent.

With the development of micro-electronic technology and 3G's commercialization, the embeded software developing based on mobile terminal device and the developing of mobile increment service platform 3G oriented will face up new challenges and opportunity[5]. According to 3G standards[6], the communication bandwidth of mobile devices will achieve 2Mbps, such that make it possible to realize the P2P resources sharing among mobile devices.

This paper mainly focuses on the implementation strategy of P2P resources sharing on mobile devices. There exist the following problems: on the one hand, The mobile devices can not broadcast messages frequently because of the limitation of its computing capability and storage capability; on the other hand, it can not provide long time stable service because of battery capability. Taking the above problems into consideration, we try to realize the P2P resource sharing of mobile devices through Real Time Estate management Server(RTES) .RTES mainly include the following components: Do Seed module, Build Index module, Query module, Downloading Process Control module, Real Time On-line Information Collecting module. RTES is built on the general purpose Linux operation system, by constructing the real time management components, it can manage and control the mobile devices in time.

The paper is organized as the following: In section 2, we give a full introduction to the real-time estate management server; In section 3, we introduce the implementation strategy of the real-time components detailedly, including the gathering of the mobile device real-time information and the management of the downloading process; In section 4, we adjust some parameters that are used by real-time control units by some experiments to validate the real-timed and feasibility; In section 5, we give the conclusion and the future work; Lastly giving the thanks to the individuals and enterprise who have given us some help among developing this system.

2 System Description

2.1 Software Overview

The system used a mixed P2P model. We design it refer to the disposal logic of BitTorrent. The whole software architecture is given in Figure 1:

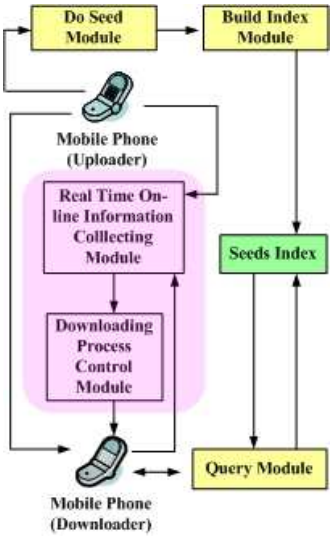


FIGURE 1: *The whole Software architecture of RTES*

- Do Seed Module: The module of Do Seed is designed like the strategy of BitTorrent: user initiatively reports their resources. With listening from the server side socket, RTES listen the users' do seed requests. When a user sends a message of do seed from his Mobile Phone (MP) through its client socket, the RTES receives the message includes: file name, file type, file size, telecom carrier of the MP, the pieces information of the file, and then save these messages into seeds file.

- Build Index Module: With the seeds information from the Seed Do module, the Build Index module mainly achieve segment the words of the seeds file and build forward and offset index of the seeds file. We save the indexed results as seeds index for searching requests[7].

- Query Module: Users will edit the searching keywords from their query windows. RTES receives the searching requests through socket. The Query Module will segment the keywords and search the seeds index file to find the direct seed information. As a query result, RTES will return 10 seeds information to each MP.

- Downloading Process Control Module: Users select a resource from the query results for downloading. RTES will receive a downloading request from MP by server socket. This request includes all pieces messages on the request MP. RTES will employ a resource selecting algorithm to select pieces and addresses for MP(Downloader), and send the selected results to the downloader MP.

- Real Time On-line Information Collecting Module: This module exits a time alarm. RTES listens on all on-line MPs(these MPs include: still on line after do seed, MPs on downloading, still on line after downloading) on-line message. If the MP did not send on-line message after 10 minutes, then we can conclude that the MP was off-line. On RTES, this peer information would be deleted.

2.2 The Developing and Runtime Environment of System

The RTES runs on the environment of 3G network, so there must exits connection service between mobile carriers and Internet. In this system, we designed a gate way to translate the protocol between the mobile devices' communicating protocol and Internet. The gate way can also resend the data package between MPs and RTES. Thinking of different mobile carriers would use different 3G standards(includes: TD-SCDMA, CDMA2000, WCDMA), we need to build absolute gate way for every mobile carries, for instance: China Mobile gate way, China Unicome gate way and China Telecom gate way. With these gate ways, RTES will only think of the protocol of Internet. The position of RTES in network is shown in Figure 2:

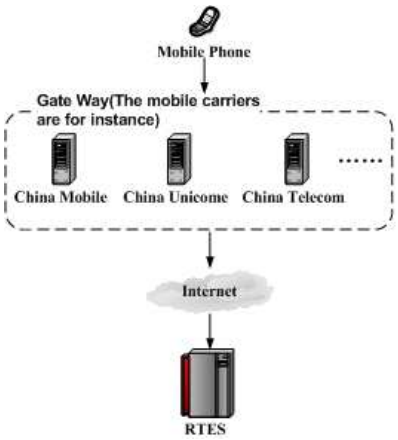


FIGURE 2: *Position of RTES in Internet*

Because of network communications very frequently, RTES would need to resolve many real time problems. We use general purpose linux operation system to settle these real time transact problems.

At last we run RTES on the operation system of RedHat Linux AS3 Update4. The developing environment was: Fedora3 + eclipse-SDK-3.1.2-linux + j2sdk-1_4_2_12-linux-i586 + org.eclipse.cdt-3.0.2-I200602071447-linux.x86. These are all open source software and follow the rule of GPL. RTES was coded by C++ language.

3 Real Time Process Manage and Control

To realize P2P resources sharing on mobile devices, RTES must update and manage the real time estate of the MPs of uploader and downloader. These estate information mainly includes: real time on-line estates, pieces of downloading process from downloader and pieces of uploading process from uploader.

3.1 Real Time Estate Collection and Management on Mobile Peers

The purpose of collecting the real time on-line estate on mobile device is to let RTES know which peers are still on line and can apply a whole or some part pieces resources for other peers to download. In this system, we use the technology of multi threads to set server socket on RTES. The server socket listen on all peers which need to report their real time on-line estates. These peers includes: users on downloading and users who apply the whole resources. A alarm was set on RTES. If a peer do not report its on-line estate after 30 minutes, RTES will regard this peer as a off-line peer and delete the information of this peer. When the off-line peers initiatively connect RTES on the next time, RTES will establish the peers information again. The on-line message was defined as: "03:on-line". RTES assigns threads by thread executor, the algorithm if described as Algorithm 1. The assigned threads listen on the mobile devices, the listing process is described in Algorithm 2.

```
%Algorithm 1: using a thread executor
%ThreadedExectutor executor; //thread executor
%While(1){
%  int fdConn = accept();
%  Task task(new MyThread(fdConn);
%  executor.execute(task);
%}

%Algorithm 2: Mythread.cpp, thread task
%void run(){   //thread execute
%  signal(SIGALRM, handle);
%  loop alarm(30), if off line delete peer,
%  else flag =false;
%  while(1){
%    string sRecv = receive();//receive message
%    int iType = parseType(sRecv);//parse type
%    switch(iType){
%    //online message
%    case 03: flag = true; break;
%    }
%  }
%}
```

3.2 P2P Downloading Process Management on Mobile Peers

In this system, we use a strategy to cut pieces of a whole resource file. We defile 256KB as a piece. Every resource file can be cut into different pieces according to this rule. During the process of downloading, each sub piece can be downloaded from different peer. Thinking of the computing capability of mobile devices, we set a mobile device can only download less than 3 different pieces from 3 different peers on the same time.

After the mobile devices execute the query operate, mobile peers will submit the resource information to RTES, or during the downloading process, when a mobile peer finished a piece downloading, the mobile peer will request for other pieces downloading. When RTES receive these requests, it will employ a resource selecting algorithm to select a piece for mobile peer to download. The resource selecting algorithm will also select the object peers(in this system, we offer 3 best peers for MP) from the table of on-line peers for mobile peer to download a piece. The pieces selecting process mainly satisfy the following rules:

- The first piece randomly selected.
- From the second piece, select a least piece exits in the whole P2P network. This rule is called least resource first.
- During the least resource first selecting process. If two or more pieces satisfies the least rule, RTES will select the piece nearest to the end of the resource file.

Thinking of service instability offered by mobile devices, in the process of selecting the downloading address we think of these factors may affect the speed of downloading: if the mobile devices (uploader and downloader) are from a same mobile carrier (we name this factor "ca") or not, if the two peers are already connected by a socket (so) or not, the numbers of downloader from the uploader's downloading queue (dn), the time of updating service duration(tm). We set some weights for these factors: W1, W2, W3, W4. The BestPeer can be calculated by:

BestPeer=f(ca,so,dn,tm)=
W1*f1(ca)+W2*f2(so)-W3*f3(dn)-W*f4(tm)

RTES will return user 3 best peers to download a piece. The mobile device will select the first peer for downloading. If the selected peer could not reach or the peer off lined during the downloading process, the mobile device would connect to the other peers. In section 4, we will give a group of best weights for these factors we tested. The mobile device resource selecting algorithm is described in Algorithm 3:

```
%Algorithm 3: mobile device resource selecting
%string sRecv = receive();//receive MP message
%int iType = parseType(sRecv);//parse type
%string sRecvStr = sCliStr; //state of MP
%switch(iType){
%  case 04:      //request for download
%    //pieces select
%    sSend = BestPiece(sCliStr);
%    //peer select
%    sSend.append(BestPeer(sCliStr).c_str());
%    break;
%}
%send(string sSend);//return selectted results
```

4 Experimental Analysis

To validate and test RTES, in this paper, we simulate the network bandwidth and computing capability of mobile devices to design and code a client simulate software. This simulated client mainly includes such functional modules: User Do Seed module, User Do Query module, Parse and Buffering the Information of Seeds module, User Report Real Time Estate module, User Upload resource module and User Download Resource module. To adjust the weights of valuables in Table 1, we completely tested the RTES, the results is also shown in Table 1.

No.	W1	W2	W3	W4	Peer Number	Response Time
1	0.3	0.2	0.3	0.2	2000	66.44
2	0.3	0.2	0.3	0.2	3000	87.37
3	0.3	0.3	0.3	0.1	2000	63.21
4	0.3	0.3	0.3	0.1	3000	80.95
5	0.3	0.3	0.2	0.2	3000	82.57

TABLE 1: *Different valuables of weight, RTES's average response time to MP*

From Table 1, we can see that: in situation of group 3 and 4, the average response time from RTES to mobile devices is quite short. Hence, we can conclude that such factors affect RTES bigger: whether or not a same mobile carrier, whether or not already on socket connection, numbers of downloading queue on uploader peer. Synthetically, RTES's average response time to the mobile peers' request is within 90ms, the situation of the mobile peers less than 3000. This proves the system is real time and effective.

5 Conclusion and Future Work

Via developing RTES, we find a new method to settle the problem of P2P resource sharing on mobile devices. This work will bring a new developing sky and application foreground to mobile increment service. In the later research and developing works, we will mainly focus on the following tasks:

- Go on to optimize this system and look after the mode of business usage.
- Optimize the client software. Develop embedded client software combined with real 3G mobile phone products.
- Search for the method of explanting the client software to PDA, STB and other mobile devices.

6 Acknowledgments

While researching and developing this system, we received some help from Guangzhou 3G Network Technology Co.Ltd and some experts from other trades. Thanks all of them.

References

[1] M. Ripeanu, 2001, *peer-to-peer Architecture Case Study: Gnutella, Network*, University of Chicago Technical Report, TR-2001-26.

[2] Gnutella Homepage: http://gnutella.wego.com/.

[3] Maze Homepage: http://maze.pku.edu.cn/.

[4] BitTorrent Homepage: http://bt.5qzone.net/.

[5] Stoica I, Morris R, Karger D, et al, , *Chord: a scalable peer-to-peer lookup service for Internet applications*, PROCEEDING OF ACM SIGCOMM, pp149–160.

[6] 3GPP Homepage: http://www.3gpp.org/.

[7] Li Xiaoming, Yan Hongfei and Wang Jimin, 2004, *Search Engine: Principle, Technology and Systems(in chinese)*, Beijing: Science Press, 7-03-014633-6.

Embedded Real-Time Linux on Chip: Next Generation Operating System for Embedded System

Wei Hu, Tianzhou Chen, Bin Xie, and Qingsong Shi
College of Computer Science, Zhejiang University
Hangzhou, Zhejiang, 310027, P.R.China
{ehu, tzchen, xiebin, zjsqs}@zju.edu.cn

Abstract

Real-time Linux is used in embedded systems because the source of Linux is openly and freely available. But current real-time Linux is often an integrated software platform, including all functions together. They store in flash or hard disks, loaded into RAM on board when the system starts. But it has some disadvantages when it is used for embedded systems. In this paper, we present the embedded real-time Linux on chip, which contraposes these disadvantages to make optimization. Embedded real-time Linux on chip proposes a new architecture for embedded system with SOC based on Linux. The fundamental principles for embedded real-time Linux on chip, which is a module-based architecture, are: the kernel runs in SRAM minimum and the scheduler of modules is optimal. Embedded real-time Linux on chip architecture consists of two main parts: Microkernel on chip and Module Management on flash. Microkernel and all the modules are stored in flash on chip. Microkernel will be loaded into SRAM on chip when the system starts. When a module is needed, the management module is called by module scheduler in microkernel and this module will be loaded into SRAM. Our experimental results show that it makes Linux more efficient for embedded systems and potential for good real-time responses.

1 Introduction

In recent years, embedded systems have been more and more popular and hardware of embedded systems becomes more and more powerful and sophisticated. The embedded systems have provided increasing sophisticated functions. Rich peripherals are provided to give users novel services. And GPS, Cameras, USB, SD card, etc. are used as extended devices for the embedded systems. In addition to the rich peripherals, software functions in the embedded system increase rapidly. And the embedded systems could be used in the following fields: digital household appliances, mobile terminals, automobile electronics and digital instruments.

The embedded systems demand services of a sophisticated, state-of-the-art operating system [6].According to the operating system design concept which comes from the desktop operating system and limited by the hardware, current embedded operating systems such as Embedded Linux [1], Symbian [2], Palm [3], and Pocket PC [4] and so on, all reside in SDRAM. There exist many constraints if they are in SDRAM especially real-time responses. Linux is open source operating system. Generally speaking, Linux is an integrated software platform, including all functions together. Linux offers powerful and sophisticated system management facilities, a rich cadre of device support, a superb reputation for reliability and robustness, and extensive documentation [6]. But it has some disadvantages when it is used for embedded systems. When Linux is used for embedded systems, it has to be improved to satisfy the real-time requirements.

SOC technology provides new powerful features for the embedded hardware. Chip manufacturers have integrated much hardware into single chips, such as SRAM, Flash and WLAN. With the help of the progress of chip manufactures, more and more hardware is intended to be integrated into chips. Sophisticated software architecture is needed to adapt to this progress and take advantage of SOC. SRAM on chip, which is one of the new features collaborating with cache is popular and many high-end embedded processors, such as Intel PXA 27x family [5], have covered this feature to facilitate the system developer to improve the performance of the whole embedded system. SRAM on chip is at least 10 times

faster than the normal DRAM, and proper use of SRAM on chip can give the whole system performance a large rise. It is known that the speed of SRAM on chip is increased by 60% a year versus only 7% a year for DRAM [7]. Though the SRAM on chip is important to improve the performance of embedded systems, few of the embedded operating systems have mended to utilize this feature, and Linux is not the exception.

In this paper, a new architecture is presented. The embedded real-time Linux on chip is designed for the embedded systems based on the SRAM on chip. The embedded real-time Linux on chip is stored in flash on chip and loaded into SRAM on chip when the system starts. Thus the whole operating system will reside in SRAM on chip.

The rest of the paper is organized as follows. Section 2 describes architecture of the embedded real-time Linux on chip. Section 3 gives the experimental environment and results. Section 6 concludes.

2 Architecture of the embedded real-time Linux on chip

Commonly the embedded operating systems are stored in Flash or some other external storage. And these traditional embedded operating systems have many functions integrated together even if they are embedded operating systems with a microkernel because there are much memory that can be used for this kernel. Meanwhile, because SRAM on chip is at least 10 times faster than the normal DRAM, the performance of the embedded Linux will be improved much if the SRAM on chip can be used properly.

However, the size of SRAM on chip is too small compared with the normal DRAM to place so many data and instructions in SRAM on chip. Thus we have to design the embedded real-time Linux on chip based on the following principles:

It must ensure that this Linux kernel can run in SRAM. Because of the size limitation of SRAM on chip, the kernel has to be designed as small as possible. Thus some parts in traditional embedded Linux must be taken from the kernel and the remainders have to be cut down or modified in order to make all parts of OS able to be contained in the kernel. The most important designing principle is minimum and optimal. This new Linux kernel will be a microkernel named after SMK (SRAM MicroKernel).

According to the foregoing design principles, we design six parts in the microkernel: Task Management, SRAM Management, Power Management, Resource Management, Module Scheduler and Security. The architecture is shown in 1.

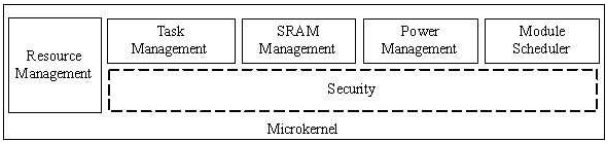


FIGURE 1: *Architecture of embedded real-time Linux on chip*

Commonly microkernel is stored in flash on chip. When system starts, microkernel is loaded from flash on chip.

2.1 Task Management

Task Management is responsible for processes and threads scheduling. Task Management is designed to be able to schedule tasks by different policies which are used mixedly in the embedded real-time Linux on chip.

The most important policy is to support real-time responses. Because of the gap between main memory and CPU, embedded operating systems for real-time are difficult to design and implement. With the evolution of CPU, this gap will be much broader. In memory hierarchy, cache mainly based on SRAM memory is used to reduce the speed gap between CPU main memory and other external storage. The same policy is used by SMK to narrow the gap. Scheduling in SRAM will obtain faster responses to the real-time tasks. The scheduling algorithm is real-time scheduling algorithm based on set division of resource for embedded operating systems [8, 9].

The traditional embedded systems like to place more tasks in memory for scheduler. But not the same to the traditional embedded systems, SMK only places enough tasks into SRAM to ensure the efficiency of scheduling. The relative security mechanisms can be added into this part if necessary.

2.2 SRAM Management

Now there is a very small size of SRAM provided by the processors. The typical size of the SRAM on chip is 256KB [5]. Thus SRAM Management must provide fine management for SRAMOS.

Different from the common memory management, we divide SRAM into different sections which are allocated to different processes and threads. Three sections are divided: Kernel Section, Real-Time Section and User Section. Kernel Section is allocated to the SMK. SMK runs in this part of SRAM on chip to ensure the security of kernel data and instructions.

Real-Time Section is allocated to the Real-Time tasks. This section is reserved and ready for real-time tasks at any time. User Section is allocated to the

common tasks. Kernel Section and Real-Time Section can use the space of the User Section if necessary by swaping out the code of the common tasks. But User Section can not use the space of Kernel Section and Real-Time Section. Further, different Memory Banks of SRAM are divided into pages to manage.

2.3 Power Management

This part takes charge of power-efficient management. A large number of embedded systems are driven by batteries, not by wired power supply. Efficient power usage is required to increase utilization time. Normal embedded OS makes use of DPM [10, 11, 12] or other optimized algorithms to reduce power consumption. But previous platform architecture must use SDRAM on board as memory. And SDRAM costs power very much. SMK runs in SRAM on chip. The ratio of power consumption for SRAM is expected to be 10% of SDRAM [7]. And the new methods from DPM [] are used in power management of SMK. So power efficiency of new architecture will be more excellent than previous architecture.

2.4 Resource Management

Because SMK is not a powerful kernel compared with the traditional embedded Linux kernel for the hardware restriction, a new mechanism is adopted to provide more functions. SMK will encapsulate the resources on chip such as CPU, registers, and provide virtual resources on chip to an embedded OS, such as embedded Linux, which can run with the SMK concurrently. But this Linux does not know the existence of the SMK and it uses the virtual resources on chip provided by the SMK. Thus SMK should be able to manage the resources on chip.

2.5 Security

To the entire kernel, common security mechanisms are adopted in SMK such as processes protection, thread protection and so on. And In different parts of SMK, security mechanisms are adopted. They can also be considered as a part of security.

2.6 Module Scheduler

One of the principles of Chip OS design is minimum microkernel on chip. It is guaranteed by Module scheduler. Module scheduler switches the useless modules out, switches the required module in and inserts it into kernel. This process is dynamic. In real-time system, modules should be divided into real-time relative and real-time irrelative. Real-time relative modules won't be swapped out.

2.7 Modules Management on Flash

All the other modules will be stored in flash on chip or removable devices. And modules are indexed in B+ tree, so that module could be queried and accessed quickly. The management module is called by module scheduler in SMK. Module scheduler asks for the necessary modules and inserts them into SMK. What's more, modules management could be in removable devices, so that users could change GUI, DBMS or other tools through changing removable cards, e.g. SD cards.

When the system shuts down, the SMK will be stored in flash on chip. As we know, SRAM on chip is not a long-last memory in which data can not be preserved without electricity. If the system shuts down unexpectedly or system crashes, there is not enough time to complete the store procedure. It is really a serious problem.

To solve this problem, we provide a periodical pre-store mechanism named Chip Snapshot. A buffer named Chip Snapshot Buffer (CSB) is opened in main memory. When SMK on chip executes, all of the information existing in SRAM will be transmitted to CSB after a predefined time range, and the system is not busy at that time. Then the content of CSB will be written back to disk as a file named CSBFile. Thus if some emergencies occur, after restart of the system, microkernel will be initialized first. In this process, flash on chip and CSBFile will be checked. If it can not find the right version of the SMK in flash on chip, by comparing the time chop recorded in the flash on chip version and CSBFile version. The CSBFile will be used, the system will restart correctly, and the interrupted processes and threads can continue to execute.

3 Experimental results

Experimental Environment: Our embedded real-time Linux on chip is implemented by modifying the Linux 2.6.x and using the GNU tool chain that produces code for the ARM v5e embedded processor family. And the same code of the Linux 2.6.x is modified for the same hardware and by the same tool chain as the comparison embedded operating system (named ELinux).

First, we test performance on variables in SRAM on chip which means these variables are placed into SRAM on chip. It proves to improve the performance by almost 17%. Table 1 below lists three benchmark programs and some relevant information of them.

Benchmark	Source	Total Data Size	Runtime (cycles)	Description
FFT	UTDSP	4216bytes	475K	256-point complex FFT
FIR	Trimaran	1235bytes	271K	Finite impulse respose algorithm
IIR	UTDSP	1278bytes	114K	4-cascaded IIR biquad filter

TABLE 1: *Three benchmark programs and their information*

The benchmarks FIR was obtained from the trimaran [13] benchmark suit; and FFT and IIR were obtained from [14]. These three benchmarks represent code that would be used in typical applications. The first benchmark, FFT, which has 2 functions and 5 global variables, performs a fast Fourier transform on a 256 point data set. The second benchmark, FIR, which has 2 functions and 4 global variables, is an implementation of the finite impulse filter algorithm. The third benchmark, IIR, which has 2 functions and 5 global variables, is an implementation of an infinite impulse response filter algorithm.

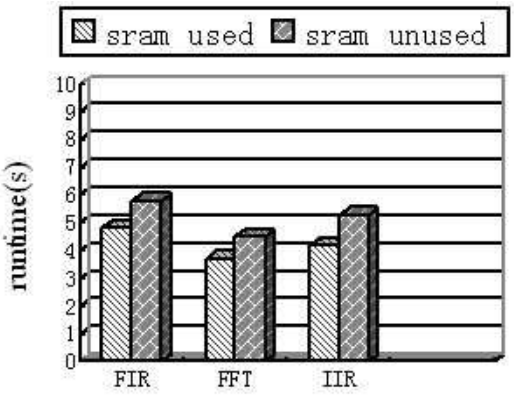


FIGURE 2: *Showing difference in runtime between sram used and sram unused*

Figure 2 shows the improvement between SRAM used and SRAM unused. It indicates obviously that, compared with SRAM unused, the speed can increase by 17% in the situation when SRAM is used for stack and heap.

Then we test the whole speed up of our system.

Application	Source	Lines of Code	Assembly instr.
Patricia	MIBench	563	1237
Mad	MIBench	9856	70452
Ispell	MIBench	10273	26084
pgp	MIBench	32102	48265
FFT	MIBench	342	1013

TABLE 2: *Application Characteristics*

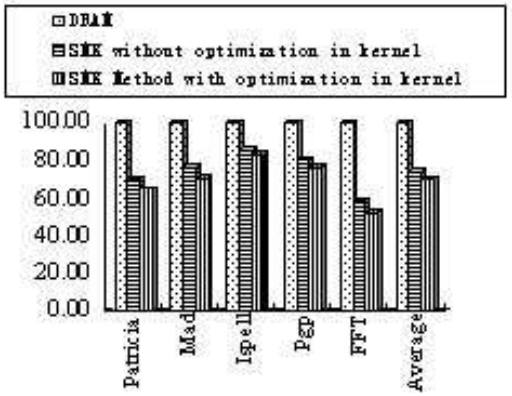


FIGURE 3: *Runtime Speedup*

The benchmark is presented in Figure 3 for three configurations: all-DRAM, our system without optimization in kernel and our system with optimization. Without optimization in kernel, we gain 24% speedup, and achieve 4% addition, namely 28% in total, when considering kernel.

To test the power efficiency of our system, we construct a testbed for it. The prototype system is SimBed [15]. SimBed constructs a processor model which is written in C. Thus it can emulates the M-CORE microcontroller, a low-power, 32-bit CPU core with 16-bit instructions [16, 17]. We construct our testbed whose concept comes from SimBed but damagingly more simple than that.

The object embedded operating system is a cut down version of embedded linux (ELinux) for this test and SRAMOS. The experimental results are shown in Figure 4.

We choose 2ms as the time period. These initial test results show that SRAMOS is 39% lower power consumption than the ELinux.

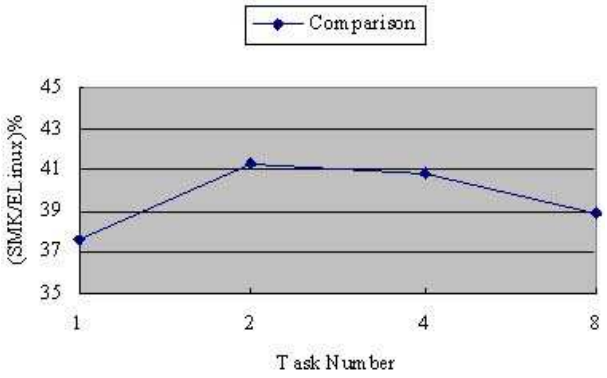


FIGURE 4: *Power performance experimental results*

To get more accurate results we will construct new simulate testbed in the future.

4 Acknowledgements

This research was supported by National Natural Science Foundation of China under Grant No. 60673149.

5 Conclusion and future work

In this paper, we present an embedded real-time Linux running in SRAM on chip. This kernel is able to provide more efficient embedded operating systems. It can respond faster, has more efficient usage of SRAM and lower power consumption. It makes the embedded operating systems smaller and more flexible. But because programmable SRAM on chip is a new technique of hardware, more designs and detail implementation need to be improved. And at the same time, the more accurate simulation and test environments should be constructed for this kernel.

References

[1] Joel R. Williams, 1999, *Embedding Linux in a Commercial Product: A look at embedded systems and what it takes to build one. Linux Journal, Volume 1999*, Issue 66es (October 1999), Article No. 3.

[2] Seizen S, 2005, *Symbian OS v9.1 functional description. Revision 1.1, Feb. 2005*, www.symbian.com/technology.

[3] Eddie Harari, 2000, *Palm Pilot Development Tools*, Linux Journal, Volume 2000 , Issue 73es (May 2000), Article No. 8.

[4] Bruce E.Krell, 2002, *Pocket PC Developer's Guide*, Osborne/McGraw-Hill.

[5] Intel, *Intel PXA27x Processor Family Developers Manual*, http://www.intel.com/design/pca/prodbref/253820.htm.

[6] Rick Lehrbaum, 2000, *Using Linux in Embedded and Real-Time Systems*, Linux Journal, Volume 2000, Issue 75es (July 2000), Article No. 10, 2000.

[7] J. Hennessy and D. Patterson, 1996, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann, Palo Alto, CA, second edition, 1996.

[8] Jiangwei Huang, Tianzhou Chen, Yi Lian, Hongjun Dai, 2004, *Dynamic Power Management of Complex Systems Using Flow Chart and Poisson Process. Proceedings of the Seventh IASTED International Conference on Power and Energy Systems(PES 2004)*, 436-441, Clearwater Beach, FL, USA.

[9] Chen Tianzhou, Huang Jiangwei, Dai Hongjun, *The dynamic power management for embedded system with Poisson process*, Journal of Zhejiang University SCIENCE, Vol.6A Suppl 1.Aug, 70-74.

[10] 2003, *IBM and MontaVista Software: Dynamic Power Management for Embedded Systems.*

[11] Le Cai and Yung-Hsiang Lu, *Dynamic Power Management Using Data Buffers*, School of Electrical and Computer Engineering, Purdue University

[12] G. A. Paleologo, L. Benini, et.al, 1998, *Policy Optimization for DynamicPower Management*, Proceedings of Design Automation Conference, pp.182-187, Jun. 1998.

[13] CONSORTIUM, T. T. 1999. The Trimaran benchmark suite. Available at http://www.trimaran.org/.

[14] University of Toronto Digital Signal Processing (UTDSP). 1992. University of Toronto Digital Signal Processing (UTDSP) Benchmark Suite. Available at http://www.eecg.toronto.edu/.

[15] Kathleen Baynes, Chris Collins and et al,2001, *The performance and energy consumption of three embedded real-time operating systems*, Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems, Pages: 203 210, Atlanta, Georgia, USA, 2001.

[16] J. Turley. *M.Core shrinks code, power budgets*, Microprocessor Report, vol. 11, no. 14, pp. 1215, October 1997.

[17] J. Turley, 1998, *M.Core for the portable millenium*, Microprocessor Report, vol. 12, no. 2, pp. 1518, February 1998.

Towards Runtime Monitoring in Real-Time Systems

Martin Pohlack, Björn Döbel, and Adam Lackorzyński
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
{pohlack, doebel, lackorzynski}@os.inf.tu-dresden.de

Abstract

In this paper we present the state of our work on runtime monitoring for real-time systems: a way to observe system behavior online without unpredictably disturbing real-time properties.

We discuss generic requirements to achieve these properties wherefrom we deduce our monitoring framework architecture. We describe this architecture in detail and discuss several challenges for our implementation called Ferret. We also explain why common operating system primitives, such as message passing or system calls, should not be used for monitoring in the general case and propose a very low-intrusive alternative. We also propose a way of measuring the intrusiveness caused by monitoring.

We applied our technique in different scenarios ranging from simple temporal debugging, resource requirement estimation, gaining behavioral information of peripheral hardware devices to build timing models for providing real-time capable service on top of them, up to whole-system views, such as the interaction between concurrently running system threads. Our research platform also contains a para-virtualized version of Linux that we use to run legacy applications. We discuss how to apply our framework to these components with real-time requirements being only one of several important aspects. We also show how to compare the behavior of our para-virtualized Linux kernel with the behavior of the native variant.

In this work, we demonstrate how to gain a continuous whole-system view by using only Ferret sensors in all layers of our system, starting from the underlying microkernel, basic microkernel programs, real-time applications, and the para-virtualized Linux kernel, as well as Linux user-space applications.

1 Introduction

Computer systems have been growing in complexity for decades, by introducing new software layers and by increasing functionality in established layers. Accordingly, the number of bugs increased as complexity grew. Some problems can be found with static analysis. Runtime monitoring can help where static analysis fails by live peeking into running systems.

As many current systems are faced with these problems, there are many solution attempts. In this work we explore how to apply runtime monitoring especially to real-time systems. Our research platform exhibits some interesting features which influenced our design, in addition to plain real-time properties.

We use a para-virtualized version of Linux [11] to run legacy applications, whenever no real-time functionality is necessary. Even if certain applications have real-time requirements, it is often not the whole application, but only a small part of it. We therefore also support real-time and non–real-time components in one system, interacting with each other [15, 7]. As a consequence, one system configuration might comprise many different software layers with varying requirements. We want to monitor the system in all those layers.

Of course, creating events and storing them must be fast and nonblocking for the monitored components. And last, but not least important, monitoring must be simple to use.

Therefore, we built our monitoring framework Ferret such that it only uses shared memory for transporting monitoring data. This approach does not require any system calls — and as such costly kernel entries — at runtime and also makes Ferret independent of the concrete interface of the layer where we want to monitor (microkernel system calls, Linux system calls, etc.).

The remainder of this paper is structured as follows: In Section 2, we describe related work, followed

by Section 3, detailing our design. We describe our experience with several typical application scenarios in Section 4. In Section 5, we conclude our paper and discuss future work.

2 Related work

In this section, we describe existing monitoring approaches for operating systems and evaluate their applicability to real-time system. Thereafter, we describe available utilities related to monitoring and evaluation of obtained data.

The *Linux Trace Toolkit* (LTT) [19] consists of patches to the Linux kernel that instrument a fixed number of locations inside the kernel. These sensors can be switched on to collect subsystem-related events. A kernel module stores data and provides it to user-space applications for further processing. Our FERRET monitoring framework is similar in that it provides means to generate events, and collect and store data. It is different to LTT, because it does not contain predefined instrumentation for any part of our system environment. However, we can reuse the locations identified by LTT in our Linux version.

Dynamic Probes (DProbes) [12] allow users to dynamically instrument arbitrary locations within an application or the Linux kernel. Whenever a so-called probe point is hit, the framework executes user-defined probe handlers. DProbes require probe handlers to be written in a special language. Dynamic probes have the advantage that they do not pose any probe effect on the system, if monitoring is turned off. DProbes mainly provide means for dynamic instrumentation, whereas FERRET aims at providing an infrastructure for monitoring. Thereby, both means can complement each other.

Kernel Probes (kProbes) [10] introduce the DProbes concept to the Linux kernel. They are dynamically inserted into a running kernel by loading a kernel module. Any location inside the kernel can be instrumented by replacing instructions with a trap event opcode. On x86 architectures, `INT3` is used for this purpose. When such an instruction is hit, the kProbes framework performs all necessary tasks for event generation. It executes handlers, single-steps the original instruction, and finally returns control to the code that raised the exception. In addition to arbitrary kProbes, function-entry probes (jProbes) and function-return probes (kRetProbes) are available. kProbes provide means for dynamic instrumentation in the Linux kernel, which we use to place FERRET sensors.

Sun's Solaris operating system uses *DTrace* for tracing. DTrace also does not have any probe effect if tracing is turned off for dynamic probes. Static probes have no-ops in place if deactivated, which are patched with calls to DTrace trampoline code on activation. In addition to kProbes, DTrace is able to probe user-space applications. Probes are written in a specialized language called D. Although DTrace offers versatile aggregation support, it is not especially suited for real-time systems, for example, probe code is executed with interrupts turned off in the kernel.

Each of these approaches uses system calls to hand over data between in-kernel sensors and user-space monitors. Dynamic probing additionally uses exceptions to handle probes. These means add an additional overhead to the probe effect caused by the instrumentation. We therefore do not use dynamic probes in the real-time parts of our system, but only static probes. On the other hand, dynamic probes are an extremely flexibly way of gaining information about unforeseen problems in running systems. We therefore use kProbes in our Linux kernel version (a non–real-time part) to instrument it with FERRET sensors. We describe the kProbes port in Section 3.5.

In his thesis "Monitoring, Testing and Debugging of Distributed Real-Time Systems" [18] Thane discusses the problem of intrusiveness for systems with embedded software sensors. He argues to generally leave them in the system even for the final deployment. After all testing has been done with the sensors in the system, removing them for the final deployment would be a too big change. Also, by making them part of the system already early in the design process, there is no *additional* overhead involved in having runtime monitoring, so the intrusiveness is (defined to) zero.

While this approach might work and even be required for embedded systems, we target a different class of systems where we can not even enumerate all potentially required sensors. Also, in our scenarios, the applied sensors and their type were quite problem specific. While the single sensors have a negligible overhead, always enabling all of them would slow down the system considerable. We can, of course, use Thane's approach for simple, hard real-time parts of our system, where functionality and monitoring requirements are known beforehand and are static. In this case type, number, and functionality of the sensors can be determined.

Magpie [2] is a toolchain that can, based on observed events of one or several connected systems, extract requests. Requests are typically the unit of interest for human interpretation or performance evaluation, for example, everything that belongs to a single `HTTP` request (potentially crossing machine boundaries on the server side for accessing a database), or everything related to displaying a single video frame. What comprises a request is highly

problem specific and can be defined in schemata. Also other event post-processing is defined there.

Originally, Magpie worked only as a client for Event Tracing for Windows (ETW). It can process traces online and offline, but does not give real-time guarantees. However, online usage was shown to be feasible in a typical business scenario [2].

ETW however, has properties that make it unsuitable for real-time systems. Communication with clients works with a set of buffers that are handed to consumers once they are full; there is no upper latency bound (e. g., if buffers are filled slowly). In [2] Barham and colleagues mention an approximate processing delay of one second for the online version of Magpie. Additionally, ETW notifies event consumers with call-back functions that introduce additional causal relationships between the observed and the observing entity. This infringes the desired property of the minimal probe effect. We avoid this problem in FERRET by *not* synchronously notifying monitors.

As Magpie has interesting properties for finding, building, and post-processing requests, we modified the Magpie toolchain to work with FERRET and adapted its inner workings to our L4 architecture. In Section 4 we present example traces processed with our extensions to Magpie.

Different sources give different numbers about ETW overhead: [13] estimates 1500–2000 cycles and [14] about 1000 cycles. For posting ETW events from user-space, at least one kernel entry is involved. In FERRET we do not pay kernel-entry costs for each event.

3 Design

While designing FERRET we tried to bring in line general requirements for runtime monitoring systems (e. g., low overhead, ease-of-use, versatility, ...), as well as requirements derived from the real-time nature of the target system (DROPS [3]). In this design discussion we focus on the latter properties that are low intrusiveness, low probe effect, or Heisenberg's uncertainty principle for software.

We aim at a software-only solution, that is, we have no additional hardware devices snooping memory busses, CPU cycles, or unusual high precision timers, because we want to deploy our framework easily on standard machines, basically everywhere where our system runs.

DROPS consists of many heterogenous components, including L^{4}Linux and normal time-sharing software on top of it. Also, we have several scenarios where real-time and non–real-time components interact. Therefore, we want to be able to monitor all types of components and their interaction.

Many systems implement monitoring for user-level components with the help of system calls that care for sensor buffer management, taking timestamps, ordering, and atomic execution of monitoring code.

We have not chosen this approach for several reasons. First, it includes the additional overhead of a system call (kernel entry). Second, some components in our system cannot and must not directly interact with the microkernel using system calls (e. g., L^{4}Linux user-space programs). Third, using kernel primitives for monitoring might simply be too intrusive as it might change the system's behavior we currently want to observe (e. g., scheduling). Instead, we only use shared-memory buffers, which can be completely pre-mapped and pinned, so that monitoring itself only incurs memory access, which should be possible on all environments. Furthermore, we use superpages and cache-line aligned data structures to further reduce sensor overhead.

Instrumented real-time and non–real-time applications can post events by only using memory accesses. A non–real-time monitor can collect these events online or offline. In the offline case, sensor buffer memory must be large enough to hold all events. The monitor waits until the experiment is finished and collects data afterwards. It does not influence the experiment at all in that case. For longer running experiments, the monitor works online and periodically polls sensors and collects events from them. It can process, filter, or store events. Therefore, the monitor can either run with non–real-time priority if there is enough slack time in the system or it can be scheduled as real-time task. In both cases, the monitor runs concurrently to the experiment, but there is no observable relationship that would influence traces, as we refrain from synchronously notifying monitors of the availability of new data. This would be costly in the event creation path and might create new communication relations that we did not have in the unmonitored system. We can calculate the necessary amount of shared memory for storing events if the event rate is known beforehand [16].

3.1 Roles and nomenclature

In the following we define terms used throughout this paper:

Events are assumed to be instantaneous and have a timestamp associated. Events can carry additional payload, where the header typically contains major and minor numbers, as well as an instance identifier. Events are typically posted into sensors but may also be serialized into a

persistent event stream for later evaluation.

Event types describe the payload data layout for a class of events and have a semantic attached as they stand for a certain action in the system. Events are instances of event types.

Sensors are unidirectional means of transportation for events. They are writable for event producers and readable by monitors. Internally, sensors are composed of a shared-memory region with meta-data describing the data layout and identifying the type and instance of a sensor. Additionally, they contain a container area for events.

Probe points are locations in programs where events may be created and posted. Instrumentation places sensor code at probe points either statically or dynamically.

In FERRET, we have three roles involved in tracing, which roughly correspond to the three roles defined in the "POSIX Trace standard 1003.1q" [9] (PTs). We have traced processes (also called traced processes in PTs) as event producers, we have monitors as event consumers (called trace analyser processes in PTs), and we have one sensor directory in FERRET (the trace controller process in PTs). The interaction of these roles is shown in Figure **??**.

In the following, we will discuss the responsibilities of each of these roles in FERRET and discuss differences to PTs.

Traced process With FERRET, the creation of sensors is triggered either by the traced process or by a third party on behalf of the traced process. In each case, the sensor itself is setup and configured in a memory region by the sensor directory that then maps the shared-memory region to the traced process.

Actually tracing events happens by memory accesses into the sensor area inside the traced process. Therefore, the traced process uses inline functions defined in FERRET header files.

FERRET comes with a set of common event types predefined. Customized types can easily be defined by the user as well.

Monitor Monitors use the sensor directory as a directory service to find required sensors either via symbolic names (e. g., `/ferret/l4linux-kernel/syscalls`) or with well known constants. After looking up sensors, monitors register themselves as consumers. The sensor directory gives them read-only access to the corresponding shared-memory region. These two mechanisms (the read-only access and the indirection over the sensor directory) prevents information flow back to monitored processes from the monitors through FERRET.

After acquiring access to all required sensors, monitors can periodically look into their sensors for new events. Evaluation can happen online or offline by storing event streams for later use.

Sensor directory The FERRET sensor directory corresponds roughly to the Trace Controller Process in PTs. It is responsible for creating and initializing new sensors, granting access for other producers and consumers, managing the sensor name space, and the handling of instances. FERRET supports several instances of a traced component running side by side, so, beside the sensor identifier, we use an additional instance identifier to address sensors. Closing of sensors and starting or stopping the whole framework is also handled by the sensor directory.

After setting up a sensor and granting access to producing and consuming tasks, the sensor directory is not involved in the event flow. Filtering of relevant events can be done either in the producer, by just not posting certain events, or in the consumer by ignoring events. At a more coarse-grained level, filtering can also happen by subscribing to specific sensors. FERRET allows for very fine-grained sensor usage, also across processes, such that events might be grouped thematically, rather than by source. In the PTs, event filtering is done by the Controller Process, which is therefore involved in all event transportation.

3.2 L^{4}Linux

L^{4}Linux is a port of the Linux kernel to an L4 environment. It uses L4 kernel primitives as well as the basic software environment (L4Env) running on top of the microkernel to provide the necessary environment for running a Linux kernel. These services include communication primitives, execution contexts, interrupts, memory services and the like. For example, L^{4}Linux queries the memory server for a chunk of memory that it then uses as the main memory.

Internally, the L^{4}Linux kernel is divided into several threads within the same address space. The main thread runs the actual kernel code. L4 requires that threads that want to receive interrupt messages need to attach to an interrupt. Consequently, each interrupt that is used by L^{4}Linux requires an interrupts thread. These threads execute bottom half code and then notify the main thread. Besides the main thread and interrupts threads, spe-

cial L4Linux drivers may start own threads to receive asynchronous notifications. Additionally, L4Linux has a so called tamer thread that is used to serialize access to critical sections in case of contention. There are additional management threads as well as service threads of L4Env.

3.3 Layers

From the monitoring point of view we have four different layers in our System: the microkernel (L4 Fiasco in our case), basic microkernel programs, a paravirtualized Linux kernel, and Linux programs. We support monitoring in those layers and describe this in the following.

For monitoring kernel and system behavior, we use Fiasco's built-in tracebuffer, which can be configured at runtime. We access the tracebuffer as shared-memory region and read events from it containing kernel entries. The tracebuffer supports a set of standard events, such as context switches, inter process communication (IPC), page faults, and so on, which can be selectively activated. The kernel as an event source is sometimes important as we cannot atomically take timestamps in user-space directly before or after IPC or other system calls. Using the kernel here eases accounting. Also, it is very convenient to use the kernel as event source for gaining a general system view as sensors in only one place have to be activated. The FERRET framework provides the tracebuffer as a normal event list sensor to monitors.

Basic microkernel programs, with our without real-time properties, can be monitored with normal FERRET sensors. FERRET uses two L4Env services for setting up sensors: names (name server) and dm_phys (physical memory manager). The sensor placement in the virtual address space is handled by FERRET if the L4Env region mapper is available or it can be specified manually. When instrumenting names and dm_phys themselves with FERRET sensors, we have bootstrapping problems, naturally. In names, we solved this by deferring sensors setup until dm_phys registered at names. For dm_phys we will use a preinitialized sensor to circumvent cyclic dependencies.

We could have prevented those two special cases by reimplementing a lot of functionality of names and dm_phys in the FERRET framework but decided not to do so, as the gains for all this code duplication seemed small.

The current version of the L4Linux kernel is an L4Env program, so monitoring it with FERRET works normally. Additionally, the L4Linux kernel sets up one event list sensor for its user-space programs and pages the sensor's memory into their address spaces on demand. It acts as a proxy between FERRET's sensor directory and L4Linux user-space programs, as those programs are normally not allowed to contact other L4 programs. Another advantage of this architecture is that only one entity manages the virtual address spaces of Linux user-space programs, the L4Linux kernel.

3.4 Sensor types

To address the different problem domains we are facing and to minimize intrusiveness, we provide different types of sensors in FERRET. The most simple sensor is a counter type we call *scalar*. This type is basically used for counting the occurrence of events, for example, the number of deadline misses in a time interval for a real-time task.

The next sensor type is the *histogram*. Histograms can be accessed either as arrays of scalars (using the bin number) or as real histograms with offsets and index scaling. Overflows and underflows are counted additionally. Amongst other things, we use histograms for acquiring the resource usage for various routines in the system, for example, the video and audio decoding steps in our video player Verner [17]. We use these numbers directly for CPU time reservation in Verner. In L4Linux, we use the array mode of the histogram sensor for counting the calling frequencies of system calls [4].

Histograms are not restricted to one dimension, but can have multiple dimensions and layers.

The most general form of sensors are *event lists*. Events can be posted from all positions in the system and can contain arbitrary data. All events are timestamped and can therefore be totally ordered per processor. Using event lists is useful when early aggregation is not possible, for example, if the way to aggregate is yet unknown or if the time relations of events are important.

In the general case, event lists are read from and written to by several parties concurrently, that is, there might be several producers in different address spaces. Our event list sensor is based on the Concurrently Invocable Sensors from [16] and uses lock-free algorithms for achieving synchronization.

We defined a common event header denoting the origin and type of events, followed by custom data. We describe the data layout for the custom parts of all events in the Magpie toolchain (cf. Figure **??**).

3.5 Porting kProbes

The kProbes mechanism in Linux allows to insert trap points at arbitrary positions in the kernel. When the execution flow passes such a point, a

trap is raised and kProbe trap handler functions are called.

Having kProbes available in L4Linux allows us to use legacy Linux instrumentations. The SystemTAP project [5] aims at providing a scripting language to create dynamic instrumentation. Its developers focus on kProbes and there is already a large range of instrumentations for common problems available.

Installing a kProbe works by saving the instructions at the kProbe point to a private area and overwriting the point to be probed with the shortest possible instruction that causes a trap or exception. When the point is hit, the kProbe module calls user defined handlers. Then, the original code, saved in the private area, is executed in single step mode and finally execution is resumed after the kProbe.

The i386 kProbes implementation uses the one byte opcode INT3 as the trapping instruction. However, INT3 is also used to call the Fiasco kernel debugger. We replaced the INT3 opcode in kProbes with HLT, that also causes a general protection fault when called in user mode. HTL is also a good choice as it is definitely not used in the Linux kernel otherwise, except in the idle loop, which we implemented differently in L4Linux. Other alternatives would have been CLI or STI, but L4Linux would then be restricted to run without full I–O privileges, which is convenient sometimes.

We also changed the exception handler within L4Linux such that an exception on HLT is handled by kProbes. Additionally, the instruction pointer of the exception needed to be adjusted, as for the HLT instruction it points *onto* the instruction, whereas for INT3 it points *after* the instruction.

4 Application scenarios

In the following, we describe the application of the Ferret monitoring framework to typical scenarios we encountered while working with our system. We chose the scenarios to highlight different properties of the framework that we think are important.

4.1 Resource usage estimation

The first application scenario is centered around our video player Verner [17]. In [8] we describe how we achieve real-time guarantees for video decoding. In short, we adapt the quality of the post-processing step and thereby change its CPU-time demand. We count the number of deadline misses over the previous n frames and adapt the post-processing quality accordingly. We can either lower the quality (if many deadline missed occurred), keep it the same (if we had only few misses), or raise it (if there were none). We can tolerate several deadline misses because we buffer some frames.

Interesting from the runtime monitoring perspective is that we only need to embed very little sensor code into the application (the time slice overrun handler). Thereby, we can completely separate the adaptation decision from the functional core of the video decoder. We only need a simple scalar sensor in the decoding component.

We also augmented Verner with histogram sensors for measuring the CPU time demand for certain routines, most importantly, the video decoding step, post-processing video frames, and the audio decoding step. We can view the distributions live, while the video is playing for observing system behavior and we use collected execution time distributions for admission and scheduling of the whole video player application.

We also used event list sensors for verifying buffer fill levels between components in the decoder chain and for collecting information about the development of the decoding times over time (or stream position).

4.2 Resource usage modeling

With DOpE [6], Drops has a real-time display component, which can guarantee refresh rates for a requested rectangular area. DOpE keeps track of average and worst-case times for copying pixels from a shared-memory representation to graphics memory. Time estimation for copy routines is currently based on area size (pixel count) to be copied.

To verify this estimation we instrumented the inner copying routing of DOpE and took the time in CPU cycles for copying rectangular areas.

```
for (j = dy + 1; j--; ) {
    /* copy line */
    d = (u32 *)dst; s = (u32 *)src;
    for (i = dx + 1; i--; ) *(d++) = *(s++);
    src += scr_width;
    dst += scr_linelength;
}
```

We compute the time per pixel in place and store the information in a two-dimensional histogram indexed by the width and height of the rectangle. The histogram has also two layers, whereas the first layer contains the accumulated copy time, and the second layer counts the number of occurrences for this width-height combination.

We directly aggregate the information online to minimize the memory load in this experiment as we are a taking huge number of measurements. We create redraw requests with uniformly distributed width and height between 1 and 400 pixels using a small benchmark program. We also measure each point at least 100 times and compute average copy times.

We took measurements on the following two machines.

Machine A has an AMD Duron processor with 1,200 MHz. First and second level caches have 64 byte cache lines. The machine has a 64 kB Level 1 Instruction-Cache (2-way associative), a 64 kB Level 1 Data-Cache (2-way associative), and a 64 kB Level 2 Unified-Cache (8-way associative).

Machine B has an older Intel Pentium-Pro with 200 MHz. First and second level caches have 32 byte cache lines. The machine has an 8 kB Level 1 Instruction-Cache (4-way associative), an 8 kB Level 1 Data-Cache (2-way associative), and a 256 kB Level 2 Unified-Cache (4-way associative).

In the experiments depicted in the Figure 3 we see that the assumption of fixed cost per pixel is a viable approximation for a large range of rectangular sizes. However, we also see diversions in several places, which we discuss in the following:

1. The mountainous area on the left side results from copy operations on rectangles with a small width. The huge increase in copy time per pixel for very short pixel rows probably stems from computing the pixel row addresses for source and destination buffer.
2. There are small trenches parallel to the height axis, corresponding to the cache line size. Copying whole-numbered multiples of cache lines sizes reduces overheads *per pixel*.
3. There is a valley in the front corresponding to the total processor cache size. The measurements depicted in the Figures 3a, 3b, and 3c were taken with a horizontal screen resolution of 1024 pixels, resulting in aliasing effects with the processor cache size. Effectively, every cache color[1] is bound to a small set of pixel columns. This leads to trashing the own cache set when copying areas with more than a certain height, independently of the width of the rectangle.
   Machine A has a data cache of 128 kB (L1 and L2 together as the cache hierarchy is exclusive). Running with a graphics mode of 1024 pixel per line, with 2 bytes per pixel results in a cache-trashing height of 64 lines, computed as follows:
   $$128\text{ kB}/(2\ \frac{\text{B}}{\text{pixel}} * 1024\ \frac{\text{pixel}}{\text{line}}) = 64\text{ lines} \quad (1)$$
   This is also exactly what we see in Figures 3a and 3b.
   Compare Figures 3b and 3d. The most prominent difference between Figures 3b and 3d is that data for Figure 3d was taken with an `800x600` resolution, whereas data for Figure 3b was measured with an `1024x768` resolution. In Figure 3b, the cache trashing line can be clearly seen (parallel to the width axis), whereas in Figure 3d it is gone, as their is no aliasing between cache colors and pixel columns.
   Machine B has an effective data cache of 256 kB (maximum of L1 and L2 as the cache hierarchy is inclusive). Using equation 1 with otherwise equal settings we compute a cache-trashing height of 128 lines that can also be seen in Figure 3c.

For comparison we also ran our experiments with memory type range registers (MTRR) disabled, as this was the initial situation when DOpE was created in the year 2002. We see that enabling MTRRs for the framebuffer area results in an approximately three-fold speedup (compare Figures 3a and 3b). However, the relative differences in the histogram have grown with enabled MTRRs as well (compare the mountainous area on the left with the height of the flat area in the middle and right side), making the fixed-cost-per-pixel assumption questionable.

Also, in the measurements taken on machine B the differences in the copy times per pixel contrast even stronger as depicted in Figure 3c.

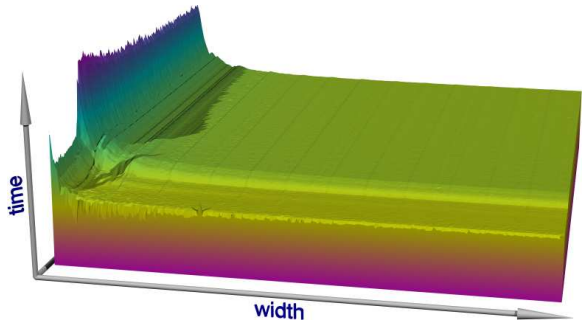
From the experiments we see the typical optimization for throughput (e. g., using MTRRs) hurts predictability and thus might create problems for real-time applications. However, an execution time model based not only on the area but on the length of both rectangle sides seems feasible from the measurements shown. Model calibration to a target machine could happen at component deploy time, or, if only few measurements are required, at startup time. Also, refining the model online seems practical for soft–real-time problems.

4.3 Taming L^{4}Linux

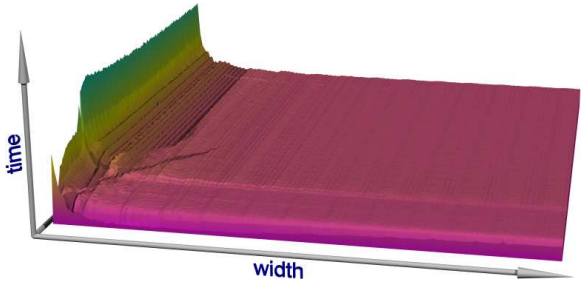
In this section we will describe one specific problem we encountered with L^{4}Linux, its cause, and a way how to identify the problem now and in the future. This shall demonstrate the utility of our framework for finding timing bugs in such complex components as operating system kernels.

Native Linux uses CLI and STI instructions to protect critical sections by disabling and enabling interrupts. L^{4}Linux uses a replacement for CLI–STI as it runs without kernel privileges in user-mode where

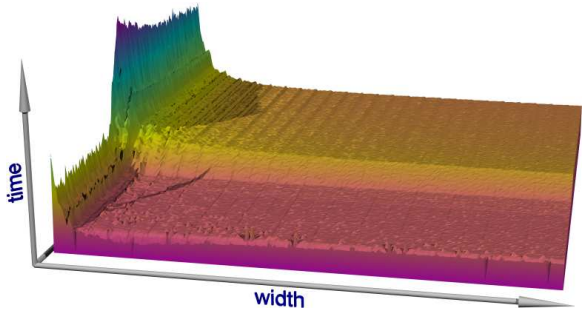
[1] A cache color is the set of cache lines that can cache the same physical address.



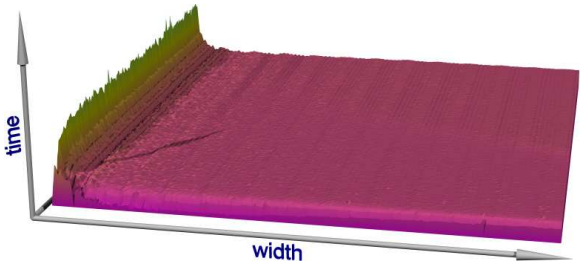
(a) Basic experiment with MTRRs disabled.



(b) For this experiment we used MTRRs to enable caching for the framebuffer region, which results in an approximately three-fold speedup.



(c) Also with MTRRs enabled but on *machine B*, the memory access profile looks quite different.



(d) The same setup as in Figure 3b but the resolution is reducded to `800x600`, eliminating the cache aliasing effect.

Figure 3: Depicted is the time for copying a single pixel to graphics card memory, depending on the width (horizontal axis) and height (axis "into" the paper) of the rectangular area copied. The width and height axes range from 1 to 400 pixels each, the time axis shows relative times for each machine (so you can compare 3a, 3b, and 3d quantitatively, which were measured on *machine A*)

these instructions are not allowed. CLI–STI is replaced with a mutex, which is taken using atomic instructions in the noncontention case. In the contention case, a blocking IPC is sent to one synchronizing *tamer* thread T. T runs on the highest priority inside the L^{4}Linux kernel, thereby it is able to execute its code atomically with respect to all other L^{4}Linux kernel threads. When leaving a critical section and another thread wants to enter the critical section, the leaving thread also notifies T that, in turn, wakes up one waiting thread according to its queuing policy. Again, the underlying assumption here is that by running T on the highest priority inside L^{4}Linux it can run its code atomically with respect to all other L^{4}Linux threads.

L4 kernels have a feature called donation, which optimizes a common communication case, where a client sends a short request to a server, which immediately processes it and returns the result. However, the server in this scenario runs with the time slice *and the priority* of the client while processing the request.

Some L^{4}Linux driver stubs communicate with external servers via IPC, for example, for using external hardware drivers. The external server's worker threads may run on the same priority or even on a higher priority than the tamer thread inside L^{4}Linux as both are separate subsystems.

In our case one L^{4}Linux-internal stub driver thread A was notified by an external thread W running on the same priority as L^{4}Linux's tamer thread. By notifying A, W temporarily transfered its priority to A. As a consequence, although extremely rarely, T was interrupted in its atomic sequence when A itself wanted to enter the CLI–STI critical section.

After identifying this problem we wrote a monitor that detects this problem at runtime. Therefore, we wrapped the tamer's atomic sequence with start–stop events. Additionally we enabled logging of context switch events in the microkernel. The monitor checks for the absence of the following condition: There must never be a context switch to an L^{4}Linux kernel thread (except the tamer thread itself) in-between a start event and a stop event for the atomic sequence. The monitor also keeps a history of the previous n events for later visualization and debugging of the problem. Figure 4 shows such a situation.

After we fixed the problem, this test can now be run after any changes we make to L^{4}Linux. The

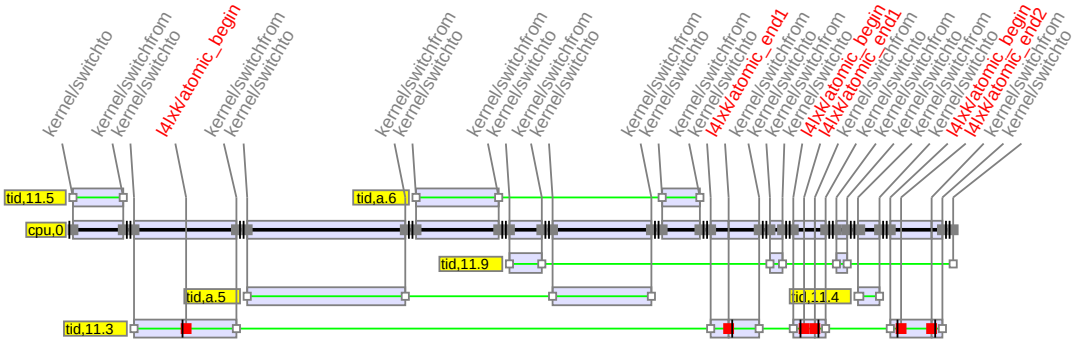


Figure 4: Visualization of the atomicity problem with the tamer thread (11.3). You see three atomic sections wrapped in the events: `l4lxk/atomic_begin` and `l4lxk/atomic_end`. The first of the three sections is interrupted with context switches to thread A.5, A.6 (both in the external driver), and 11.9 (an L^{4}Linux internal worker thread). The context switch to 11.9 violates the atomicity condition as the tamer thread is preempted by a normally lower prioritized thread.

bug would be hard to identify with other means, as it requires a whole-system view as events from several threads and the kernel are required and their exact order matters. Also, the condition is checked completely outside of the L^{4}Linux kernel's address space, which makes the checking immune to, for example, memory corruption by other potential bugs in L^{4}Linux.

4.4 Virtual and native Linux

When (para-)virtualizing operating system kernels, such as Linux, it is important to detect behavioral changes. This applies not only to the functional correctness of the virtualized version but also to non-functional properties, such as speed and memory requirements of the virtualized version.

We compared L^{4}Linux to native Linux running comparably configured kernels. We used kProbes to track control flow inside the Linux kernel. For Drops, we used kernel-level instrumentation to track scheduling information. In native Linux, an additional kProbe was used for this purpose. To reuse the instrumentation code from L^{4}Linux within native Linux, we implemented a Linux kernel module that implements parts of the Ferret framework.

One of our experiments showed a difference in scheduling behavior between L^{4}Linux and native Linux within the `vfork` system call. `vfork` is a special version of `fork` that assumes that only little work has to be done between `fork` and `exec` (or `exit`). Therefore, parent and child can share the same address space (including the stack), which saves the overhead of copying the parent's page tables. The parent process has to be blocked to prevent address space corruption until the child calls `exec` or `exit`.

While observing the behavior of `vfork` in L^{4}Linux, we found, that after the system call has been handled by the L^{4}Linux server, the Fiasco kernel switches back to the parent task before executing the child (see Figure **??**). We further investigated this behavior, because this looked like a serious bug in either L^{4}Linux or the Fiasco kernel. As it turned out, this is only an optimization artifact in Fiasco, never visible to user-space. Fiasco indeed switches to the parent process, but only to find out that it is blocked. The parent process was still in Fiasco's ready queue but flagged as blocked. Fiasco uses an optimization called lazy queuing that does not always remove IPC senders from the ready queue but marks them with a flag. Often, the IPC receiver answers fast and Fiasco can directly switch back to the sender, thereby saving two queue operations. `vfork` triggers the other case, where the parent was flagged and has now to be removed from the ready queue.

Although this bug turned out to be a false positive, this example shows that Ferret can be very helpful in finding and debugging behavioral differences between native and (para-)virtualized variants of Linux. Timing information from the events can also be used to point out performance differences.

When evaluating performance, the overhead induced by monitoring needs to be considered in order to measure correct values. The two main sources of intrusion are:

1. The overhead caused by measuring and storing data, and calling functions provided by the monitoring framework. We used microbenchmarks, executing thousands of monitoring calls in a row, to determine the minimum time that is needed to run these functions. Additionally, we used macrobenchmarks, executing real applications, to measure the average and maximum effect monitoring calls have on the system.
2. The cache misses that are caused because the above mentioned monitoring calls overwrite cached data that is used by the instrumented

applications. To measure the cache overhead, we again conducted micro- and macrobenchmarks and used hardware performance counters to determine the number of cache misses.

By knowing the intrusiveness of instrumentation and the monitoring framework, we are able to reassess obtained timing data and approximate the real performance of the monitored system [4].

4.5 Whole-system view

To illustrate the continuity of the FERRET framework we demonstrate how to follow the control flow from a L4Linux user-space program (`arping`), through the L4Linux server, further on to an external L4 network server (ORe), using different and adequate means to place sensors in the different layers of the system.

To visualize the control flow for this scenario, we used the following sensors:

1. The `arping` program running in L4Linux was manually instrumented to produce events before sending a request, before listening for an answer, and after processing the answer.
2. The L4Linux kernel was instrumented using a kernel module installing kProbes at the system call entry, the `send` function, and the `receive` function of the L4Linux ORe driver stub.
3. DROPS applications usually provide an IPC interface to other components of the system. This interface is typically described using CORBA IDL. The DROPS IDL Compiler (Dice, cf. [1]) translates this description into IPC code. We instrumented the server-side IPC code of the ORe network switch using a tracing plugin for Dice.
4. Information about context switches were obtained from the Fiasco kernel using events written to the Fiasco trace buffer.

Extracts of the resulting trace can be seen in Figure 6. Figure 6a shows how the ping request is sent from a user-space task to the L4Linux server (tid,12.4), which then hands the packet over to the ORe network server. Figure 6b shows the answer coming in. The ORe IRQ thread (tid,8.5) notifies the server thread (tid,8.8) and the server thread then calls back the already waiting L4Linux driver stub (tid,12.15). In Figure 6c we see the ping application doing a `select` system call on the socket and finally receiving the ping reply.

The example demonstrates that with the help of FERRET we are able to instrument and monitor applications at different system layers and use the obtained data as a basis for evaluating whole systems. The demonstrated approach is not constricted to our research system but may be applied to other layered architecture as well. For example, other virtual machine setups will require event from the low-level hypervisor, medium-level virtual machine monitors, and high-level guest operating systems and applications as well. FERRET provides a framework to create, transport, and process these events.

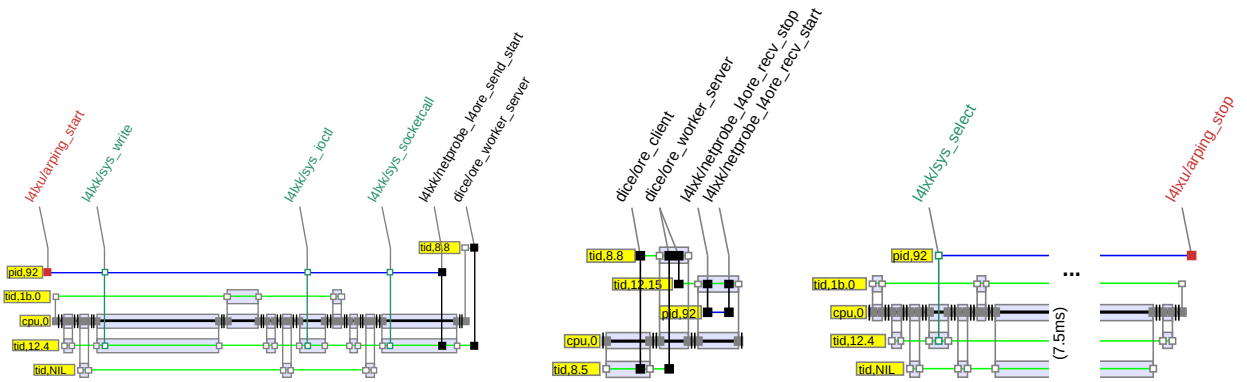
5 Conclusion

We created a monitoring system, solely based on shared-memory regions, not using any system calls in normal use. This results in very fast and predictable logging suitable for real-time and high-performance applications. We also achieve independency from the actual operating system the monitored process runs in, indicating the applicability in virtualization scenarios, as we can use this technique in all layers of our system, starting from the underlying microkernel, plain microkernel programs, real-time applications, and our para-virtualized Linux kernel, as well as Linux user-space applications. It is extremely helpful for continuous monitoring to use only one mechanism in all places and to be able to collect all monitoring data with one framework.

Runtime monitoring of real-time systems is possible and useful. However, we find that one size does not fit all. The problem range is just too large for a generic solution with one sensor type. Even if one could obtain monitoring data at zero cost (CPU cycles) the sheer amount of data would have to be handled. Therefore, we applied early aggregation in the forms of histogram and scalar sensors to reduce the amount of data to be stored, transported, and evaluated later on. At first glance this might sound counterintuitive as aggregation itself does cost some CPU time in the monitored process, but the whole system will be relieved of a lot more load that way, resulting in lower overall intrusiveness. Currently, the user must decide whether and how to aggregate.

6 Outlook

In microbenchmarks, the event list as the most complex sensor can create events with an approximate overhead (depending on the sensor configuration and the payload) of about 200–300 cycles on an 1.2 GHz AMD Duron. The biggest costs are the copying of the payload, taking timestamps (with `rdtsc`), and synchronization against concurrent writers with the help of `cmpxchg8b` instructions.



(a) Ping start *(b)* Ping incoming *(c)* Ping done

Figure 6: Visualization of the system interaction in response to the ping request by the application. Depicted is only the send part wrapped in the two user-land events `l4lxu/arping`. You see the interaction from the user-space program (pid 92) with the L^{4}Linux server (task 12), where 12.4 is the main server and 12.15 being the driver stub communicating with the ORe network server. The network server is task 8.

Specifying the intrusiveness in a larger scale makes only sense in the context of an observer, either a human judging the system's behavior or machine parts evaluating metrics. For a server environment, the most important metric might be throughput, so intrusiveness could be defined over the change in throughput. For a real-time capable video player, the number of deadline misses is important, so the change in this fraction is the intrusiveness. In a hard real-time system, there should be no new deadline misses — here the intrusiveness is binary. For a desktop system where humans might not notice small changes but penalize larger ones, responsiveness is important. So, intrusiveness might not be linear with additional CPU time required. We will next work on defining metrics for certain application classes and verify our framework within these metrics.

Currently, events can be totally ordered per processor, as we use processor timestamp counter, which are strictly monotonic. Totally ordering event across several processors would require perfectly synchronized clocks, which probably cannot be achieved for such high-resolution time sources as timestamp counters. Instead, we envision to a use partial ordering relation by taking two local timestamps for cross-processor communication. We also plan to support different timestamp source — for example, logical clocks which might be cheaper (just a counter in memory) — for situations where not the accurate timing but only the ordering is important.

References

[1] Ronald Aigner. DICE Documentation. http://os.inf.tu-dresden.de/dice/.

[2] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, 2004.

[3] DROPS Team. DROPS - The Dresden Real-Time Operating System Project. http://os.inf.tu-dresden.de/drops/.

[4] Björn Döbel. Request tracking in DROPS. Master's thesis, TU Dresden, June 2006.

[5] F. Ch. Eigler, Vara Prasad, William Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of systemtap: a Linux trace/probe tool. http://sourceware.org/systemtap/archpaper.pdf, 2005.

[6] Norman Feske and Hermann Härtig. Demonstration of DOpE — a Window Server for Real-Time and Embedded Systems. In *24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December 2003.

[7] Steffen Göbel, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. The COMQUAD component container architecture. In *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Oslo, Norway, June 2004.

[8] Hermann Härtig, Steffen Zschaler, Martin Pohlack, Ronald Aigner, Steffen Göbel, Christoph Pohl, and Simone Röttger. Enforceable component-based realtime contracts — Supporting realtime properties from software development to execution. *Springer Real-Time Systems Journal*, 2006.

[9] IEEE. *IEEE Std 1003.1-2004 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. IEEE, New York, NY, USA, 2004.

[10] R. Krishnakumar. Kernel Korner: Kprobes—a Kernel Debugger. *Linux J.*, 2005.

[11] Adam Lackorzynski. L^4Linux Porting Optimizations. Master's thesis, TU Dresden, March 2004.

[12] Richard J. Moore. A Universal Dynamic Trace for Linux and Other Operating Systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2001.

[13] Ryan Myers. Funny, It Worked Last Time : Event Tracing for Windows (ETW). http://blogs.msdn.com/ryanmy/archive/2005/05/27/422772.aspx, May 2005.

[14] Dushyanth Narayanan. End-to-end tracing considered essential. In *Proceedings of High Performance Transaction Systems – Eleventh Biennial Workshop (HPTS '05)*, Asilomar Conference Center, Pacific Grove, CA, September 2005.

[15] Martin Pohlack, Ronald Aigner, and Hermann Härtig. Connecting Real-Time and Non-Real-Time Components. Technical Report TUD-FI04-01-Februar-2004, TU Dresden, 2004. http://os.inf.tu-dresden.de/papers_ps/tr-rtnonrtcomp.pdf.

[16] Torvald Riegel. A generalized approach to runtime monitoring for real-time systems. Master's thesis, TU Dresden, 2005.

[17] Carsten Rietzschel. VERNER – ein Video EnkodeR uNd playER für DROPS. Master's thesis, TU Dresden, 2003.

[18] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology, Stockholm, May 2000.

[19] Karim Yaghmour and Michel Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX Annual Technical Conference, General Track*, 2000.

RTOS Core Concepts and RT-safe Synchronization Mechanisms

K.P.Shiva Kumar and A.Satya Dev
shiva_kumar539@yahoo.co.in

Abstract

The quest of providing real time services to the end users are answered only by the use of Real Time Operating Systems(RTOS). The general purpose operating systems were not able to provide these services to the users as a result the online usage was priorly limited. But it is only after the advent of RTOS that the machine-user interation has become more friendly. The general operating systems were not able to provide the services for two reasons. Firstly,they disable the interrupts to protect internal data. Secondly they disable the scheduler during execution of long code. Since scheduler and interrupts are necessary tools for providing real time services it is not possible for them to provide them. The basic architecture of RTOS can be classified under three categories as Real time executive,Monolithic kernel,Microkernel. Microkernel proves to be the better architecture amonng the three.

No paper available at day of print.

Linux Lineament for Real Time Systems

B. Thangaraju
balat.raju@wipro.com

Abstract

Traditionally Linux is used for Server and Desktop systems. The open source model helps Linux to gain momentum consistently in Embedded and Real Time arena. Even though the vanilla kernel is not appropriate for real time systems unless it is modified according to real time system requirements. Many real time products are becoming an integral part of our life today. The developer needs to identify the features of a real time system because a dedicated real time system has minimum number of tasks and services unlike a general purpose system. Once the features are identified, and then need to apply the necessary available open source patches and rebuild the kernel. A real time Linux should have the real time features, which are recommended by the POSIX standard like real time signals, POSIX message queues, shared memory, high resolution timers, robust mutex, fixed priority pre-emptive scheduling, system latencies (preemption/scheduler/interrupt) should be deterministic and independent of system loads. Since many real time systems are powered by battery so need to implement an efficient power management policy. A real time device is expected to be up very soon after the power is on, so the various ways to reduce the boot up and shutdown time are to be identified. The size of the Linux kernel and the root file system are the major drawback to use directly in an embedded system and many real time product companies hesitate to use Linux because of the memory constrains. The methodologies to reduce the size, fit the Linux into the real time systems and security to protect an important data into a device will be analyzed thoroughly. This presentation starts with issues to convert general Linux into a real time system and intricacies of real time features will be discussed in detail.

No paper available at day of print.

General Performance Assesement of the L4/Fiasco Micro-kernel *

Cheng Guanghui, Zhou Qingguo*, Nicholas Mc Guire, Wu Wenzhong
SISE, Lanzhou University
Tianshui South Road 222, Lanzhou, Gansu, 7300000 P.R.China
chenggh04@st.lzu.edu.cn, zhouqg@lzu.edu.cn, mcguire@lzu.edu.cn

Abstract

This paper presents the experimental study of the performance of L4Linux on top of L4/Fiasco and Linux. Holding the hardware constant and using the different benchmark tools to measure a list of performances in order to give us a global view for performance comparison of L4Linux/Fiasco and Linux which could separately stand for the micro-kernel and monolithic kernel. Our experiments and analysis show the concrete measurement about their performances and in the end we present our conclusions. In Addition through these benchmarks we find some problems and maybe they are new task for the L4 community.

1 Introduction

The debate with micro-kernel and monolithic kernel has been very hot all the time since the concept of micro-kernel appeared and maybe everyone could remember that famous one named "Linus vs Tanenbaum". About one decade years ago the Herman Hartig, Jochen Liedtke and others published a paper "The performance of of u-kernel-based systems" in which the authors gave us a conclusion about the performance loss when comparing with monolithic kernel and micro-kernel. As a result the 5%-10% is as a conclusion when talking about the performance loss of micro-kernel vs monolithic kernel. In recent 10 years the Fiasco, L4Linux and Linux have many new versions. Especially the Linux kernel updates very very fast and from 2.0 to 2.4 and from 2.4 to 2.6 it varied very much. We repeat the similar experiments they did 10 years ago but with the tools and system of newest version.

L4 is a second-generation micro-kernel specification first designed and implemented by Jochen Liedtke. Now this specification of the L4 API has implementation in many different flavours, one of which explicitly targets hard real-time systems - L4/Fiasco. L4/Fiasco, which is being developed at TU Dresden since 1996 is currently an implementation of the L4 V2 API specificatoin. The L4/Fiasco is the core of DROPS (The Dresden Real-Time Operating System Project) system which supports running real-time and time-sharing applications concurrently on one computer. L4Linux is a port of the Linux kernel to the L4 u-kernel API. Because of existence of L4Linux we could use the same tool to test two systems with different architecture.

In order to get a global and fair overview for performance comparison of L4Linux and Linux we have four different benchmarks such as AIM9 , Lmbench, bonnie++ and NetPIPE. In other words it is also as the view of comparison of micro-kernel and monolithic kernel.

The rest of this paper is organized as following. In section 2: first we describe the test environment for evaluating them. Then we separate these results and reorganize them into 7 parts such as Numerical Operations, Process Operations, File Operations, Memory Operations, Process Communications, Disk I/O Performance, Network Performance. This classification mostly origins from the report of Lmbench benchmark. In section 3 we could state our conclusions about this test and give our opinions about the L4/Fiasco even the micro-kernel.

*Contact us: Qingguo Zhou; Phone: 086-0931-8912025; Email: zhouqg@lzu.edu.cn

2 Performance Benchmark

2.1 Experimental Setup

The goal of this paper is to evaluate the L4Linux/Fiasco and Linux as the example of micro-kernel and monolithic kernel. In micro-kernel side we choose the newest version of Fiasco and L4Linux : Fiasco 1.2 and L4Linux-2.6.17 (All the source code is downloaded from the cvs). In monolithic kernel side we choose the Linux-2.6.17 (The kernel is downloaded from the offical kernel ftp site). we use the same file system from Ubuntu-6.06 and the hardware with AMD Duron 1.6G and 512M RAM. But in order for accuracy of comparison we limit the Linux and L4Linux could only use 256M memory. This is the basic test envirionment and if the benchmark need special hardware we could describle in the following section. The version information about benchmark tools is as following: Lmbench-3.0-a3, AIM Bechmark Suite IX(AIM9), NetPIPE_3.6.1, bonnie++-1.03a.

2.2 Experimental Method

The benchmark once time is not reliable very much becuase of occasionality so every group of experiment we must have enough samples. So we have Lmbench benchmark 30 times and finally we have the average value. Because difference of AIM benchmark results with both L4Linux/Fiasco and Linux is trivial according to the results themselves. So we have the AIM benchmark 10 times. The situation of bonnie++ is like the Lmbench we do 30 times. The NetPIPE benchmark is very special and this test in Linux is very stable but L4Linux/Fiasco is very unstable. As a result the NetPIPE test in Linux is 10 times but the NetPIPE test in L4Linux/Fiasco needs to be 30 times. We will describle the NetPIPE benchmark in the following in detail.

2.3 Performance Metrics

2.3.1 Numerical Operations

Lmbench	integer	float	double
L4Linux	1	1	1
Linux	1	1	1
AIM9	**add**	**mul**	**div**
L4Linux	1	1	1
Linux	1	1	1

TABLE 1: *Basic Numerical Operations*

In the Lmbench benchmark about numberic operations in fact we test three types of number such as integer, float and double. In every type they have many kinds of operations such as add, mul, div and others. As different type we also give some particular operations For example mod of integer number.

In the AIM9 the types of benchmark are more including double,float,long,int,short and they only give 3 operations with add, mul and div.

Becuase they have too many kinds of numeric operations and all the results between these two kernels are almost the same. Therefore, in the TABLE 1 we use "1" to replace the real value because the raw data of L4Linux and Linux is the same. From the TABLE 1 we could find the performance of basic numberic operations with L4Linux/Fiasco and Linux almost has no difference and the micro-kernel doesn't have any visible performance loss.

2.3.2 Process Operations

Lmbench	null-call	null-I/O	stat	open/close	Slct-TCP
Linux	0.10	0.32	1.36	2.59	17.41
L4Linux	2.04	2.49	4.39	8.60	16.33

Lmbench	sig-inst	sig-hdnl	fork	exec	sh
Linux	0.42	1.70	185.83	1089.87	3960.53
L4Linux	3.10	6.06	1382.4	3395.5	9132.13

TABLE 2: *Process Operations-lmbench*

From the TABLE 2 we could find the system call of micro-kernel is usually about 3 times than the monolithic kernel at least except select-TCP but the performance of select-TCP has different actions as for others. The L4Linux is equivalent with Linux in select-TCP. In this table the smaller is better.

AIM9	exec	fork	signal -trap	function -call1	function -call15
L4Linux	331	3925	427938	114480668	66408559
Linux	331	3925	427938	114453504	68317184

TABLE 3: *Process Operations-AIM9*

From the TABLE 3 we could find in a solid period that creating or execting a process in L4Linux is much slower than in Linux. But when having function call they have the same performance measurement. In this TABLE the bigger is better because it could deal with tasks faster if the number is more.

In most situations the system call of micro-kernel is much slower than monolithic kernel from the Lmbench and AIM9. But this conlusion is not constant and it is only something experimental.

2.3.3 File Operations

Lmbench	0kFile Create	0KFile Delete	10kFile Create	10KFile Delete
Linux	45.43	15.16	122.73	29.49
L4Linux	63.14	15.91	143.37	36.93

TABLE 4: *File Operations*

In the TABLE 4 the smaller is better. From this table we could find that performance of Linux is also higher than L4Linux when they have some file operations.

2.3.4 Process Communication

Lmbench	pipe	af_unix	tcp	tcp-conn
Linux	5.41	7.03	13.56	55.73
L4Linux	18.32	19.71	29.34	109.07

TABLE 5: *Proc_Communication Latencies*

In the TABLE 5 we compare the communication latencies and we could find that the latency of L4Linux is double of Linux at least. And in this table the smaller is better.

Lmbench	pipe	af_unix	tcp	file-reread
Linux	135.27	137.37	118.2	358.46
L4Linux	144.87	146.77	122.53	376.83
Lmbench	bcopy (libc)	bcopy (hand)	mem-read	mem-write
Linux	534.30	251.95	251.97	534.67
L4Linux	501.78	251.01	250.87	501.1

TABLE 6: *Proc_Communication Bandwidth*

In the TABLE 6 the bigger is better and we compare the communication bandwidth of L4Linux and Linux. According to the result they have the similar performance in the communication bandwidth.

From the TABLE 5 and TABLE 6 we could know the communication latencies of L4Linux is much slower than Linux but the communication bandwidth of L4Linux and Linux are the similar.

2.3.5 Disk I/O operations

AIM9	disk_rr	disk_rw	disk_rd	disk_wrt
L4Linux	47306	38899	195565	76312
Linux	67936	55275	358607	99330
AIM9	disk_cp	sync_rw	sync_wrt	sync_cp
L4Linux	53142	6055	1958	1881
Linux	73723	631	168	168

TABLE 7: *Disk Operations(1)*

In the TABLE 7 the bigger is better. "disk_rr" means "Disk Reads (K) Per Second" and others are similar. According to this table the disk_read and disk_write in L4Linux is also slower than in the Linux. but the actions relative with sync in L4Linux is much more than ones in Linux. According to the data in the TABLE 7 the sync speed in L4Linux is 10 times than in Linux at least.

bonnie++	seq_char_ read(k/sec)	cpu%	seq_blk_ read(k/sec)	cpu%
L4Linux	29160.7	95.4	63063	43.4
Linux	3063.1	15.5	3840.5	12.4
bonnie++	seq_rewr- ite(k/sec)	cpu%	seq_char_ write(k/sec)	cpu%
L4Linux	25973.8	16.6	25073.1	84.0
Linux	1444.2	26.1	2871.2	88.6
bonnie++	seq_blk_ write(k/sec)	cpu%	random _seek(/sec)	cpu%
L4Linux	61586.3	18	282.6	1
Linux	2917.9	7.5	199.6	3.1

TABLE 8: *Disk Operations(2)*

bonnie++	real(m)	user(m)	sys(m)
L4Linux	78	15	29
Linux	547	80	96

TABLE 9: *Time Expense*

From the TABLE 8 we could find that the L4Linux reads or writes the disk much faster than Linux although they take up more CPU resources. But even if they use the same CPU capacity the L4Linux does much faster than Linux, too. We have the bonnie++ 30 times in total. From the TABLE 9 it only needs 78 minutes to finish all 30 tests in L4Linux but in Linux it needs 547 minutes to accomplish it.

2.3.6 Network Performance

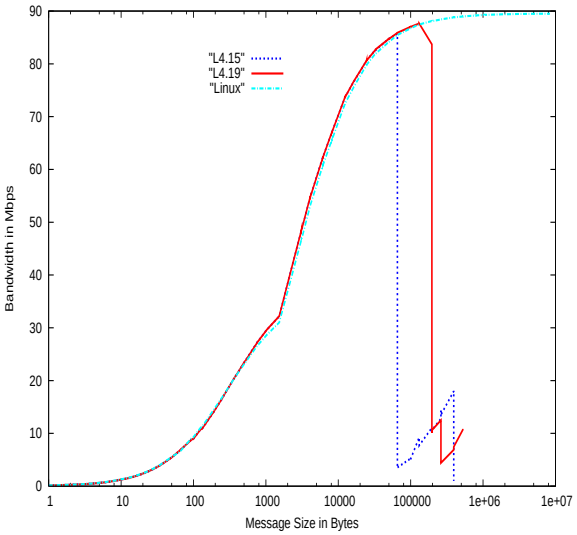


Figure 1: NetPIPE Figure(1)

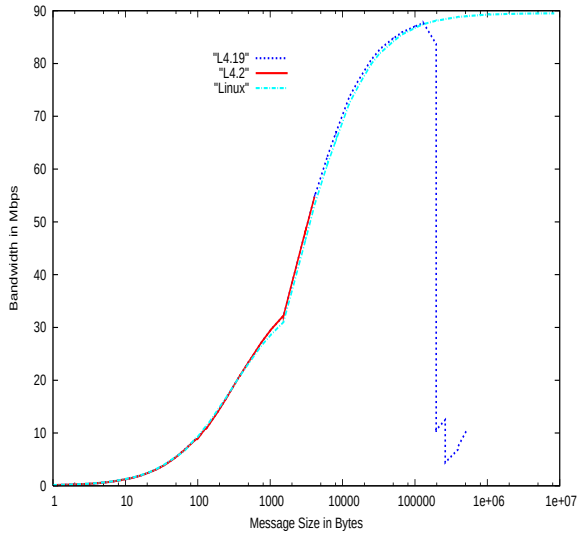


Figure 2: NetPIPE Figure(2)

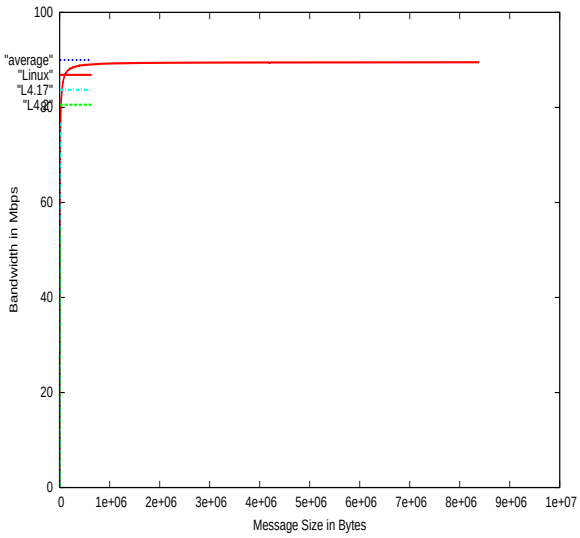


Figure 3: NetPIPE Figure(3)

NetPIPE	average	maximum	minimum
Linux	89.4997	89.5038	89.492934
L4Linux	68.4186	87.6859	54.7619

TABLE 10: *Network Performance*

Testing of network performance showed some serious problems. We are not yet able to clarify fully if these problems are related only to the newest 2.6.17 release of L4Linux or more fundamental. Nevertheless we present the results here although we don't overinterprete due to the problems.

The basic setup is simple, we run NetPIPE on the native Linux installation and then rerun it under the L4Linux paravirtualization of the version-same kernel(2.6.17). Some problems seem to have been related to the PCI bus, the first tests were made with mainstream PCI card (RTL8139 and EEPRO100) - these test runs did not complete but simply hung for ever all the time. And the L4Linux output the "kernel panic" words. We tried different types of PCI cards but the same problems still appeared.

Later we modify the relative file in L4Linux the "kernel panic" problem was resolved but the test didn't complete in most of time yet (2 times in 30 times are completed normally). Now the problems seem to be related to fairly large packet sizes and not directly related to the PCI bus. We think maybe it is a more fundamental problem in the current L4Linux version (CVS version 2.6.17).

In the Figure 1 you could see the 2 samples in which the netpipe could exit normally named L4.19 and L4.15 in comparison with the Linux sample. In the Figure 2 you could see the maximum sample of L4Linux named L4.19 , the minimum sample of L4Linux named L4.19 and the Linux sample. In the Figure 3 you could see the average L4Linux sample named average, the maximum sample named L4.19, the minimum named L4.2 and the standard Linux sample.

From these tests we think the L4/Fiasco or L4Linux is not stable in some situation. But L4Linux has a strong potential in the improvement of network performance because sometimes the maximum bandwidth could be up to 87M but the Linux is only 89M. And in 30 times experiment they have 5 times that the bandwidth is more than 80M. So we think it is very urgent and important to improve the stability of L4/Fiasco or L4Linux. If somebody could fix it it is possible that the average maximum bandwidth is up to 80M. Maybe how to resolve it is our nextstep of studying L4/Fiasco.

3 Conclusion

In this paper we have a series of benchmarks about performance comparison with the L4Linux on top of Fiasco and Linux. First ,we present the first-hand performance benchamrk data about the L4Linux-2.6.17 based on fiasco-1.2 and the Linux-2.6.17. In the end part of section 2.3.4 - Process Communications we could see a strange phenomenon. The communication latency of L4Linux is much slower than Linux but the communication bandwidth of them are similar. In the TABLE 8 there is a test "seq_rewrite" different from others which make use of more CPU resources to have more throughout-

put. But the seq_rewrite makes use of less CPU resouces to up to more throughoutput. According to principle of bonnie++ we could think that the data transferring speed in L4Linux is much higher than Linux. Then, in total we still think the performance loss of microkernel exists but in the disk operations the performance of L4Linux is very surprising to us especially bonnie++ bechmark. Finally, in the netpipe bencharmk althgough we find some problems we think they have a strong potential. If we could fix that problem the maxium bandwidth is possible to up to the level in Linux.

4 Acknowledgements

We'd like to thank Adam Lackozynski of Dresden University of Technology very much. We asked him many problems when building the benchmark envionment and he was very patient and quick to give me the answer.

Thanks to the support of Cold and Arid Regions Environment and Engineering Research Institution, Chinese Academy of Sciences.

Thanks to Professor LiLian of SISE, Lanzhou University.

Thanks to all other students in DSLab of Lanzhou University. They help us to review this paper and make the latex format.

References

[1] Andrew S. Tanenbaum, Jorrit N. Herder, Herbert Bos, 2006, *Can we Make Operating Systems Reliable and Secure?*, Computer, vol 39, no.5 pp.44-51

[2] Herman Hartig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, 1997 , *The Performance of u-Kernel-Based Systems*, 16TH ACM SYMPOSIUMON OPERATING SYSTEMS PRINCIPLES

[3] Larry McVoy, Carl Staelin, *lmbench: Portable tools for Performance analysis*

[4] Qingguo Zhou, Wang Baojun, Nicholas McGuire, 2004, *Case Study of Performance of Real-Time Linux On the X86 Architecture*, THE SIXTH REAL-TIME LINUX WORKSHOP

[5] Herman Hartig, Michael Hohmuth and Jean Wolter, 1998, *Taming Linux*, THE PROCEEDINGS OF PART '98

[6] Michel R. Dagenais, *Disks, Partitions, Volumes and RAID Performance with the Linux Operating System*

[7] Nicholas Mc Guire and Qingguo Zhou ,*Benchmarking - Cache issues*, THE REAL-TIME LINUX WORKSHOP 2005

[8] Hermann Hartig, Michael Hohmuth, Noman Feske, Christian Helmuth, Adam Lackozynski, Frank Mehnert and Michael Peter, 2005, *The Nizza Secure-System Architecture*, THE PROCEEDINGS OF COLLABORATIONCOM 2005, SANJOSE,CA,USA

Introducing the STRTL Live CD

Arthur Siro
TECNUN, Escuela Superior de Ingenieros de la Universidad de Navarra
Paseo de Manuel Lardizabal 13, 20018, Donostia-San Sebastian, Spain
asiro@ceit.es

Emilio Sanchez
CEIT, Centro de Estudios e Investigaciones Tecnicas de Guipuzkoa
Paseo de Manuel Lardizabal 15, 20018, Donostia-San Sebstian, Spain
esanchez@ceit.es

Abstract

First time installation of a real-time Linux flavour can be quite a challenge epecially for those unfamiliar with Linux. Partitioning a hard with a pre-installed OS and the possible risk of data loss will even scare some away in the firstplace! Furthermore, the cryptic looking commands, instructions & technical language required for the usage of a real-time Linux variant and its corresponding prototyping target can put off staff and students of the control lab. This first version of the 'STRTL Live CD offers a quick-and-dirty, plug-and-play type solution to this problem.

Keywords: Real-Time Operating System (RTOS), Real-Time Workshop (RTW), Simulink Target for RTLinux (STRTL), Real-Time Application Interface (RTAI)

1 Introduction

Rapid prototyping applications are now standard tools in control practice. Basically, these tools integrate a computer aided control system design tool CASCD like the propriety MATLAB/Simulink/RTW[1] package or freeware SCILAB/SCICOS[2] suite and a (mainly) PC based real-time development target. These real-time environments come in both commercial and freeware flavors. RTLinux/GPL[3] and RTAI[4] are examples of freeware targets while the real-time kernel-mode driver for Microsoft Windows 2000/XP, and the VxWorks RTOS by Wind River Systems, Inc are some of the commercial or propriety options available in the market. RPTs basically facilitate the design and testing of control systems before deployment on a final product. The STRTL[4] and RTLab[6] are open source implementations of RPTs. These target RTLinux and RTAI, respectively. Several commercial products are also available. The Real-Time Windows Target (RTWT)[1], xPC target[1], the dSpace Real-Time Interface (RTI), Tornado target[1] (which targets VxWorks) are but a few examples.

Now, commercial software is always a more polished product. Commercial RPTs come with several blows and whistles, like elaborate GUIs and documentation that are just not found with their freeware counterparts. But their cost/price is simply not worth it for many practical control applications, especially in education. This is when a freeware RPT can be the best option.

RTLab supports both the Matlab/Simulink/RTW as well as the SCILABS/SCICOS suite whereas STRTL -at the moment- only supports the Matlab/Simulink/RTW package. However, to get these applications to work, one has to have RTLinux/RTAI installed first. This might sound trivial for many 'linuxers familiar with kernel configuration and builds but it could be forbidding or daunting for many high level computer users of other popular OS's. First time installation of a fresh real-time Linux system does require some planning -and effort!; partitioning hard disks with pre-installed OS, making backups the data of these systems due to the possible risk of data loss etc. A quick solution would be to purchase extra hard disks. But then before the newbie can begin the trial and error process of a fresh kernel build, some little research is necessary; literature on

tools and utilities needed (e.g. binutils, modutils, gcc, make etc), quick web browsing for answers to FAQ's etc. Then comes the agony of defeat (as almost nothing works the first time) before one can get a clean working system. Now most control engineers/ students are used to propriety software with 'nice' GUI; Control is not a trivial science in itself and many may neither have the time nor the energy to spend on other (cool) things. Moreover, institutions may also not be too exited with the idea of incurring extra costs and time for the sake of experimenting with non-commercial 'geek' software -whose technical support is mainly via mailing lists. Many will just opt for packaged commercial software with cute GUI and documentation and direct technical support to save them from the hustle and bustle.

It is with this background that the STRTL live CD AKA STRTL-LCD was developed. Like any live Linux CD, STRTL-LCD objective is to give the user a chance to witness Linux (RTLinux in this case) in action on almost any RTLinux supported desktop computer without having to install anything. Its plug and play approach, zero risk of pre-installed system data loss, and very little extra cost, if any, is hoped to make things a lot easier for anyone open to giving rapid prototyping on a real-time Linux target a try.

2 STRTL-LCD Description

STRTL-LCD is a live, real-time oriented, mini-Linux distribution meant to run off a PC rather than an embedded device. In this respect, STRTL-LCDs design requirements are based more on functionality rather than the restrictions of system size/resources. Its main job is to compile, build, load and execute a RTW C-code version of a Simulink model in real-time. However, though STRTL-LCD's functionality and target hardware may not exactly qualify as an embedded Linux system, it has been -to a certain extent -implemented with an embedded Linux oriented approach. For example, RAMDISK usage is modest; just enough to hold a decent BusyBox[7] based root file system, the RTLinux modules and an executing instance of your (once) Simulink model while the rest of the System data, files and miscellaneous applications are stored on CDROM. The idea here is to help users gauge the hardware/software requirements of their final embedded Linux system. In fact, the main difference between implementing a controller on a PC running STRTL-LCD and implementing it on an PC compatible, RTLinux complaint, embedded device e.g. PC-104(/PCI) is that STRTL-LCD supports the compilation and build of the RTW generated code apart from executing it in real-time. As mentioned above, all system data, files, tools including gcc and friends, the STRTL code base and other system includes required for the compilation and build of your control model are stored on CDROM. These get mounted on boot and are pointed to by symbolic links but do not get loaded into memory along with the compressed root file system. Another peculiar feature of RTecNUx with respect to an embedded Linux system is that since STRTL-LCD is expected to work on a variety of computers rather than a known hardware setup, it has to include as much support as possible for the various Linux compatible network interface cards, NIC. These are configured and compiled as loadable kernel modules but stored on the CDROM to avoid bloating up the RAMDISK based root file system. The /lib/modules folder on this root file system is actually a symbolic link to a real 'modules folder on CDROM. This 'modules folder naturally contains the other loadable kernel modules e.g. filesystem drivers.

Total RAMDISK size is about 15MB. On boot, the root file system takes about 7MB. The extra 50% of RAMDISK is generously reserved for your model code and binaries, and the library archive and object code generated from the RTW library source during the build process.

A three command sequence on the console (at root directory) reveals STRTL-LCDs general file system layout:

```
(none):/# ls -l; ls -l /lib/modules

drwxr-xr-x  2 root 0 1024 Sep 24 20:34 RTLinux
lrwxrwxrwx  1 root 0   20 Sep 24 21:51 STRTL_M7->
/UTILITY/STRTL_M7/
dr-xr-xr-x  8 root 0 2048 Sep 29 07:55 UTILITY
drwxr-xr-x  3 root 0 1024 Sep 23 00:26 bin
drwxr-xr-x  3 root 0 2048 Sep 29 12:18 dev
drwxr-xr-x  5 root 0 1024 Sep 29 12:18 etc
drwxr-xr-x  2 root 0 1024 Sep 20 23:06 home
drwxr-xr-x  2 root 0 1024 Sep 20 23:30 lib
drwxr-xr-x  2 root 0 1024 Apr 20  2001 mnt
dr-xr-xr-x  22 roo 0    0 Sep 29 12:18 proc
drwxr-xr-x  2 root 0 1024 Sep 24 21:01 sbin
drwxrwxrwx  2 root 0 1024 Apr 19  2001 tmp
lrwxrwxrwx  1 root 0   13 Sep 24 21:51 usr->
 /UTILITY/usr/
drwxr-xr-x  5 root 0 1024 May 11  2001 var
drwxr-xr-x  3 root 0 1024 Sep 20 23:03 work
lrwxrwxrwx  1 root 0   16 Sep 24 21:51 /lib/
modules -> /UTILITY/modules
```

and,

```
(none):/# mount

/dev/root on / type ext2 (rw)
none on /proc type proc (rw)
/dev/cdrom on /UTILITY type iso9660 (ro)
```

As it can be seen from the first listing, the /STRTL_M7, /usr and /lib/modules directories are actually symbolic links to corresponding real directories within the /UTILITY directory. The 'mount command then shows that /dev/cdrom is mounted on /UTILITY.

3 STRTL Description

Like most real-time application frameworks, the control algorithm within STRTL is executed with the highest priority and in real-time. Data gathered from the DAQs board is buffered within shared memory for a user mode server to upload to a Simulink process in a client machine. Simulink then displays the data via Scope blocks or other graphic displays. Parameters from the Simulink process can be passed to the real-time model via the user mode server of the target machine. Communication between the two machines/ processes is via the TCP/IP protocol. Below is a graphic illustration of the STRTL framework.

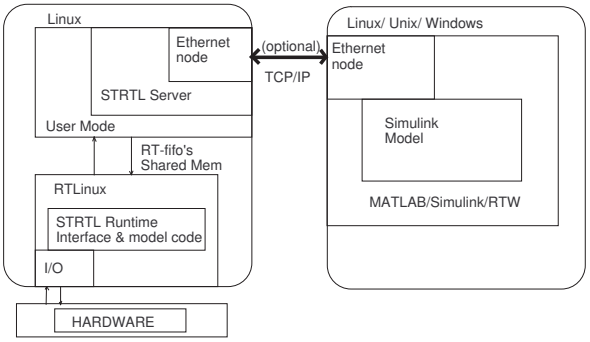


FIGURE 1: *Real-Time Model execution under STRTL*

4 Discussion

A RAMDISK based mini-distribution of a real-time Linux variant is nothing new. In fact, the inclusion of a freeware, rapid prototyping tool, RTLab in a live Linux distribution KNOPPIX has even been reported[8]. However, and as mentioned previously, what STRTL-LCD strives to do is to provide an embedded like environment/feel on the PC during the rapid prototyping process so that one can roughly gauge the hardware/software requirements of their final 32/64 bit PC compatible and RTLinux compliant embedded device e.g. the popular PC-104 form factor[9]. In this sense, STRTL-LCD attempts to provide a 'transparent, low-cost, open source, all round solution for control system development and implementation.

Nevertheless, there exist some issues as regards the code generated by RTW that you need to be aware of if you are planning to implement an STRTL-LCD application directly on an embedded RTLinux platform. Since this code was generated for rapid prototyping, it has not been optimized for speed, memory usage and simplicity. This does not necessarily mean that your application wont run; the tradeoff is that you should not expect production code. The reason for this is that the rapid prototyping framework provides generic, high-level API that does not change between model definitions - precisely to enable you to prototype quickly. This common API can be viewed as a layer of abstraction between your particular Simulink Model and the real-time target environment. Thus the overall code framework incurs some overhead and uses some not-so-compact data structures.

5 Lab Application: A 2 DOF Teleoperated robot

Laboratorio de Electrnica y Automtica of TECNUN Engineering Campus of the University of Navarra is a lab with about 10 desktop PCs. Until recently, most control practice or research was either carried out using commercial proprietary products such as the Real-Time Interface, RTI of dSPACE, Inc, the WinCon target of Quanser Consulting, Inc., the Realtime Windows Target (RTWT) by Mathworks, Inc. or was simply hand coded in C/C++ and assembly to run from microcontroller based circuits connected to the control system.

The following lab experiment was carried out on STRTL-LCD. A simplified control algorithm for a 2 degree of freedom, DOF, robot was tested. This was mainly due to the lack of time needed to model a more complex algorithm, rather than anything else. The robotic system is an electronically coupled master-slave parallel robot. A position-position control scheme was used by which the Master Robots position from the encoders is compared to the Slave Robots position and vice versa to maintain tracking and force feedback.

The control algorithm was executed at 1000 KHz on a 350MHz , Pentium II CPU, with 64MB of RAM. After the build/link phase and deletion of the RTW library archive and associated object files, RAMDISK usage was slightly under 10MB.

The overall STRTL-LCD system, tools and utilities including gcc and friends, make and binutils were based on Slackware 10.1 and Busybox v1.1.2. RTLinux-3.2-rc1 on a 2.4.29 kernel was used.

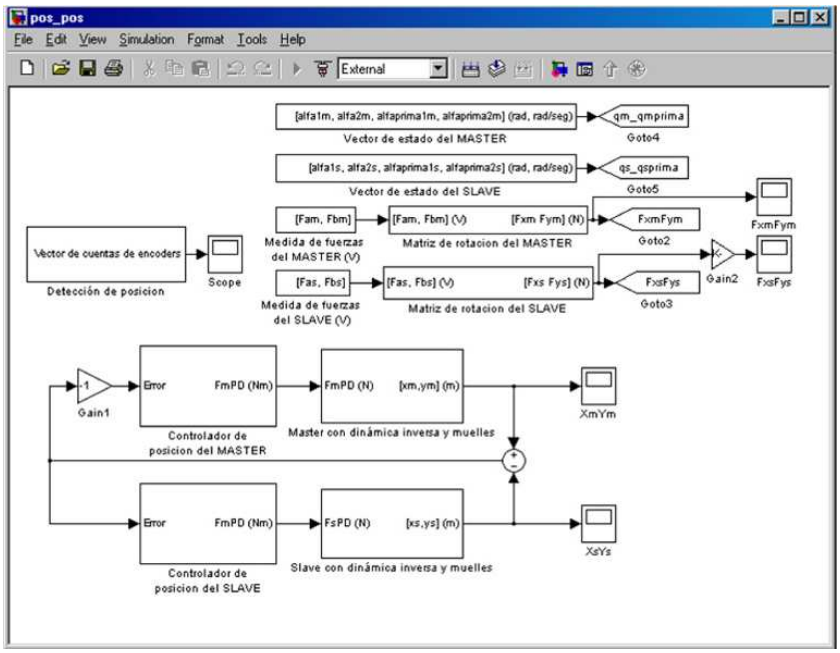


FIGURE 2: *The Simulink Diagram Model*

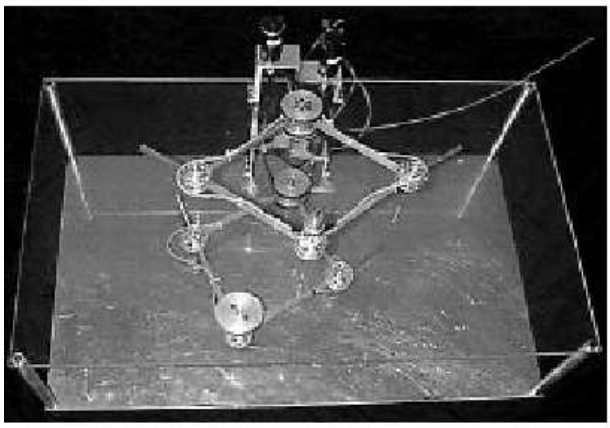


FIGURE 3: *The 2 DOF teleoperated robot setup*

6 Conclusion and Possible development paths

STRTL-LCD is just beginning to take its first few steps. The STRTL for Matlab 6 was recently upgraded Matlab 7 by the author(as the original developer no longer maintains it) and so feedback from the community at large is needed. Inclusion of RTAI, a tiny X server and Qt for RTLab -and more decent name for the CD!- are possible development pointers.

References

[1] www.mathworks.com

[2] www.scicos.org

[3] www.rtlinux-gpl.org

[4] www.rtai.org

[5] Raul Murillo Garcia, F. Womle, Brian G. Stewart, David K. Harrison, 2 June 2004, *Hard Real-Time Control Using the Simulink Target for RTLinux*, International Jornal of Computers and Their applications, Volume 11, Number 2.

[6] R. Bucher, L.Dozio, 2003, *CASCD under RTAI Linux with RTAI-Lab*,Real-Time Linux Workshop, Valencia

[7] www.busybox.org

[8] Roberto Bucher (SUPSI), Lorenzo Dozio, Paolo Mantegezza (DIAPM), 2004,*Rapid Control Prototyping with Scilab/Scicos and Linux RTAI*,1st Scilab International Conference, France

[9] Nurpinar Akdeniz and Tekin Aydin, 2002, *A Novel Approach To MiniRTL Implementation for Instrumentation And Control Applications*,4th Real-Time Linux Workshop, Boston, USA.

Analyzing RTLinux/GPL Source Code for Education

She Kairui, Bai Shuwei, Zhou Qingguo, Nicholas Mc Guire, and Li Lian
Distributed and Embedded Systems Lab
School of Information Science and Engineering
Lanzhou University
Tianshui South Road 222,Lanzhou,P.R.China
{shekr06,baishw06}@lzu.cn
{zhouqg,mcguire}@lzu.edu.cn

Abstract

With decades development, RTLinux/GPL was widespread applied to both scientific research realm and industry. RTLinux/GPL has made a grate success in these realm, But has little affection in education, especially the undergraduate education. Most of the RTLinux/GPL documentation are focus on the practicability, other than emphasizing the basic RealTime Operatiing System theory. So RTLinux/GPL is very hard to applied to education realm. This article aimed at RTOS education, containing real time principle of RTLinux/GPL and the RTLinux/GPL modules implementation in detail. It would make the realtime operating system learning process easier with both theory and real source code analyze, and would attract more people to interested in RTLinux/GPL. This article presented RTLinux/GPL principle and implementation details, containning realtime clock, interrupt, realtime schedule strategy. We hope this article can help people to apprehend RTOS, especially the RTLinux/GPL.

1 Introduction

Nowadays, application of RTOS(RealTime Operating System) has becoming more and more prevalent, resulting that RTOS takes more and more important role in industry. So the research and development of RTOS is very active. But RTOS has little affection in education, especially the undergraduate education, for lack of appropriate document and teaching material. The RTOS educational instruction only focus on the essential RealTime System principle, without abundant RealTime case study.

RTLinux/GPL is distributed under GPL, This feature is worthful, students can get RTLinux/GPL source code freely, even they can modify the source code. But most document of RTLinux/GPL are introductory or about Application Programming Interface, which with little corresponding principle, which makes newcomers shrink back at the sight.

In order to further spread the RTOS in education realm, we take the RTLinux/GPL as an example, we will explain the essential principle of RTOS,such as RealTime schedule algorithm, and the concrete implementation in RTLinux/GPL. This article covers RealTime Operating System Architecture, RTLinux/GPL implement principle, clock, timer, interrupt management, RealTime schedule algorithm(RM,EDF), and the implementation dissection of these mentioned concepts in RTLinux/GPL.

2 RealTime OS and RTLinux/GPL

Generally, RTOS refer to Operating System with certain real time resource schedule and communication capability[1]. According to the real time capability, RTOS can classified as Hard RTOS and Soft RTOS. In this article the referred RealTime always indicate Hard RealTime,except for particular declaration.

During the RTOS design phase, designer must consider one basic conflict requirement, on the one hand the custom anticipate the target system support hard real time capability, on the other hand they also want the the system provide abundant function and feature like desktop PC system. There exists two solution to dilemma: Expand the exists RTOS,

*This research was supported in part by Cold and Arid Regions Environment and Engineering Research Institution, Chinese Academy of Sciences

or make a general purpose operating system Real-Time capable via add certain software layer. this article take RTLinux/GPL as an example, explaining the second solution implementation in detail.[2]

As general purpose OS, Linux system is a time sharing OS based on time slide Round Robin algorithm, further more, Linux kernel is nonpreemptive. generally, Linux time resolution is 10ms, and not satisfy the hard real time requirement.

RTLinux/GPL is a typical dual-kernel, one is Linux kernel, which provide various features of general purpose OS, other one is RTLinux kernel, which support hard real time capability. Figure 1 illustrate the RTLinux/GPL architecture.

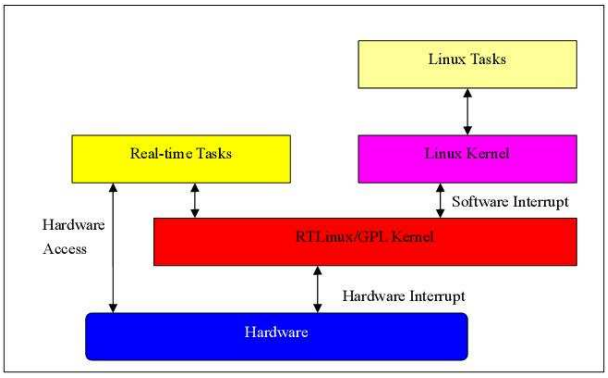


FIGURE 1: *RTLinux/GPL Runtime Model*

3 Clock and Timer

Clock is the hardware resource used for time management in computer. Timer is hardware or software facility that allows functions to be invoked at some future moment,after a given time interval has elapsed[3]. Generally speaking, hardware clock contain a hardware timer, but the only hardware timer is far from enough in multi-task system. So multi-task system need software timer to provide more timers.

This section will describe the RTLinux/GPL low level hardware clock management and soft timer management.

3.1 Clock

The current RTLinux/GPL version support two kinds of hardware timers, APIC and 8254, they are used for multi-processor system and uni-processor system respectively. RTLinux/GPL provide the identical clock control API to manage both hardware clock. We can use the API to achieve timer setting, read or write time,etc.

The RTLinux/GPL clock control APIs:

```
int init(struct rtl_clock *c);
void uninit(struct rtl_clock *c);
hrtime_t gethrtime(struct rtl_clock *c);
int sethrtime(struct rtl_clock *c, hrtime_t t);
int settimer(struct rtl_clock *c,
    hrtime_t interval);
int settimermode(struct rtl_clock *c, int mode);
void handler(struct pt_regs *r);
```

3.2 Timer

RTLinux/GPL can provide one or more soft timer for each real time task, and RTLinux/GPL use linked-list to management the soft timer[4]. Generally,all soft timer use the same low level hardware timer, operating system will select such soft timer in the linked-list based on certain timer schedule algorithm, then use the selected soft timer structure to set the hardware timer via timer control API. The Timer Interrupt will trigger task schedule at the special moment. How to operate the soft timer, such as create and destroy? RTLinux/GPL soft timer API implementation POSIX compliant.

Timer manage related API:

```
int timer_create(clockid_t clock_id,
    const struct sigevent *signal_specification,
    timer_t *timer_ptr);
int timer_gettime(timer_t timer_id,
    struct itimerspec *ts_set);
int timer_settime(timer_t timer_id, int flags,
    const struct itimerspec *new_setting,
    struct itimerspec *old\_setting);
int timer_delete(timer_t timer_id);
```

Figure 2 illustrate clock and timer hierarchy model.

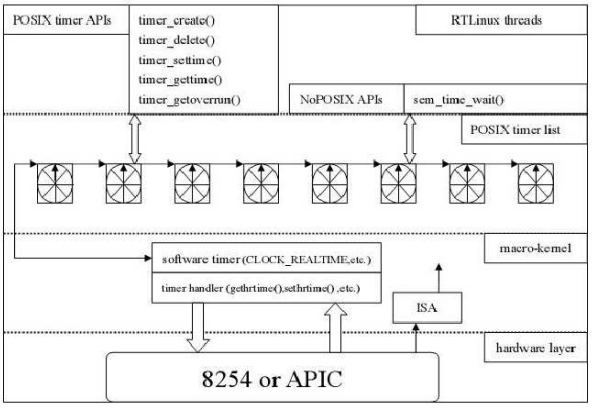


FIGURE 2: *Clock and Timer Hierarchy Model*

4 Interupt

In this section, we will start with Linux Interrupt, and then discuss RTLinux Interrupt implementation.

IBM compatible PC use two 8259A chips in cascade to make up of the PIC(Programmable Interrupt Controller), which can provide 15 IRQs. Each hardware device controller capable of issuing interrupt requests to the PIC, PIC can converts interrupt request into a corresponding interrupt vector and then store the vector. After PIC send a raised signal to the processor INTR pin, PIC will wait until CPU acknowledge. CPU would use this interrupt vector as index and get a corresponding interrupt service routine entry from IDT(Interrupt Description Table).

As for x86 architecture, CPU support 256 interrupt vector, IDT data structure is an array named idt_table, witch includes 256 entries.

interrupt initialization:

head.S is part of the uncompressed section of the vmlinuz image that is directly called by the bootloader - it is responsible for basic setup of the low level resources so that the hardware is actually accessable, it then calls the start_kernel symbol with is the entry point into the decompressed kerner prope.

call SYSTEM_NAME(start_kernel)

start_kernel is the compressed kernel entry, initialization of 8259A chips and interrupt gate is performed by the function call init_IRQ.

RTLinux/GPL running as modules after Linux kernel startup. RTLinux/GPL will takeover the Interrupt control when these modules was loded. RTLinux/GPL modules will patch the running linux kernel.

In linux each interrupt will come to the function common_interrupt, and then jump to do_IRQ(jmp do_IRQ). The primary role of patching is to change the jump destination to RTLinux interrupt entry(jmp rtl_intercept), so all interupt will be intercepted by RTLinux/GPL.

Figure 3[5] illustrate the RTLinux/GPL interrupt intercept

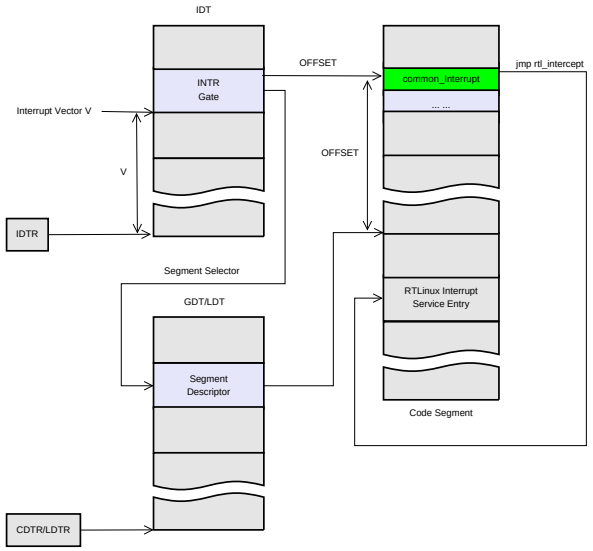


FIGURE 3: *RTLinux/GPL Interrupt Intercept*

RTLinux interrupt will be responsed immediately, as for Linux interrupt, if Linux interrupt is enable, Linux interrupt service routine will be called immediately, otherwise this Linux interrupt will be blocked until Linux interrupt is enabled. This process known as Soft Interrupt.

RTLinux interrupt API:

```
//add/remove real time interrupt handlers
int rtl_request_irq(unsigned int irq,
    unsigned int (*handler)(unsigned int irq,
        struct pt_regs *regs));
int rtl_free_irq(unsigned int irq);
//install/remove software interrupt handlers
int rtl_get_soft_irq(void (*handler)
    (int, void *, struct pt_regs *),
    const char * devname);
void rtl_free_soft_irq(unsigned int irq);
//schedule a Linux interrupt
void rtl_global_pend_irq(int irq);
```

The function schedule the interrupt specified by argument irq to be happen when the system enters the Linux mode.

5 Schedule Policies of Real-Time Operating System

Except for high resolution clock management and efficient interrupt process capability, task schedule policy is another critical factor directly affect the Real Time capability in the multi-task System. A multi-task System allows more than one task to be loaded into the executable memory at a time[6], the loaded

tasks share the CPU, so how to arrange the task execute sequence is important? We can choose appropriate schedule algorithm for corresponding requirement. The purpose of a real-time scheduling algorithm is to ensure critical timing constraints. RTLinux/GPL is a multi-task system with real time capability, and RTLinux/GPL implemented two priority based schedule algorithm,RM(Rate Monotonic) and EDF(Earliest Deadline First). RM schedule algorithm based on static priority, and EDF based on dynamic priority[7]. Static priority means each task was assigned a fixed priority. Generally, priority assigning was according to the task attribute, In RTLinux/GPL, realtime task always with higher priority than non-realtime task. Dynamic priority means the task priority is alterable in term of its resource demands, so dynamic priority schedule algorithm is more flexible for task schedule and resource assign.

5.1 RealTime Task, Priority, Preemptible

Before schedule policies presentation, I will explain several important concept firstly.

Real-Time task: The scheduler operational objects are real-time tasks, which require the tasks must be executed or finished at a given time(typically milli- or microseconds).

Priority: The priority is based on a predetermined assignment value, or importance to different types of tasks. In the RTLinux system, the Linux kernel has the lowest priority, and the real-time threads has the high priority(not less than 1000).

Preemptible: If the higher priority task could preempt CPU from the running task with lower priority by force, the system is preemptible, or non-preemptible. Such as, the old linux distributions are non-preemptible, but the RTLinux system is preemptible.

5.2 priority based Rate Monotonic algorithm

5.2.1 Rate Monotonic algorithm

Rate Monotonic algorithm based on fixed priority, system assign a priority for each task according to the task expected execution time, task with less execution time will assigned higher priority.

RM algorithm principle:

- schedule independent periodic task with fixed priority.
- priority assign policy: less cycle task will assign higher priority, assume cycle of non-periodic task is infinite.
- higher priority preemption.

following example would illustrate the the RM algorithm, considering three tasks: T1, T2, T3, T1 and T2 are periodic task, cycle of T1 is P1=5s, cycle of T2 is P2=10s, the task execution time is C1=C2=2s, T3 is non-periodic task, execution time is C3=5s, assume three tasks are ready at the same time, Figure 4 is the RM task schedule diagram.

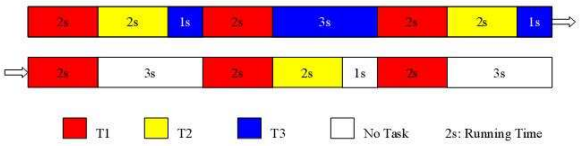


FIGURE 4: *RM Task Schedule Diagram*

The diagram reveal that the task with least cycle is T1, which was assigned highest priority. Task T1 was first scheduled before task T2 and T3, and T1 can preempt T3. T2 was scheduled after T1 and before T3, T3 was last scheduled.[8]

In order to work correctly, certain preemption must be satisfied :

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_n}{T_n} \leq n(2^{1/2} - 1)$$

n is the total task number in the SystemCi($i \leq n$) stands for the longest execution time of task i, Ti($i \leq n$)stands for the cycle of task i.

5.3 RM algorithm implementation in RTLinux/GPL

considering following code fragment

```
257 if ((t->pending & ~t->blocked) &&
258  (!new_task || (cmp_prio(t, new_task)>0) ) ) {
259   #ifdef CONFIG_RTL_SRP
260    if (!current_sysceil ||
261      (cmp_preempt_level(&t->sched_param,
            current_sysceil) > 0))
262   #endif // CONFIG_RTL_SRP
263    new_task = t;
264  }
```

line 257 decide whether the process is blocked, line 258 select the task with highest priority. we don't care about the rest lines, which is resource storage style related.
the cmp_prio function:

```
51 static inline int cmp_prio(pthread_t A,
                             pthread_t B)
52 {
53 #ifdef CONFIG_RTL_SCHED_EDF
54 register int tmp;
55 if ( (tmp= (A->sched_param).sched_priority -
              B->sched_param).sched_priority) )
56 return tmp;
57 else
58 return ( (B->current_deadline) >
            (A->current_deadline) );
59 #else
60 return (A->sched_param).sched_priority >
         (B->sched_param).sched_priority;
61 \#endif
62 \}
```

line 53 to 59 EDF related, we will discuss it next section. considering line 60, if the return value is 0, then the task with highest priority will be selected , but context switch would not perform immediately. Scheduler will check the clock mode first, if clock mode is ONESHORT, scheduler will reset the hardware timer to issue interrupt at a specified moment, and then switch the context.

5.4 Earliest Deadline First algorithm

5.4.1 EDF algorithm

EDF schedule algorithm is based on dynamic priority. System assign a priority for each task according to the Deadline of the task dynamically, Task with earliest deadline will assigned highest priority[9]. In order to schedule correctly, certain preemption must be satisfied:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_n}{T_n} \leq 1$$

n is the total task number in the SystemCi($i \leq n$) stands for the longest execution time of task i, Ti($i \leq n$) stands for the cycle of task i.

Take an example, task T1, T2, T3, cycle: P1= 11s, P2=10s, P3=12s, execution time C1=4s, C2=5s, C3=5s. At moment t=0, each Deadline is: D1=P1-C1=7s, D2=P2-C2=5s, D3=P3-C3=10s, D2 < D1 < D3, so priority of task T2 is higher than T1 and T3, T2 will execute first. After task T2 finished, namely the moment t=4s, Deadline of each task is: D1=D1-t1=3s, D2=D2+ P2-C2=14s, D3=D3-t1=6s, now D1 < D3 < D2, so task T1 will be scheduled next, ...

Figure 5 illustrate the schedule flow:

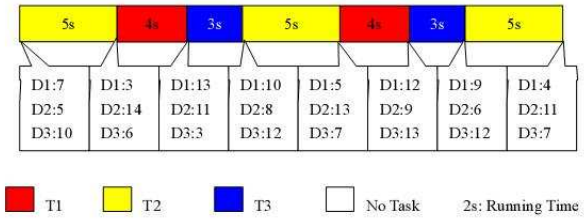


FIGURE 5: *EDF Schedule Algorithm*

5.4.2 EDF algorithm implementation in RTLinux/GPL

Each thread in RTLinux/GPL holds two deadline attribute: one is relative deadline(sched_deadline), other one is absolute deadline(current_deadline). RTLinux/GPL provide two function to set sched_deadline:

```
pthread_attr_setdeadline_np();
pthread_setdeadline_np();
```

there is no API to set the absolute deadline(current_deadline),
function pthread_make_periodic_np() will set the absolute deadline:

```
p->current_deadline = start_time +
    p->sched_deadline  p->period;
```

The core of RTLinux/GPL scheduler is function rtl_schedule(), which will search all the non-blocked thread, and the function call cmp_prio will select the task with highest priority and least absolute deadline(current_deadline).

if ((t->pending & t->blocked) &&(!new_task || (cmp_prio(t, new_task) > 0)))

```
static inline int cmp_prio(pthread_t A,
                           pthread_t B)
{
#ifdef CONFIG_RTL_SCHED_EDF
register int tmp;
if ( (tmp= (A->sched_param).sched_priority -
           (B->sched_param).sched_priority) )
return tmp;
else
return ( (B->current_deadline) >
         (A->current_deadline) );
#else
return (A->sched_param).sched_priority -
       (B->sched_param).sched_priority;
#endif
}
```

System will check the mode to decide whether need to call find_preemptor() to get the preempt time, and the use this time to set the timer

6 conclusion and prospection

This article focused on RTOS education, made an expatiation about RTOS principles, and take RTLinux/GPL as an Example, illustrate the concrete implementation. At present most of the content of RTOS education is about the essential RealTime System principle. It will make the RTOS learning process much easier via illustrate a typical RTOS implementation, and will promote the development of RTLinux/GPL indirectly. Due to time limitation,this article only describe most basic parts, and RealTime communication and process management need to be addressed.

References

[1] Victor Yodaiken and Michael Barabanov, 1997, *RTLinux Version TWO Design documentation about RTLinux in FSMLabs* `http://www.fsmlabs.com`h

[2] Der Herr Hofrat, 2002,*Introducing RTLinux/GPL*, p8

[3] Daniel P.Bovert & Marco Cesati, 2005, *Understanding the Linux Kernel*, O'Reilly & Associates,Inc., 0-596-00565-2.

[4] Yang Lifeng,*Embeded RTOS Development and Design*, `http://www.51kaifa.com/zxyd/read.php?ID=107`

[5] Mao Decao & Hu Ximing, 2001, *Linux Kernel Source Code Scene Analyze*, Zhejiang University Press, 7-308-02704-X/TP.209.

[6] Gary J.Nutt, 2000, *Operating Systems a modern perspective*, Addison Wesley Longman,Inc., 0-201-61251-8.

[7] Amit Choudhary, Nitin Shrivastav, Ramnath Venugopalan, 2002, *Implementation of EDF, PCEP and PIP in RT-Linux Under the guidance of Dr,Mueller*

[8] Patricia Balbastre, Ismael Ripoll, *Integrated Dynamic Priority Scheduler for RTLinux*

[9] Chu Fenmin, Dai Shenghua*Study of RTLinux Scheduling Policy*,MicroComputer Information, 2003, Issue 11.

Porting XtratuM to PowerPC

Zhou Rui, Bai Shuwei, Nicholas McGuire, and Li Lian
Distributed and Embedded System Lab
School of Information Science and Engineering
Lanzhou University, P. R. China
zhour04@st.lzu.edu.cn

Abstract

As a new nanokernel, XtratuM is developed in the framework of the OCERA project [1] for executing several operating systems, with the ability to support real-time operating systems, in the same hardware but running in separate address spaces (temporal and spatial partitioning). XtratuM is a thin virtualization layer or software abstraction virtualizing the essential resources (interrupt, timer, memory, and CPU). To run a kernel, basically the timers, interrupts and address space must be provided. To date XtratuM has been implemented on x86 architecture with support for some versions of Linux kernels, such as 2.4.27 and 2.4.28 - which also are able to run RTLinux/GPL as one rt-domain. Regarding other processor architectures - porting XtratuM to ARM is in process [2]. In this paper, we mainly focus on our work of porting XtratuM to PowerPC based on initial investigation of XtratuM and PowerPC architecture. According to the mechanism of XtratuM to get hardware timer, at present, we've implemented the same function on PowerPC. Then we will discuss about our working process here. Our porting XtratuM to PowerPC will help to expand the applicable area of this free-software nanokernel, which will also promote embedded and real-time applicable abilities based on PowerPC, e. g. the simultaneously running real-time operating systems can sufficiently make use of the powerful and stable performance of PowerPC to guarantee the quality of all these systems. Further more, due to the different performance requirements of various types of embedded and real-time operating systems, using PowerPC processor family scalable from low-end 32bit systems like the PPC405 up to high-end multi-threaded 64bit hardware like the CELL as the target architecture seems a suitable selection.

1 Introduction

For embedded and real-time systems, providing reliable services is very critical. Based on operating system process, the ADEOS (Adaptive Domain Environment for Operating Systems) [3] project has introduced the concept of domain management to developers. The basic concept is the virtualization of interrupts, as this is the primary means of preemption and process control. And the concept of XtratuM, aiming to realize a minimum implementation of ADEOS, has been introduced by the University of Valencia in the framework of the OCERA (Open Components for Embedded Real-time Applications) project. In OCERA, XtratuM is developed as a thin but important layer able to communicate directly with the hardware. For the latest XtratuM-0.3, it can support several real-time operating systems, in the same hardware but running in separate address spaces (temporal and spatial partitioning). XtratuM will provide a flexible environment for sharing hardware resources among multiple operating systems. In this case, XtratuM enables multiple kernel components called domains or guest operating systems to exist simultaneously on the same hardware. There is no need for these domains to see each other, but all of them must be aware of XtratuM (paravirtualized guest OS). As a very light weight approach, XtratuM provides a simple and convenient API to access hardware with very reduced service primitives, so it pays no attention to the complex implementation of higher level domains. The most attractive idea of this type of domain management such as XtratuM for us is that the abstraction layer itself is a simple, clear and well portable layer. For XtratuM, all versions of it have the same API, so once the RTOS to be ported or developed to XtratuM, all these benefits of XtratuM will be available just with recompiling the guest OS. Further more XtratuM is designed to be able to replace the current HAL (Hardware Ac-

cess Layer) of RTLinux/GPL, which eliminates the RTLinux patents concerns.[2]

But why do we need the support of XtratuM for PowerPC? It seems it's a good general method for porting guest operating systems, especially RTOS to hardware with the help of XtratuM and as PowerPCs are commonly used in the RTOS application domain, this port has been initiated. However, at present, XtratuM has been only implemented on x86 architecture. Other processor architectures are under way, i.e. porting XtratuM to ARM. So as a newly developed helpful tool, it seems natural for XtratuM to expand to more architectures than now. Let's consider from the point of view of hardware: due to the nature of many embedded systems, especially systems with real-time demands and the demand for high reliability along with low power consumption of them, a wide performance range is compulsive. As a highly scalable processor family, the series of PowerPC are very likely to overcome these primary constraints well. So PowerPC selected as the target architecture scalable from low-end 32bit like the PPC405 up to multi-threaded 64bit systems like the CELL seems an appropriate choice. Also, The PowerPC processor is well suited for fanless operations in extended temperature ranges, which is an important feature for high stability and safety critical systems. By porting XtratuM to PowerPC we hope to simplify porting embedded and real-time operating systems to PowerPC based on free-software. For the other thing, it'll expand compatibility of PowerPC for embedded and real-time operating systems, which will be good for the future market of PowerPC in control and automation related areas.

2 Poring Process

Before porting, we conducted an in depth analysis from the hardware and software point of view. For XtratuM itself, it provides a simple and convenient API to access interrupt mechanism and timer devices, which is basic service that XtratuM provides to the system. As interrupt and timer are essential devices for the kernel, it makes XtratuM operate as a supervisor system which is executed at the highest priority to handle the domains similar to the way an operating system manages a process: the OS can create, suspend, kill, etc. each process. So far, we can see clearly that these features and functions of XtratuM closely related to interrupt and timer devices, further more virtual memory management, not necessarily simplifies the porting XtratuM to new processor architectures, simplifies porting guest OS to XtratuM. Therefore, it's obviously necessary to get clear about how XtratuM manipulates interrupt and timer, and how interrupt and timer operates in PowerPC architecture. During the porting, the embedded development boards based on PowerPC4xx processor are being used, but due to the API compatibility within the PowerPC family of processors extending to other PowerPC CPUs should be fairly simple.

2.1 Timer

The current x86 implementation of XtratuM can support two types of timer: PIT (Programmable Interval Timer) for single processor and APIC (Advanced Programmable Interrupt Controller) for SMP (Symmetric Multi-Processor). The timer handler of XtratuM provides the low level hardware timer API via functions like read_hwtimer(). For different domains on XtratuM, there are some rules on how to coordinate them. One way is that they can be sorted by virtual timer as a domain heap. For each domain, the virtual timer can be created by functions like xm_sethwtimer(), based on the hardware timer got by timer handler. Also, timer interrupt generated by PIT and APIC will be taken over by the XtratuM nanokernel, which results in less latency than if handled by the individual domains.

XtratuM provides at least one virtual timer, and the exact number of timers implemented by it depends on the available number of hardware timers. In PowerPC, four timer facilities: a TB (Time Base), a PIT (Programmable Interval Timer), a FIT (Fixed Interval Timer), and a WDT (Watchdog Timer) are provided. WDT aids system recovery from software or hardware faults. However, it is not really a timer for XtratuM due to its dedicated function and rough granularity but rather a device. It's relatively slow and can deliver interrupts in the range of seconds. The FIT provides timer interrupts having a repeatable period. FIT is functionally similar to an autoreload PIT, except that only a smaller fixed selection of interrupt periods are available limiting its usability in the general case. So the main time base on PowerPC is still the PIT. Besides, to work with these virtual timers, XtratuM also provides a high-level API to deal with them.

It's obviously for the difference between PowerPC and x86, there must be different ways to access and manipulate the timers. For example, in XtratuM x86 implementation, timer handler uses read_hwtimer() to get hardware timer. In detail, it calls read_TSC(), which runs x86 assembly instruction "rdtsc" to get the value. In x86, TSC (Time Stamp Counter), which is a 64-bit variable, is incremented every CPU tick (1/CPU_HZ). For example, at 1GHz CPU, TSC is incremented by 109 per second. PowerPC provides similar capability as Time

Base, which is also 64-bit, but this kind of time counter is increments at either equal the processor core clock rate or as driven by a separate timer clock input. So in order to make read_hwtimer() function correctly in PowerPC, read_TB() may need implemented by our porting instead of read_TSC() and run PowerPC assembly instruction as "mftb" or "mfspr" in it. PIT in PowerPC is a 32-bit SPR (Special Purpose Register) that decrements at the same rate as Time Base. And similar to Time Base, it is read using "mfspr". When a non-zero value is written to the PIT, it begins to decrement. A PIT event occurs when a decrement occurs on a PIT count of 1.[7]

Basically, the following code section implements geting the hardware timer from PowerPC in an XtratuM complient way.

```
#define rdtb(val) __asm__ __volatile__
    ("mftb \%0" : "=r" (val))
typedef signed long long hwtime_t;
static hwtime_t read_TB (void)
{

    hwtime_t t;
    rdtb(t);
    return (hwtime_t) t;
}
```

So read_hwtimer() in XtratuM for PowerPC can call this read_TB() to get the hardware timer from PowerPC for timer setting and manipulation of further real-time tasks.

2.2 Interrupt

XtratuM manages all hardware interrupts available on the actual hardware architecture being used. Once XtratuM is implemented, all interrupts of any domains will pass through it. In the current XtratuM x86 implementation, these interrupts are controlled by XtratuM. XtratuM is placed between the hardware and the domains requiring virtualizing the interrupts, thus preventing the domains from directly accessing interrupt related functions.

In XtratuM x86 implementation, this is done by xm_arch_takeover() from the init module function in XtratuM.o. But for PowerPC, the interrupt mechanism is obviously different from x86, so our porting needs to enable XtratuM to handle and virtualize PowerPC interrupt appropriately. In XtratuM x86 implementation, at first, xm_arch_takeover() will simply save the real interrupt management functions (those that actually modify hardware settings) and replace them with the virtual interrupt management functions - those that only call XtratuM primitives for manipulation of XtratuM internal state. Then, the IDT (Interrupt Descriptors Table) entries, which contain the address of the interrupt handlers, are replaced with the XtratuM interrupt handler addresses. Once the IDT has been modified, interrupts are completely managed by XtratuM. XtratuM's IDT version is used to hold the entry points to the real hardware functions, basically copied from what ever the domains provide, and the domains get the entry points replaced by calls to virtual interrupt functions. XtratuM has already prepared a private IDT named root_idt_table used within the root domain to replace the real IDT, and xm_irq_addr is used to save the address of the XtratuM IRQ (Interrupt ReQuest) handlers. The real IDT named real_idt_table is then set up with calls to the hw_set_irq_gate, which assigns XtratuM's new IDT to the respective vectors. At this point a hardware interrupt would no longer reach the domain entries but get the handler from the xm_irq_addr. [5][6]

The difference is that PowerPC has no IDT, and only provides a single interrupt pin (this point is the same as x86) and interrupt vector. This means for PowerPC, XtratuM has to manually read the vector number from hardware, which is also the same as for x86, then look up and call the service routine connected to this vector number. The essential difference is that for PowerPC, a UIC (Universal Interrupt Controller) provides all necessary control, status, and communication between the various internal and external interrupt sources and the processor core. UIC contains registers like UIC0_SR (UIC Status Register), UIC0_ER (UIC Enable Register), UIC0_CR (UIC Critical Register), UIC0_PR (UIC Polarity Register), UIC0_TR (UIC Trigger Register), UIC0_MSR (UIC Masked Status Register), UIC0_VR (UIC Vector Register) and UIC0_VCR (UIC Vector Configuration Register). And the privileged "mtdcr" and "mfdcr" instructions of PowerPC are used to read and write the UIC registers. So our XtratuM for PowerPC needs to get the interrupt vector information from UIC and takeover these interrupts. UIC0_VCR can contain either the base address for an interrupt handler vector table or the base address for the interrupt handler associated with each interrupt. The actual interrupt vector, with the address of the interrupt handler that services the interrupt, is generated in UIC0_VR. So in order to get interrupt handler address for virtual interrupt, XtratuM needs to access UIC0_VR to fetch the address and save it in some place like xm_irq_addr for the domains using. [7]

We also must consider the interrupt events for porting. In XtratuM x86 implementation, there are 17 reserved hardware interrupts and 15 domain defined events. However, in PowerPC, UIC also assigns

32 interrupts including 23 internal interrupts, 7 external interrupt and 2 reserved interrupts. Therefore, the redefine of interrupt event types in porting process is of course indispensable.

2.3 Patching Kernel

XtratuM is in the form of a patch for the common Linux kernel. This patch modifies the Linux kernel in two ways: replacing all the assembler disabling/enabling interrupt instructions by calls to XtratuM hooks (which initially point to the default Linux functions) and inserting several hooks in the Linux kernel code to virtualize low level hardware functions. These hooks will be used later to virtualize the interrupts and the hardware timers. At present, the latest version XtratuM-0.3 only can support Linux kernel 2.4.X. But for current PowerPC embedded development boards, basically 2.6.X kernels are more or less mandatory. For the first steps a currnet 2.4.X PowerPC kernel [4] seems sufficient for our PPC405 boards. Further more, Linux kernel 2.6.x has also been released a long time ago. If we plan to implement our porting based on this series of kernels, of course we must think over the difference between Linux kernel 2.4.x and 2.6.x carefully.

Based on above analysis for our porting, we can say the main procedure of porting XtratuM to PowerPC is like this:

- Get familiar with the principles and specifications of XtratuM

- Analyzed the fundamental hardware specifications of PowerPC4xx processor core used

- Check out the sections closely related to x86 features in current XtratuM x86 implementation, especially the x86 assembly code sections

- Implement these sections to fit the specifications of PowerPC architectures.

- Develop the patch for the mainstream Linux kernel to provide the needed XtratuM interface.

- Test and benchmark the port, and feed back results to the continued code development

3 Status

The port kickstarted only a few months ago. At the beginning, we installed XtratuM-0.3 on an x86 machine with Slackware. Then we focused on analyzing the source codes and examples it provided to get clear about the principles and specifications of XtratuM. Also we're trying to get familiar with the fundamental hardware specifications of PowerPC4xx processor core used for porting. This initial investigation of XtratuM goes on well and this paper is mainly based on it. Another important issue for such a project are the tools. As the porting is closely related to hardware, introducing and selecting correct tools plays an import role in the process. We've got a BDI2000 for hardware debugging, and tried to implement KGDB between x86 host and PowerPC target. It seems both of these two tools are essential for the porting process as low level hardware related faults can't be detected easily from within a faulting OS-nanokernel.

The porting process is being documented and run under project management, documents accumulated during this effort are available at:

`http://dslab.lzu.edu.cn/~zr/xtratum_ppc.html`

and should finally be merged with works at `http://www.xtratum.org` once the basic port is ready for release.

4 Perspectives

XtratuM is a new approach to real-time Linux, and our porting to PowerPC is new both for XtratuM itself and for PowerPC. It seems the porting will be a good combination for embedded and real-time operating systems applications and the performance of PowerPC, so both XtratuM and PowerPC can benefit from this once accomplished. Further more, as XtratuM is one of the fundamental components in OCERA, as OCERA anticipates (and partially provides) support for PowerPC this is a quite natural next step in the evolution of XtratuM. It'll be helpful for application and development of embedded and real-time operating systems especially in the safety critical domain.

Based on our porting process, except the initial investigation of XtratuM, the know-how of PowerPC architecture, especially the PowerPC4xx processor and embedded development boards used here, we hope that fully documenting the steps we take will help others extend XtratuM to further architectures. Based on these work, we need to do:

- XtratuM ported to PowerPC4xx processor board

- Implementation of a minimum standalone domain on XtratuM/PowerPC (32bit)

- Case study based on XtratuM on PowerPC4xx processor board

- Technology documentation in a publishable form

- Introductory documents allowing usage of this new technology

So porting is just the first important step for the whole development of XtratuM combined with PowerPC.

5 Acknowledgements

This work was supported by National Natural Science Foundation of China through projects: Research on Computational Chemistry E-SCIENCE and its Applications (Grant no. 90612016) and Research on Job Scheduling in network computing (Grant no. 60473095). And this work is also supported by China National Technology Platform.

Thanks a lot to Prof. Nicholas Mc Guire, Prof. Li Lian, Dr. Zhou Qingguo and all the other people of DSLab, SISE, Lanzhou University. Their help makes me know more and more about free-software and my work go on the right track.

References

[1] Maintainer: Miguel Masmano. http://www.ocera.org

[2] Miguel Masmano, Ismael Ripoll, & Alfons Crespo. *XtratuM Overview.* http://www.xtratum.org/papers/Intro/xtratum_intro.html

[3] Maintainer: Philippe Gerum. http://home.gna.org/adeos

[4] PowerPC Linux, Denx Software. http://www.denx.de

[5] Nicholas Mc Guire. 2006. *XtratuM Hardware Initialization.* http://dslab.lzu.edu.cn/docs/publications/arch_init.pdf

[6] Nicholas Mc Guire. 2006. *XtratuM-0.3 Kernel Modifications overview.* http://dslab.lzu.edu.cn/XtratuM.html

[7] International Business Machines Corporation. 2003. *PowerPC 405EP Embedded Processor User's Manual Preliminary.* p269–278

Embedded RTLinux: A New Stand-Alone RTLinux Approach

Miguel Masmano[1], Apolinar González[2], Ismael Ripoll[1], and Alfons Crespo[1]
[1] Universidad Politecnica de Valencia, Spain
[2] Universidad de Colima (Mexico)
mimastel@doctor.upv.es apogonl@ucol.mx
{iripoll, alfons}@disca.upv.es

Abstract

RTLinux is a hard Real-Time OS (RTOS) which uses an original approach to develop complex hard real-time applications in a fairly easy way: Executing a hard RTOS (RTLinux itself) jointly with Linux in the same box. This approach enables splitting up a hard real-time application according to its criticality, the part with deadline requirements is run on the RTOS while the part with no "special" time requirements is executed on Linux.

However, this approach presents two drawbacks. Firstly, Linux has a big footprint, especially in the 2.6 series, which is unsuitable for system with low resources. On the other hand, it is possible for a Linux application to directly disable interrupts, disrupting the real-time properties of RTLinux.

Several years ago, people from our department developed Stand-Alone RTLinux (SA-RTL), a RTLinux-compatible standalone kernel which was able to execute RTLinux applications in a bare machine. However, this approach was not completely optimum since was built cutting and pasting code from RTLinux, being rather difficult to maintain. Besides, it did not provide binary compatibility with RTLinux applications.

In this paper we present Embedded RTLinux (ERTL), our second approach of what a Stand-Alone RTLinux system should be. This second kernel, unlike SA-RTL, just replaces Linux with a minimal set of drivers, allowing to execute the original RTLinux jointly with a RTLinux application on a bare machine.

1 Introduction

[1]

The aim of a general purpose operating system (OS) is to provide a good average response time, not to guarantee maximum bounds of execution time. This makes general purpose OSes inadequate to achieve hard real-time performance, with guaranteed deadlines.

On the other hand, besides running correctly an application, the goal of a real-time operating system (RTOS) is to fulfil the timing requirements of the application.

The most RTOSes often provide a spartan programming interface with just the fairly necessary primitives (thread, I/O, and synchronisation procedures), avoiding other interesting features (network, graphic interface, etc) in benefit of the time determinism. Thus, turning the implementation of a complex application into a very tedious task.

RTLinux is a hard RTOS which works around this drawback by using an innovative approach. It executes a RTOS (RTLinux itself) and a general purpose OS (usually Linux) in the same box. The RTOS is who actually manages the whole physical hardware while the OS is executed over a virtual hardware layer. The general purpose OS is run on the background as the least priority thread of the RTOS.

RTLinux enables and eases splitting up a quite complex, hard real-time application on two or more logical parts according to their timing requirements. Executing, subsequently, each of these parts in the most suitable kernel. For example, an application to control a motor can be easily divided in the control routines of the motor, with hard real-time constraints, which will be executed on RTLinux. And the user interface, without any timing requirement, which will be run on Linux with all the advantages that it represents.

However, although this approach presents multiple benefits, it also shows some drawbacks. Firstly, the large footprint of the Linux kernel (up to several Mbytes) can suppose a problem for system with low

[1]This work has been supported by the Spanish Governement Projects: TIC2005-08665 and DPI2005-09327

resources. Secondly and more important, any Linux application with root permissions can disable interrupts, disrupting the normal operation of RTLinux.

Several years ago, to work around these problems, some people of our research group proposed a stand-alone version of RTLinux. This new version of RTLinux, called SA-RTL, would replace the original RTLinux when its use were not suitable. However, SA-RTL shows two important disadvantages. On the one hand, since it was built cutting and pasting code from a specific version of RTLinux, it is hard to keep it up-to-date. On the other, SA-RTL does not implement binary compatibility with RTLinux applications but source-level compatibility, forcing to recompile the whole RTLinux application.

In this paper, we present a new stand-alone version of RTLinux called Embedded-RTLinux (ERTL for short), unlike SA-RTL, ERTL does not implement a new version of the RTLinux kernel but it replaces the whole Linux kernel with a thin software layer which enables executing directly RTLinux and a RTLinux application on a bare machine.

This paper has been organised as follow, next section shows a brief overview of RTLinux. Section 3 describes a previous work carried out by our grup the Stand-Alone RTLinux. Section 4 details the eRTL design criteria. Section 5 performs an evaluation of eRTL. Finally, some conclusions are drawn.

2 Overview of RTLinux

Linux is a general purpose OS with multiprocessor support, management of large amounts of memory, efficient handling of a large number of running processes, etc. In fact, nowadays, Linux is a full-featured, efficient (good throughput) OS, but with little or no real-time performance.

Currently, there are two different approaches to provide Linux with real-time performance:

1. Modifying Linux to make it a kernel with real-time characteristics. The modification would consist in adding more **predictability**, more **responsibility**, and a **real-time API**.

2. Adding a new operating system (or executive) that takes over some hardware subsystems (those that compromise the real-time performance) and who gives Linux a virtualised view of the controlled devices.

RTLinux was the first project following the second proposed approach: Linux as the general purpose OS and a newly developed executive (RTLinux itself) running underneath Linux.

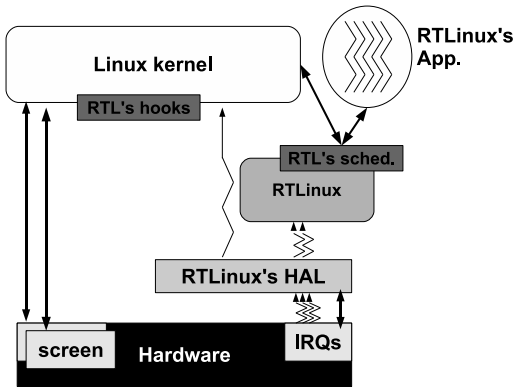


FIGURE 1: *RTLinux's architecture*

The RTLinux internal architecture, figure 6, can be split up into two separate parts: a hardware abstraction layer (the RTLinux's HAL), and the executive operating system.

The RTLinux's HAL[2] is in charge of the hardware devices that conflict with Linux (basically, the interrupt controller and the hardware timers). This layer provides access to these hardware devices to both the RTLinux executive and Linux in a prioritised way. Hardware virtualisation is achieved by directly modifying the Linux kernel sources applying a patch file. These changes do not replace the hardware drivers but prepare the kernel, adding hooks (see section 2.1), to dynamically replace the drivers.

The RTLinux executive implements a partial Minimal Real-Time POSIX.13 interface. The executive is highly customisable. It has been implemented in a modular way that allows to dynamically load into the system the parts of the API that are used by the application.

2.1 Taking over the Linux Kernel: The RTLinux's hooks

As mentioned above, some Linux's drivers (basically, interrupts and timers) are modified to directly deal with the RTLinux's HAL instead of with the hardware. These modifications are carried out inserting *hooks* (in the RTLinux terminology, a hook is a pointer to a function initially initalised to null) in the Linux's source code. The execution of a hook is only performed when the hook points out to a valid function (actually, the hook will be called always that its value is not null).

[2]The mechanism of intercept hardware access off a general purpose OS to implement a real-time layer is covered by U.S. Patent No. 5,995,745, titled: "ADDING REAL-TIME SUPPORT TO GENERAL PURPOSE OPERATING SYSTEMS". The owner of the patent permits the use of the patented method if the software is released under the General Public License (GPL). For more details read "The RTLinux Open Patent License, version 2.0" [8].

The next example, extracted from the file *entry.S* of an already-patched Linux kernel, shows an example of how a hook in RTLinux works:

```
movl rtl_emulate_iret, %eax;
testl %eax, %eax;
je 1f;
call *%eax;
1:
```

A C translation of this assembly code could be:

```
if (rtl_emulate_iret )
  (*rtl_emulate_iret )();
```

This hook, `rtl_emulate_iret`, emulates an interrupt return when the processor winds up the execution of the interrupt handlers, as can be seen, rtl_emulate_iret will be only executed when its value is not null.

These hooks are strategically inserted into the Linux kernel code, enabling RTLinux to take the control of the machine once it is initialised.

The most important hooks inserted in the Linux kernel are:

- `rtl_emulate_iret`: emulates the behaviour of the *iret* instruction, that is, enabling interrupts if they were on before the interrupt was triggered.
- `rtl_syscall_intercept`: intercepts the execution of Linux's System Calls.
- `rtl_exception_intercept`: intercepts the execution of traps (memory fail, segmentation fault, division by zero, ...).
- `rtl_do_exit_handler`: intercepts the finalisation of a Linux's process.

Besides these hooks, RTLinux also replaces `cli` (disable interrupts) and `sti` (enable interrupts) with fake ones which will emulate their operation, preserving, thus, RTLinux real-time capabilities.

3 Previously existing work: SA-RTL

Several years ago, some people of our department implemented an initial stand-alone version of RTLinux: SA-RTL [7]. The aim of this project, developed in the OCERA [1] framework, was to implement a small-footsized kernel which were RTLinux-complaint, allowing the execution of RTLinux applications on a bare machine.

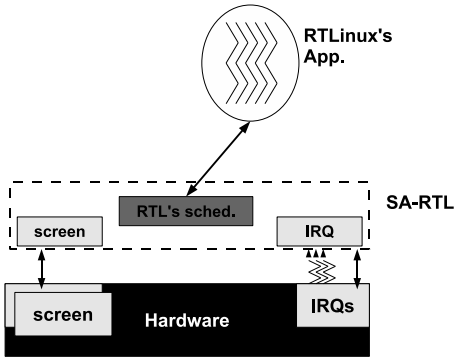


FIGURE 2: *SA-RTL's architecture*

SA-RTL, figure 2, is a small kernel, which implements the minimal quantity of drivers to: boot the machine, manage interrupts and traps, and write on the screen. Besides, to provide a better RTLinux compatibility, some code (basically, the scheduler) of RTLinux has been directly used in SA-RTL.

However, this approach has two manifest drawbacks:

- SA-RTL is only compatible with RTLinux applications at source level but not at binary level. In other words, to run a RTLinux application, its source code must be available and has to be recompiled.
- SA-RTL was built cutting and pasting code from a specific version of RTLinux-GPL[3]. Thus, complicating the support and upkeep of the kernel when a new version of RTLinux-GPL is released.

ERTL also implements some features that did not exist in the original RTLinux, among others, inter-threading memory protection, that is, the ability of executing each thread in an isolate memory space. These features, although interesting and funny, are not either POSIX standard or RTLinux standard. An application spending these capabilities will not be compatible with other RTOSes.

4 ERTL overview

Embedded RTLinux (ERTL for short) is our second, improved approach of what a stand-alone RTLinux should be.

Unlike SA-RTL, ERTL (figure 3) does not implement a new version of RTLinux but replaces the whole Linux kernel with a thin software layer. ERTL

[3]RTLinux-GPL is the GPLed version of RTLinux, currently maintained by the RTLinux community.

provides, on the one hand, the minimum functionality to: boot the machine, manage interrupts and traps, and read from the keyboard and write to the screen. And, on the other, the RTLinux's hooks, allowing, thus, the execution of an unmodified version of RTLinux on its top.

The main advantage of ERTL is that it is compatible with RTLinux (in fact, it executes the actual RTLinux once it has already been compiled for Linux) and the RTLinux applications at binary level, what means that any application that already-worked for RTLinux will work on ERTL.

Currently, ERTL is only available for the i386 architecture. However, due to its simplicity, it should be rather portable to other architectures where RTLinux is already working.

4.1 ERTL architecture

ERTL, figure 3, provides, on the one hand, the minimum functionality to:

- Boot the machine: ERTL takes advantage of the Multiboot Specification [3]. Hence, the code involved in this process is quite simple and just has to:
  1. Create a valid Global Descriptor Table (GDT) with just two segments, a code and a data segment.
  2. Create a valid stack.
  3. Set up the ERTL's memory manager with the free memory reported by Multiboot.
  4. Jump to the first C routine.
  5. Call RTLinux's init_module functions whose address were gathered during the ERTL kernel building process (see section 4.3).
- Manage interrupts/traps: when an interrupt or a trap is triggered, ERTL emulates the Linux kernel's behaviour, that is:
  1. Store the interrupt number.
  2. Save all registers calling the Linux's SAVE_ALL macro.
  3. Call a routine called `do_IRQ`. This function deals with the i8259 driver to re-enable this device and allow future interrupts.
  4. Restore all registers calling the Linux's RESTORE_ALL macro.
  5. Call the RTLinux hook's `rtl_emulate_iret` as described in the section 2.1.
  6. Execute `iret`, restoring, thus, the execution of the current application.
- i8259 driver: This driver is a set of functions packed in the `irq_desc` structure which allows to deal with the programmable interrupt controller (i8259). RTLinux uses them to control hardware interrupts.
- Dynamic memory manager: ERTL emulates the behaviour of the `vmalloc` and `vfree` via the TLSF [5, 6] dynamic memory allocator.
- Idle task: This task emulates the whole Linux kernel, that means that this task will be executed with the lowest priority task of the system. Its only functionality is to execute RTLinux's soft interrupts as the Linux kernel would do. For instance, printing on the screen every time a RTLinux's application uses the `rtl_printf` routine.
- Screen driver: A simple screen driver which permits ERTL and RTLinux writing on the screen.

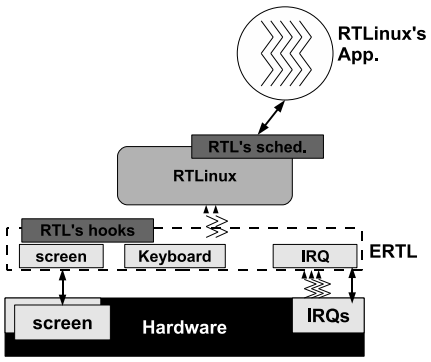


FIGURE 3: *ERTL's architecture*

4.2 Emulating Linux

In order to allow the execution of RTLinux, the following hooks have been inserted inside the ERTL's code:

- `rtl_emulate_iret`: This hook is executed just before calling the `iret`, emulating, thus, the behaviour of this instruction.
- `rtl_exception_intercept`: This hook is called just after a trap has been "issued", enabling RTLinux to manage it before Linux's trap handlers do.

The following symbols are also exported to RTLinux:

- `irq_desc`: This structure packs all functions related with the i8259 driver. RTLinux does not implement the i8259 driver, therefore, needs to use the driver provided by Linux.
- `do_IRQ`: During its initialisation, RTLinux replaces the function `do_IRQ` with its own interrupt handler.

Since ERTL is completely RTLinux-aware, it is unnecessary to patch the `cli` and `sti` instructions, since they are not executed in an indiscriminate way.

4.3 Building an ERTL kernel

ERTL provides the script *build_ertl* to ease the generation of an ERTL kernel.

The figure 4 reflects the process of building a ERTL kernel when *build_ertl* is invoked with the suitable parameters. The process can be summarised as follow:

1. Firstly, each RTLinux module file (rtl.o, rtl_sched.o, rtl_time.o, ...) and the RTLinux application are processed to rename the `init_module` routine to `init_file_name`. For example, the `init_module` function of a RTLinux module called *hello.o* will be renamed to `init_hello`. Besides, the address of these routines are written down in an external file.
2. The RTLinux module's files and the RTLinux application are linked together creating a single object file.
3. The object file resulted in step 2 is linked with the ERTL library, generating, thus, a working ERTL kernel.

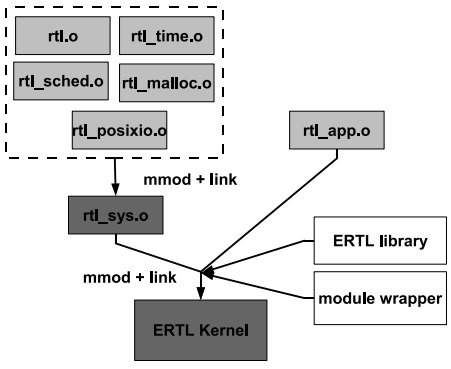


FIGURE 4: *Building process of an ERTL kernel.*

As mentioned above, ERTL implements the multiboot specification, therefore, to boot an ERTL kernel, it is necessary to use a multiboot-compliant boot loader, such as GRUB [2] and Etherboot [4].

5 Evaluation of ERTL

ERTL has been evaluated to measure the stability and the overhead introduced by the kernel. The test platform is an embedded system based on Pentium II at 300 MHz. The system has 64Mb of memory.

5.1 Stability evaluation

In order to evaluate the stability the kernel, we have evaluated the its latency. This evaluation has consisted in a periodic task which program a delay and waits for it. As soon as the task is executed, the task reads the clock value and calculates the difference between the programmed delay and the real clock, then the task period is increased in order to evaluate the kernel latency from 100 nanoseconds to 1 seconds.

```
task body task_estad is

Period: Time_span:=nanoseconds(100);
Period_Cte: integer:=1;
Period_Final: constant time_span;
next, clk: Time;
error: Time;
Periodo_Int: Integer;

begin
  while Period < Period_Final loop
     next:=clock;
        for I in 1..100 loop
            clk := clock;
            error := error + (clk−next));
            next := next + Period;
            delay until next;
         end loop;
      Put_Time(error));
      Period := Increment_Period(Period);
  end loop;
end task_estad;
```

Next figure shows the latency measured in the experiment.

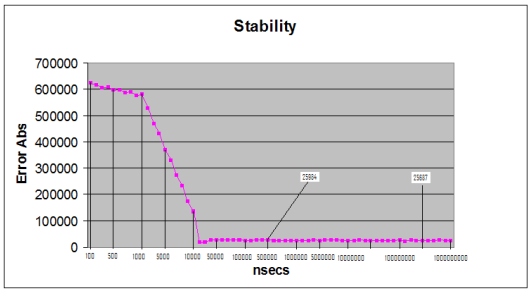


FIGURE 5: *Stability measure*

As can be seen in the plot, the periods of 10 microseconds have constant latency (25 microseconds). Nonetheless, lower periods show higher latencies because of the time taken to program the system timer.

5.2 Overhead measures

The overhead tries to measure the cost of the scheduling algorithm. In order to avoid interferences, a small application has been built to send through a digital output signals to an oscilloscope. Overhead values has been measured from the oscilloscope.

Next figure shows the measures carried out.

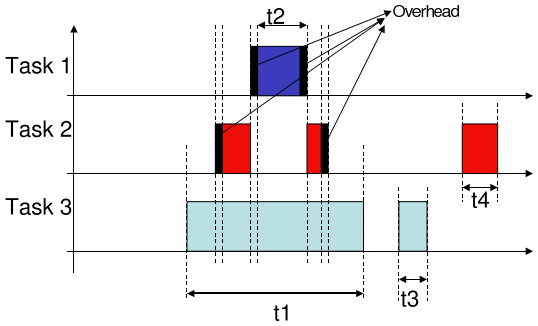


FIGURE 6: *3 tasks overhead measure*

The measure obtained has been:

$Overhead = (t1 - t2 - t3 - t4)/4 = (100\mu s - 300\mu s - 300\mu s - 300\mu s)/4 = 25s$

Another way to measure the overhead has been performed using the Hartstone benchmark[9]. This test tries to test the ability of a system to handle hard real-time applications under well-defined workloads and timing constraints.

The workload is composed by 5 harmonic tasks which are initialised to a lower period. Starting with the baseline task set, all the frequencies are scaled by 1.1, then 1.2, then 1.3, and so on for each new test until a deadline is missed. The per-period workloads of all tasks do not change. The scaling preserves the harmonic frequencies. A test finishes when a task misses its deadline. At this point we measure the CPU utilisation. In the previous version of the SA-RTL we obtained a 99,2 % of the CPU while the value obtained by ERTL is 99,6 %. The test has been written in Ada.

6 Conclusions

In this paper we presented Embedded RTLinux (ERTL), a new approach of what a Stand-Alone RTLinux system should be. This second kernel, unlike SA-RTL, just replaces Linux with a minimal set of drivers, allowing to execute the original RTLinux jointly with a RTLinux application on a bare machine.

ERTL provides the minimum functionality to: boot the machine, manage interrupts and traps, and read from the keyboard and write to the screen. And the RTLinux's hooks, allowing, thus, the execution of an unmodified version of RTLinux on its top.

The main advantage of ERTL is that it is compatible with RTLinux (in fact, it executes the actual RTLinux once it has already been compiled for Linux) and the RTLinux applications at binary level, what means that any application that already-worked for RTLinux will work on ERTL.

ERTL supports the same languages that RTLinux: C, C++ and Ada.

The stability and overhead measures showed that these measures are a bit lower than RTLinux and SA-RTL.

References

[1] *OCERA: Open Components for Embedded Real-Time Applications*, 2002. IST 35102 European Research Project. (http://www.ocera.org).

[2] Yoshinori K. Okuji, *GRand Unified Bootloader*, available at http://www.gnu.org/software/grub/

[3] Yoshinori K. Okuji, Bryan Ford, Erich Stefan Boleyn, and Kunihiro Ishiguro. *The Multiboot Specification*, available at http://www.gnu.org-/software/grub/

[4] *EtherBoot Project*, available at http://www.-etherboot.org/

[5] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real, *TLSF: a New Dynamic Memory Allocator for Real-Time Systems*, Proc. of the 16th Euromicro Conference on Real-Time Systems.

[6] Miguel Masmano, Ismael Ripoll, and Alfons Crespo, *Dynamic storage allocation for real-time embedded systems*, Proc. of Real-Time System Simposium WIP.

[7] Vicente Esteve, Ismael Ripoll, Alfons Crespo, *Stand-Alone RTLinux-GPL*, Fifth Real-Time Linux Workshop.

[8] FSMLABS, *The RTLinux Open Patent License, version 2.0*, available at http://fsmlabs.com/-products/rtlinuxpro/rtlinux_patent.html

[9] Nelson Weiderman. *Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Application*, 1990 Technical report CMU/SEI-89-TR-23.

A Clock Synchronization Skeleton Based on RTAI

Yang Huang, Peter Visser, and Jan Broenink
University of Twente
Enschede, the Netherlands
Y.Huang@alumnus.utwente.nl
P.M.Visser@ewi.utwente.nl
J.F.Broenink@ewi.utwente.nl

Abstract

This paper presents a clock synchronization skeleton based on RTAI (Real Time Application Interface). The skeleton is a thin layer that provides unified but extendible interfaces to the underlying operating system, the synchronization algorithms and the upper level applications in need of clock synchronization. The skeleton provides synchronization support to a system, whereby the achieved accuracy is the best obtainable given this software structure. By connecting an algorithm and a communication module with the skeleton, a system becomes capable to run with synchronization support. To demonstrate and validate the design, the skeleton has been tested successfully with two different synchronization algorithms based on the CAN bus. Other algorithms and communication technologies can also work with the skeleton, as long as they provide the necessary functionalities for clock synchronization.

1 Introduction

A distributed system is a collection of independent computers that appears to its users as a single coherent system [1]. However, due to the difference between the independent computers in the software application, the hardware platforms, and working environments, local clocks in these computers will deviate from each other. To achieve monolithic performance, a distributed system needs clock synchronization support, which is a key issue when constructing such a system.

A lot of research has been performed to provide an accurate and efficient software algorithm for the clock synchronization between different nodes in a distributed system. These algorithms include the classical Cristian's Algorithm [2], the Berkeley Algorithm [3], the optimal clock synchronization algorithm [4] and the algorithm providing self-stabilizing clock synchronization [5] etc. These algorithms realize the clock synchronization of a system in different manners and have been successfully applied in many applications.

However, in order to realize a synchronization algorithm in a distributed system, one has to implement not merely the algorithm, but also the corresponding mechanism in the system, including the synchronization message transmission, synchronized clock supply etc. The mechanisms in different applications are similar to each other, independent of the communication technologies and the algorithms. To avoid this repeated work, a thin layer including these mechanisms to achieve clock synchronization is needed. With the layer, a synchronization algorithm can be applied directly in a system by only describing the algorithm itself. The essential and common functionalities are provided by the layer. On the other hand, in case the communication layer is changed, the algorithms are unaffected. The corresponding interfaces between the algorithms and the platform need to be modified. In this way a modularized synchronization scheme is realized.

In this paper such a layer is presented, as a wrapper around the synchronization algorithms. It is called the *Synchronization Skeleton* and provides unified but extendible interfaces both between the user applications and a synchronization algorithm, and between the algorithm and the underlying communication layer. Therefore, a synchronization algorithm can be implemented independent of the outside world. Also, modification of either user application or communication mechanism will not effect the functioning of the algorithm. RTAI (Real-Time Application Interface) is used as the real-time oper-

ating system for the implementation of the *Synchronization Skeleton*.

In section 2, the synchronization skeleton is presented. Section 3 shows the experimental results, and conclusions are drawn in section 4.

2 Synchronization Skeleton

In this section, the conceptual description gives an overview of the *Synchronization Skeleton*, followed by a detailed introduction of the entire mechanism, including its structure, the clock flow in the mechanism and the analysis on the clock accuracy of the skeleton.

2.1 Conceptual Design

The component diagram of the skeleton is shown in Figure 1.

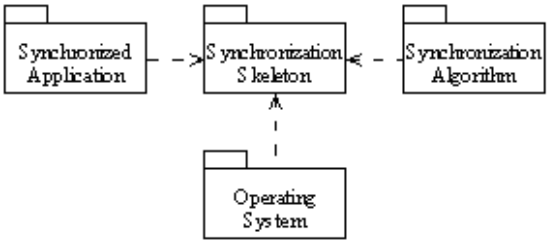


FIGURE 1: *Component Diagram of the Synchronization Skeleton*

From the top point of view, the *Synchronization Algorithm* describes the policy to realize the clock synchronization, while the *Synchronization Skeleton* provides the *Synchronized Applications* with the globally synchronized clock of the entire system. By reading the clock, applications in different nodes of the system will be kept synchronized with each other. The basic functionalities such as task scheduling, mutual exclusion, timing and communication between nodes are provided by the *Synchronization Skeleton* based on the *Operating System*. The *Synchronization Skeleton* is connected to the other three packages by a set of programming interfaces. Modification of *Synchronized Application*, *Synchronization Algorithm* or the underlying communication technology in the *Operating System* will not affect the other components. Only the corresponding interfaces in *Synchronization Skeleton* need to be changed.

2.2 Structure of the Skeleton

The structure of the *Synchronization Skeleton* is depicted in Figure 2. Except the *Synchronization Skeleton*, the blocks in the kernel space belong to the *Operating System* in Figure 1.

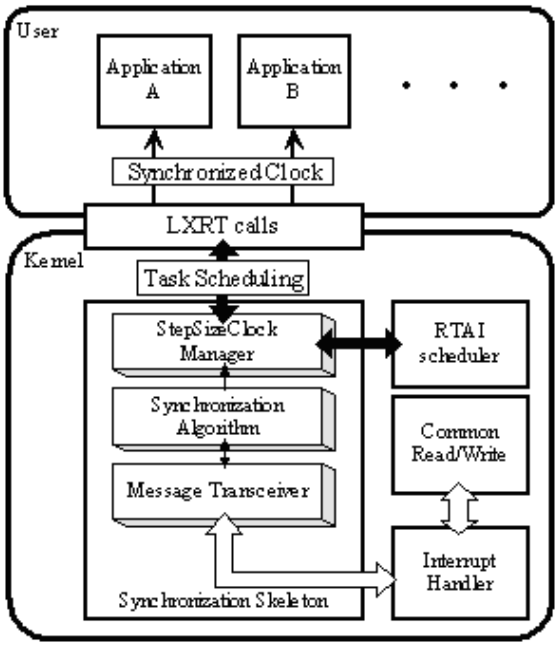


FIGURE 2: *The Structure of Synchronization Skeleton*

Located in the middle of the skeleton, the *Synchronization Algorithm* communicates with equivalents in other nodes via the *Message Transceiver*. The *Message Transceiver* is the interface of the skeleton to the communication layer. By connecting the transceiver with the communication layer, the corresponding communication functionalities in the operating system are exported to the skeleton. Because of this transceiver, the skeleton is able to work independently of the underlying communication technology. Clock synchronization in the skeleton requires deterministic message transmission. This feature, however, depends on the communication layer outside the skeleton. Thus for working with the skeleton, a real-time capable communication technology is needed.

The *Interrupt Handler*, which belongs to the communication layer of the operating system, acts as a message dispatcher. It decides whether a message is a synchronization message. For receiving / transmitting a synchronization message, the *Interrupt Handler* cooperates with the *Message Transceiver* directly. In this situation the *Common Read/Write* block of the operating system will not notice the message transmission. Otherwise the transmission is handled by the *Interrupt Handler* and the *Common Read/Write* blocks.

Within the *Synchronization Algorithm*, algorithm dependent calculation and signal processing is done. From the *Synchronization Algorithm*, a globally synchronized clock called *Step Size Clock* is pro-

vided to the *StepSizeClock Manager*, which is a tiny RTAI periodic task that resides in the core of the *Synchronization Skeleton*. The *StepSizeClock Manager* takes the clock and keeps itself synchronized with counterparts in the other nodes. In the following discussion, the notation $SSC(t)$ is adopted to denote the value of a *Step Size Clock* in the *Synchronization Skeleton* at real time t.

The *StepSizeClock Manager* helps to schedule all the user space applications according to their specific periods. In such a way, applications running above the *StepSizeClock Manager* are assured to be synchronized in the scope of the entire system. From the viewpoint of the *RTAI scheduler*, the *StepSizeClock Manager* is a proxy deciding at what time and which RTAI task is to be scheduled.

In user space, the application sees only a synchronized clock from the provided LXRT calls, which is called *rt_sync_task_wait_period*. Its name indicates the functionality: when an RTAI periodic task finishes its work within a cycle, invocation of the LXRT call suspends the task and the *StepSizeClock Manager* wakes it up when its period expires. In this way, a periodic task based on the synchronization scheme is realized.

2.3 The clock flow

The complete *Synchronization Skeleton* is designed to maintain a global clock in the system. The skeleton on each distributed node reads the local clock based on RTAI, calculates and provides the synchronized clock to the upper level applications. This can be illustrated by the figure below.

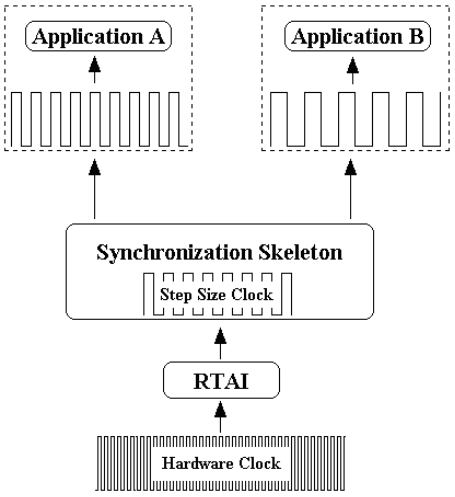


FIGURE 3: *The Clock Flow*

The hardware clock is read by RTAI, and the logical clock, *Step Size Clock*, is generated by the *Synchronization Skeleton* according to the configuration of the *StepSizeClock Manager*.

The *Step Size Clock* is the base clock in the skeleton and its period is configurable at initialization of the skeleton. Derived from the clock, different clocks can be generated and provided to the upper level applications. In this way, the skeleton can be regarded as a clock platform that offers clocks at different frequencies according to specific requirements in the user applications. Since the *Step Size Clock* is synchronized within the scope of the system, all the user applications are kept synchronized with each other in different nodes.

2.4 Accuracy of the skeleton

With the *Synchronization Skeleton*, all the nodes in a distributed system can be synchronized within a fixed clock deviation boundary. The accuracy of the clock synchronization is analyzed and determined.

Software synchronization algorithms are always discrete-update algorithms. Clock deviation between two successive resynchronizations is inevitable. Derived from the theory in [2], the maximum clock deviation, d, during period, Δt, between one node and the global clock can be expressed by

$$d = \rho \Delta t \tag{1}$$

where ρ is the clock drift rate of the node. The maximum clock deviation between any two nodes in the system is then doubled as

$$d_{theory} = 2d = 2\rho \Delta t \tag{2}$$

where d_{theory} is the theoretical clock deviation. In the real world the deviation is influenced by the environment factors. Three major factors are emphasized in the following discussion, namely the timer granularity, the deviation of the clock drift rate and the transmission time deviation.

1 Timer granularity

Each instance of *Synchronization Skeleton* maintains its own *Step Size Clock*. Since the *Step Size Clock* is a logical clock that increases discretely, its granularity has great impact on the synchronization accuracy. This effect can be presented in the following equation, taken from [6].

$$\frac{t - t'}{1 + \rho_i} - G \leq C_i(t) - C_i(t') \leq \frac{t - t'}{1 - \rho_i} + G \tag{3}$$

where t and t' are two different real time instances, ρ_i is the clock drift rate of node i in the system, $C_i(t)$ is the reading of the local physical clock of node i at real time t, and G is the clock deviation caused by the granularity.

Let the granularity of the *Step Size Clock* be g , the time when the *Step Size Clock* ticks be t_0. For any time $t \in [t_0, t_0 + g)$, the equation $SSC(t) = SSC(t_0)$ always holds because of the resolution of the *Step Size Clock*. The minimum of g here is system dependent and determines the synchronization accuracy of the skeleton.

When the effect of the clock granularity is considered, clock deviation between a local clock and the global clock, d_g, is determined by the larger value of $\rho\Delta t$ and the clock granularity g. That is

$$d_g = MAX\,[g, \rho\Delta t] \tag{4}$$

g can be set at the initialization of the skeleton, and Δt is also configurable with the knowledge of ρ. It is rational to set g not larger than $\rho\Delta t$ and make it possible to detect and limit the clock deviation by adjusting the resynchronization interval Δt. On the other hand, the granularity of *Step Size Clock* is not supposed to be very small, otherwise the system will crash due to the heavy load. An optimal solution is achieved when

$$g = \rho\Delta t \tag{5}$$

which gives sufficient time resolution while more CPU power is left to other applications. Actually equation (4) becomes (1) when (5) holds, and the influence of clock granularity is gone. In this case, the G in equation (3) can be regarded as zero and equation (2) holds.

2 Deviation of the clock drift rate

By measurement, the maximum clock drift rate in a system can be obtained. However, in reality the actual hardware clock drift rate is not always the maximum due to the changing environment. In this case extra clock deviation is introduced and should be taken into consideration when estimating the accuracy. Let ρ_{max} and ρ be the maximum drift rate of hardware clock obtained from measurement and the actual value, respectively. By equation (2) the minimum resynchronization interval is given as

$$\Delta t_{min} = \frac{d}{\rho_{max}} \tag{6}$$

With the knowledge of Δt_{min} and ρ, the granularity of *Step Size Clock* is then as

$$g = \rho_{max}\Delta t_{min} \tag{7}$$

The assumption is made that the actual clock drift rate falls in the interval of $0.5\rho_{max} \leq \rho \leq \rho_{max}$ in the following discussion. The assumption is plausible as the drift rate will not change sharply given that the outside environment remains relatively stable. Derived from equations (1) and (7), $d = \rho\Delta t_{min} < g$.

As is shown in Figure 4, given that at time t_0, right after one resynchronization, a local clock is exactly the same with the global one. Before the next resynchronization the deviation of local clock is d, which is less than the period that can be detected. Due to the clock resolution, at $t_0 + \Delta t$ no resynchronization is done. This clock deviation is found at $t_0 + 2\Delta t$, when it has grown to $2d$, which is larger than g. The resynchronization will be activated. However, the boundary of the actual clock deviation has already been violated.

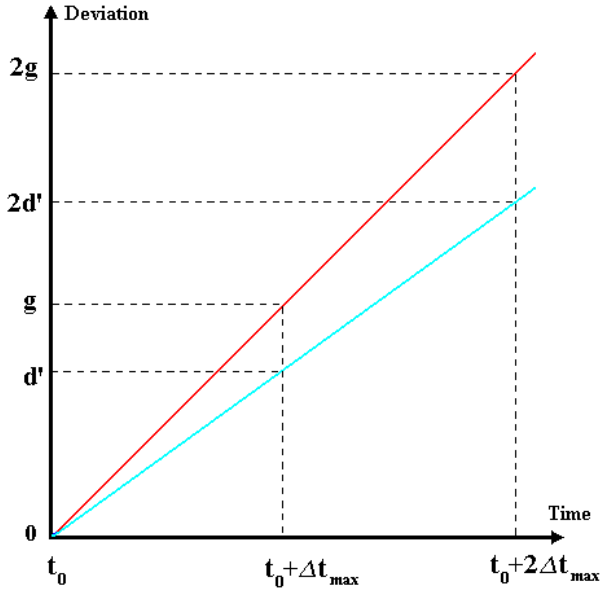


FIGURE 4: *Situation when MAX. clock deviation is violated*

Due to this effect, the clock deviation doubles again. As a conclusion, the total clock deviation between a local and global clock in the *Synchronization Skeleton* becomes

$$d' = 2d = 2\rho\Delta t \tag{8}$$

Then the clock deviation between any two nodes is

$$2d' = 4\rho\Delta t \tag{9}$$

3 Transmission time deviation

Another sort of deviation is introduced in the *Synchronization Skeleton*, due to deviation of synchronization message transmission. This part of deviation depends on specific communication technology and is expressed as d_t.

Taking all the aspects above into account, the clock deviation between any two nodes in the system is given as

$$D = 2d' + 2d_t = 4\rho\Delta t + 2d_t = 4g + 2d_t \tag{10}$$

for point-to-point synchronization message transmission, or

$$D = 2d' + d_t = 4\rho\Delta t + d_t = 4g + d_t \qquad (11)$$

for broadcasting the synchronization message.

With equation (10) and (11), the synchronization tightness of a system can be realized by setting g and Δt properly in the skeleton.

However, there are some exceptions when the actual clock deviation exceeds the maximum value. In the following situations, the clock deviation boundary cannot be guaranteed.

1 Large clock drift rate

The previous discussion is based on the assumption that by measurement the maximum clock drift rate in the system is obtained. In reality, this assumption may not hold. The actual clock drift rate would become higher than expected when the working environment changes considerably and is different with that in the measurement. For instance, when a distributed node starts up, the temperature of most electrical components in the node would be much lower than that in the normal working period. This may introduce a relatively higher clock drift rate. In case the real clock drift rate is larger than the value set in the skeleton, the synchronization tightness cannot be guaranteed by the skeleton. The corresponding strategy, however, can be defined in the synchronization algorithm to deal with this situation.

2 Heavy task scheduling

When multiple tasks are running in parallel on the same processor, and at a certain point of time more than one task need to be scheduled, the clock deviation boundary would be violated. This is due to the fact that on a single processor system, the RTAI scheduler together with the *StepSizeClock Manager* can only schedule one task at a time. In reality it is impossible to wake up more than one task in one node at the same time, and therefore, some tasks will unintentionally be delayed. In the worst case, when all the synchronized tasks are working at the same frequency and starting synchronously, which means all the tasks are to be waken up at the same time, there will be only one task be scheduled on time. The rest of them will be delayed. This delay time, therefore, should be taken into the total clock deviation, which is given as below:

$$D + (n-1)t_s \qquad (12)$$

where t_s is the scheduling time for one task and n is the maximum number of tasks, which need scheduling at the same time. However, it does not mean the system loses its clock synchronization. Only the clock deviation boundary is enlarged.

To ease the workload of the scheduler, one can either try not to have too many tasks running on the same node, or set a longer period for the *StepSize-Clock Manager*, which will decrease the synchronization tightness though. Another solution is to shift the execution time of the tasks in parallel a bit, such that the scheduling of these tasks will not happen at the same time.

3 Experimental Results

To test and demonstrate the performance of the *Synchronization Skeleton*, two synchronization algorithms have been implemented in the skeleton. The tests of the skeleton together with the algorithms have been made in several PC104 Embedded PCs, each of which has a VIA Eden ESP processor running at 600MHz. The operating system used was Linux 2.6 with RTAI 3.3. A real-time CAN driver was used for transmitting synchronization messages. Multiple user space processes were running on each PC and sending pulses at the defined frequencies via the parallel port, simulating the periodic behavior of distributed controllers. By observing and comparing the output pulses from the distributed nodes in a scope, the clock deviation in the system was obtained.

By measurement, the maximum clock drift rate in the system has been obtained as about 3 ppm (parts per million), and the deviation of CAN transmission in hardware was observed as 3 μs. Working in kernel space of the RTAI domain, software deviation in the skeleton can be ignored. To test the best synchronization tightness that can be achieved in the system, the granularity of *Step Size Clock* was set to 20 μs. Any value smaller will crash the system. By equation (10) and (11), the maximum clock deviation in the system is 83 μs or 86 μs, depending on different algorithms.

The skeleton defines the best synchronization tightness of the built-in algorithms. Two synchronization algorithms have been implemented in the *Synchronization Skeleton* to demonstrate integration of algorithms in the skeleton.

3.1 The centralized algorithm

The first one is a centralized algorithm, which is a derivative of the Cristrian's Algorithm [2]. A synchronizer that holds the global clock of the system maintains the clock synchronization. By broadcasting the clock every resynchronization interval, the

synchronizer helps all the common nodes get synchronized in the system. Since the granularity has been set, the maximum clock deviation and the resynchronization interval is determined by equation (11).

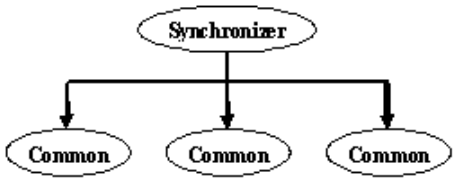


FIGURE 5: *The centralized algorithm*

3.2 The distributed algorithm

The other is a distributed algorithm, which is a simplified version of that described in [5].

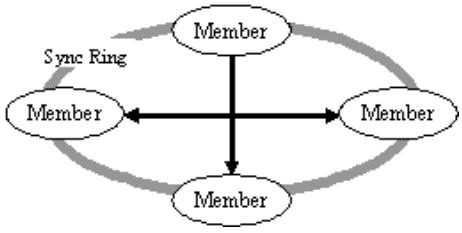


FIGURE 6: *The distributed algorithm*

In the distributed algorithm, each node in the system has the same opportunity to initiate a resynchronization. All nodes in the system form a *Sync Ring* in the system. Every node is a *Member* of the ring. With the knowledge of the node number and its position in the *Sync Ring*, a *Member* counts the synchronization messages it has received, by which the *Member* decides the time to broadcast the local *Step Size Clocks* as the global clock. This algorithm provides a more balanced system configuration since the implementation in all nodes is the same.

3.3 The results

The measurements have been performed using the setup with the two algorithms separately.

On each distributed computer there were 3 processes running in parallel and sending pulses to the parallel port at 1KHz, 500Hz and 250Hz respectively. Two groups of measurements have been done, with and without synchronization support. Also the two synchronization algorithms presented were tested. Based on the skeleton, both algorithms provide clock synchronization with the same accuracy. The measurement results are listed in Table 1.

	Synchronization OFF	Synchronization ON
Clock Deviation in 10 min	1.29ms	72 μs

TABLE 1: *Clock Deviation Measurement*

Within 10 min the clock deviation between the distributed nodes was 1.29 ms without synchronization support. This is unacceptable for most distributed applications. In the other measurement, the clock deviation was kept within 72 μs, which is in the clock deviation boundary and will not grow with time. In the measurement it was also observed that the skeleton running on a single processor needs about 15 μs to schedule one task. Therefore, except for the first task to be scheduled, the rest of the tasks running on the same node were delayed a bit.

The multi-node measurement setup is also used as a distributed control setup to control some mechanical systems for experimental purposes. Most controllers running in the setup are working at a frequency lower than 1KHz. The clock drift rate that is achieved with the skeleton is less than 10% of the controllers' working clock cycle. For such an application, the synchronization tightness is satisfactory.

4 Conclusions

This paper presented a thin layer called *Synchronization Skeleton*. The skeleton is inserted in a distributed system and provides unified and extendible interfaces to the clock synchronization algorithms, communication technologies and the user applications. In this way the realization of a new synchronization algorithm becomes much easier, and replacement of the underlying communication technology requires only modification of the interfaces. Besides, changing of either the algorithm or the communication technology will not affect the implementation of the upper level user applications.

The performance of the skeleton has been tested with 2 different synchronization algorithms in a multi-node distributed system based on a real time CAN driver. The test results showed that the *Synchronization Skeleton* together with the synchronization algorithms provides reliable clock synchronization support to the distributed control system.

Future work will be, but not limited to, the following:

Integration of more communication technologies and synchronization algorithms into the skeleton is attractive. This will bring the skeleton to a broader field of applications.

Integration of the skeleton into RTAI is appealing. The feature of clock synchronization is essential in distributed systems. Providing this feature directly from the operating system increases usability, and probably enhances performance.

Details about the implementation of the *Synchronization Skeleton* can be found in [7].

References

[1] S.Tanenbaum, 2002, *Distributed Systems C Principles and Paradigms*, Prentice Hall, 0130888931.

[2] F. Cristian, 1989, *Probabilistic Clock Synchronization*, Distributed Computing, vol. 4, pp. 146-158.

[3] R. Gusella and S. Zatti, 1989, *The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD*, IEEE Transaction Software Engineering, vol. 15, pp. 847-853.

[4] T. K. Srikanth and T. Sam, 1987, *Optimal clock synchronization*, Journal of the ACM, vol. 34, pp. 626-645.

[5] A. Daliot, D. Dolev, and H. Parnas, 2003, *Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks*, Schools of Engineering and Computer Science,The Hebrew University of Jerusalem, Technical Report TR2003-1.

[6] F. Cristian, H. Aghili, and R. Strong, 1986, Clock Synchronization in the Presence of Omission and Performance Failures, and Processor Joins, 16th International Conference on Fault-Tolerant Computing, vol. 23, pp. 218-223.

[7] Y.Huang, 2006, Clock Synchronization using Real-Time CAN, Control Laboratory, University of Twente, Enschede MSc. Report, 015CE2005.

Safe and Cooperative Coexistence of a SoftPLC and Linux

Robert Kaiser, Stephan Wagner and Alexander Zuepke
SYSGO AG
Am Pfaffenstein 14, D-55270 Klein-Winternheim, Germany
{rob,swa,azu}@sysgo.com

Abstract

Combining Linux and a softPLC in a single system stands to reason: Linux offers many facilities that modern PLCs are expected to support. However, existing Linux-based softPLC implementations so far have always placed the PLC "on top" of the Linux kernel, so its functional reliability depends on the correctness of the kernel. Due to its size, the Linux kernel can not be exhaustively validated or even proven correct. This has hampered applicability of the concept to safety-critical PLC systems. The approach described in this paper puts Linux and a softPLC "side by side" on top of a small microkernel, thus the two subsystems can coexist safely without being forced to depend on each other. In this way, the trusted code base of the PLC is reduced by several orders of magnitude, thereby enabling its certification according to applicable standards for safety-critical systems.

1 Introduction

Programmable logic controllers (PLCs) are the backbone of today's automation industry. Invented in the late 1960's [16], they have gradually evolved from a replacement for hard-wired relay circuitry to complex systems which are nowadays often expected to offer features such as a graphical user interface or internet connectivity. These features (and a lot more) are readily available from a general purpose operating system such as Linux, so the idea of combining Linux and a software-based PLC implementation (*softPLC*) in a single machine stands to reason.

However, PLCs are also frequently used in safety-critical applications. A malfunction in such a system could cause significant damage or even death or injury of human beings, so these systems have be to far more reliable than the average workstation computer. This reliability has to be proven prior to deployment by means of testing or even formal verification. While the standardised PLC programming languages ([10]) lend themselves to such exhaustive testing or validation, the same can not be said about the platform executing the code (i.e. the softPLC). Thoroughly testing such a platform is a costly task and the cost increases proportionally to the amount of code that needs to be examined. This "trusted code base" of the softPLC comprises all code that has the potential of affecting its correct function. If Linux is used as a base to support a softPLC, this amount of code increases by roughly one million lines (the code size of the Linux kernel), making a thorough validation or verification quite infeasible.

This paper introduces a way to enable safe and cooperative coexistence of a softPLC and Linux in the same machine. Unlike previous approaches, Linux is *not* used as a base for the PLC, rather, both Linux and a PLC exist side by side, based on a minimal amount of commonly trusted code. Our approach avoids dependencies between the two components that would otherwise require one of them to trust in the other's well-behaving. We first look at some existing Linux-based softPLC implementations, analysing their dependency paths. Subsequently, we present our approach (which is based on a microkernel), and finally, we show some initial results of an ongoing project which applies the presented approach. This project is aimed at integrating Linux and the well-known softPLC "CoDeSys" into a single system.

2 Existing Linux-based softPLCs

Technically, a softPLC is a program that executes specialised code, much like a Java virtual machine (JVM), however, unlike a JVM, a PLC needs to execute its code in a timely fashion. Therefore, any

softPLC needs its underlying operating system to guarantee real-time execution. Standard Linux can do this only to a limited extent, so, Linux-based softPLCs up to now have either used one of the various real-time kernel extensions, to provide "proper" real-time support for the PLC, or they have settled for the Linux kernel's limited real-time performance.

2.1 softPLCs based on Linux Real-Time Kernel Extensions

Real-time extensions to the Linux kernel such as RTAI [15] or RTLinux [2], work by integrating a dedicated real-time programming interface into the Linux kernel. There is only a formal separation between this interface and Linux in that real-time code is not *supposed* to invoke functions from the Linux kernel, but there is no way to *enforce* this rule: technically, any real-time application is able to invoke any Linux kernel function (which, in most cases, will lead to undefined behaviour). All real-time activities, and thus, in our case, the softPLC, run as privileged code in the same address space as the Linux kernel. Both Linux and the softPLC have uncontrolled access to the same code, data and I/O ports: they are not spatially separated.

The real-time scheduler runs Linux as its lowest priority process, thus, Linux only gets to execute when none of the real-time processes are ready. This means that for access to computation time, Linux is at the softPLC's mercy: if the PLC never blocks, Linux will not execute. So, in addition to the spatial dependency, Linux also depends on the softPLC temporally (but not vice versa): they are not temporally separated.

While the Linux kernel is able to affect the PLC, Linux application programs are not, unless they run with root privileges. So, assuming we can rule out any security holes (e.g. root exploits) from the Linux kernel, the softPLC need not trust Linux applications. Therefore, the trusted code base in this configuration comprises the Linux kernel and the softPLC.

2.2 User Space softPLCs

The plain, unmodified Linux kernel already comes with some facilities that allow for limited real-time functionality. If a softPLC application can accept deviations of timing in the range of a few milliseconds, it can be implemented as a Linux user process[18] [1]. Also, some of the above mentioned Linux real-time kernel extensions offer optional support to make the real-time interface accessible to user-space programs (e.g. LXRT [5]). This could also be used to implement a softPLC in user space, without having to make concessions regarding real-time performance.

With both approaches, the Linux kernel and the softPLC do not share the same address space and the PLC can not easily compromise the Linux kernel. Conversely, however, the PLC does depend on Linux making memory and I/O resources available to it as required, so, the two components are still not spatially separated.

Regarding temporal separation, if the PLC exists as a standard Linux user process, it needs to receive its allocation of computing time from the Linux kernel, while, if a real-time extension is used from user space, Linux again only gets to execute when all real-time processes are blocked. So, with both methods, there is no temporal separation: either the softPLC is at the Linux kernel's mercy, or vice versa.

Again, the total amount of trusted code, i.e. code which is needed to establish the softPLC's functionality, includes both the Linux kernel and the softPLC's code.

3 Approach: Separation of Resources

In both approaches mentioned above, the Linux kernel is in control of the spatial and -except for the LXRT case- also the temporal resources of the system. The softPLC crucially depends on this kernel: it must assume its entire correctness. Given the kernel's sheer size, this assumption is not very likely to be met, and auditing the trusted code base according to standards such as [9] is not feasible. Hence, such systems can not be used in safety-critical applications.

However, looking at a typical softPLC implementation, it turns out that only a very small part of the kernel's functionality is actually required to support it: All it really needs is a basic run-time environment, i.e. access to memory, I/O and processing time. Many of the advanced features that the Linux kernel has to offer are not required by the PLC. Nevertheless, since these features are implemented in the kernel, the corresponding code has to be trusted.

In order to reduce the amount of trusted code, our approach only implements those functions at the kernel level that are either technically impossible to do otherwise (i.e. require privileged instructions) or

[1] However, it should be noted that the timing deviations cited in [18] have been obtained by experimenting. Such experiments can only yield an *observed* worst case value, not the worst *conceivable* value. A safety-critical system must not assume that this value can never be exceeded.

that are needed to establish a secure runtime environment for user level programs. All other functionality that classical monolithic kernels like Linux tend to do in privileged mode can just as well be done by user level programs. A kernel that is designed according to these paradigms is usually referred to as a "microkernel"[12]. Such a microkernel provides only basic mechanisms which allow to divide a system's temporal and spatial resources into individual subsets. These subsets can also be regarded as virtual machines[2], each of which can host a complete operating system such as Linux along with all of its applications. Unlike RTAI or RTLinux, this approach in principle allows for any number of virtual machines to exist in a single machine, i.e. we can have more than one softPLC running independently in one machine. Any code running inside a virtual machine must run in user mode. Hence, an operating system will in most cases need some adaption. User-space applications, however, need not even be aware of the difference. Since it is confined to its own set of resources, an operating system running inside a virtual machine can not affect another operating system running in another virtual machine. The system is divided into *partitions*, which are completely independent of each other. Applying this to the case of a softPLC in combination with Linux, we can assign each of them to a different partition and thus be sure that they are independent of each other: the softPLC is no longer required to trust Linux and vice versa (See figure 1). Both subsystems coexist side by side (as opposed to one on top of the other). Both share a common trusted code base consisting of the microkernel itself (the only software component that runs in privileged mode) and a "System Software" Layer, a software module which runs in user mode, on top of the microkernel. The latter implements the policies for managing the partitions, based on the mechanisms provided by the microkernel.

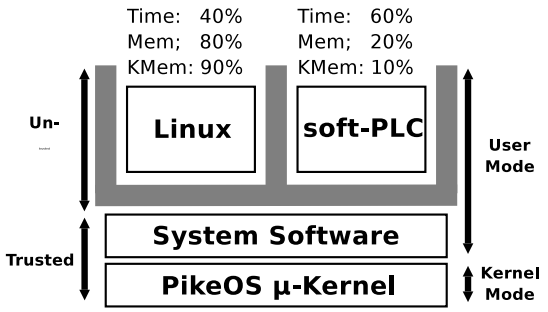


FIGURE 1: *Partitioned system*

In the following subsections, we will describe the separation of resources in more detail, looking at spatial and temporal resources in turn.

3.1 Spatial Separation

To be able to run a guest operating system in a partition, a microkernel must provide access to portions of the system's hardware, i.e. memory, memory-mapped or (on x86) port-mapped I/O registers, and interrupts. It must also provide mechanisms for the guest operating system to distribute and revoke access to these resources by applications, running on top of it. This has to be done as flexibly as possible to allow all kinds of operating systems as guests and to keep the porting complexity small [12].

To ensure independence between guest systems, these resources must be distributed by a commonly trusted party (the kernel). To do so, the microkernel assigns to each partition a set of virtual address spaces, which act as containers for resources. With these methods, the microkernel is able to guarantee that no partition can interfere with another.

But this only covers user space resources. The microkernel itself needs memory to manage pagetables, stacks, ready-, and wait queues, etc. These resources also need to be separated, or else one partition could mount denial of service attacks by making the kernel consume huge amounts of memory, e.g. by mapping one page to all available virtual addresses within its address space [14].

Spatial separation in our approach is geared to the ARINC 653 standard [1]: the distribution of resources is made statically according to a fixed configuration. The standard requires the most severe restrictions to the system setup. The static configuration of resources is implemented by the "System Software Layer". In addition to the seperation facilities, the System Software Layer also offers communication services like shared memories and notification mechanisms between partitions. This allows for secure communication between guest systems accross partition boundaries.

3.2 Temporal Separation

Most microkernels were not initially designed with the goal of real-time execution in mind. Consequently, many of them provide for spatial resource separation as described in the previous subsection. However, only few also provide the necessary facilities to enable deterministic distribution of temporal resources (i.e. processing time).

The goal of a virtualisation environment is to give every operating system executing within a par-

[2]This is also referred to as *para*virtualisation in [3]. In fact, we consider virtual machine monitors such as Xen to be specialised implementations of the microkernel concept.

tition the illusion of having a constant proportional share (i.e. a percentage) of the overall processing time available for its own use. This would imply that the execution time assigned to individual partitions increases in a linear fashion as shown by the dash-dotted lines in figure 2. In practice, however, the processor(s) can only be time-multiplexed across the partitions, i.e. every partition has a time slot during which it is active. The idealised linear progression of execution time is approximated by a ramp-like function as shown by the solid lines in figure 2.

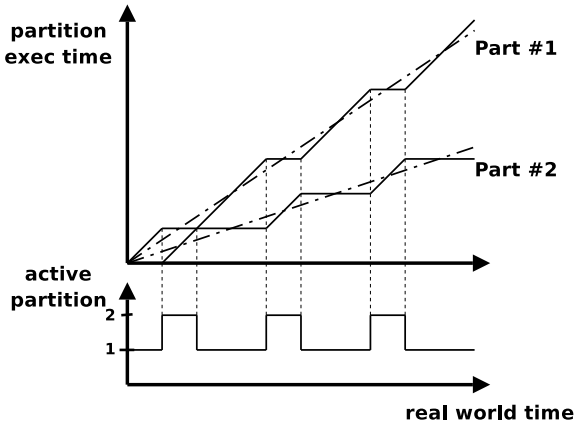


FIGURE 2: *Per-partition execution time vs. real world time*

The quality of this approximation improves as the granularity of time assignments, i.e. the absolute duration of time slots is made shorter, however, the resulting increase in switch frequency leads to excessive overhead.

A softPLC is a classical example of a time-triggered system: It has to be invoked periodically at fixed points in time. If these invocations are not synchronised against the partition switches, the softPLC will experience unpredictable delays. These delays can hit the softPLC anywhere during its cycle, i.e. they can delay the PLC from becoming active (thus increasing jitter), or they can delay its computation, thus increasing any deadline the PLC may have to maintain. Obviously, it would be desirable to switch between partitions as quick as possible in order to keep these delays small, however, we have already seen that this is limited by the switching overhead quickly becoming excessive. Another, possibly better way is to synchronise the partition switches against the softPLC's cycle: The virtual machine hosting the softPLC must be activated just in time for the softPLC to become active and it must receive an amount of time sufficient for the PLC to complete its job. In this way, a softPLC hosted by a virtual machine can deliver the same real-time characteristics as a non-virtualised one. Since a PLC is a strictly time-triggered system, the points in time when it is to be activated are known in advance, i.e. they are defined as a function of time only. From this follows that, if partition switching should be syncronised, the partition switches must also be defined strictly as a function of time only. Most existing virtualisation environments fail at this point: They allow their individual virtual machines to suspend themselves and, once a VM does so, they immediately switch to the next one. This method, while advantageous from a processor utilisation point of view, makes it impossible to predict the partition switch times and so, a time triggered system (e.g. a PLC) running in a virtual machine can not be syncronised against the partition switches.

The time partitioning concept of our approach is an enhanced variant of ARINC 653[1], a standard which is popular in avionic systems: The ARINC 653 standard assigns periodically repeated time slices of fixed duration to virtual machines. Every VM can tell in advance when and for how long it will be active, so it can synchronise its own scheduling to the scheduling of virtual machines. However, a disadvantage of the ARINC 653 method is that all VMs must consume all of their allocated time: there is no way to suspend VMs when they are idle, so VMs have no choice but to "burn away" any time that they can not use for themselves. The amount of time allocated to a VM is determined by worst case assumptions: A real-time system running in a virtual machine must be able to complete its task within this time frame under all conceivable conditions. However, worst case scenarios, though possible, tend to occur only rarely and so most real-time virtual machines tend to have far more time available than they actually use in the average case. Therefore, systems using the ARINC 653 approach tend to exhibit a rather poor processor utilisiation. If all VMs would host real-time systems only, this waste of resources would be inevitable, however, in our system, we also have Linux, a non-real-time system which could make good use of any excess computational resources. Therefore, in our approach, we combine the strictly time-driven ARINC 653 scheduler with a priority-driven scheduler: any amount of time that is assigned to, but not used by the softPLC, is dynamically re-assigned to Linux. This is achieved by placing Linux into a "background" time partition which is always in a runnable state and assigning a priority lower than the softPLC's to it.

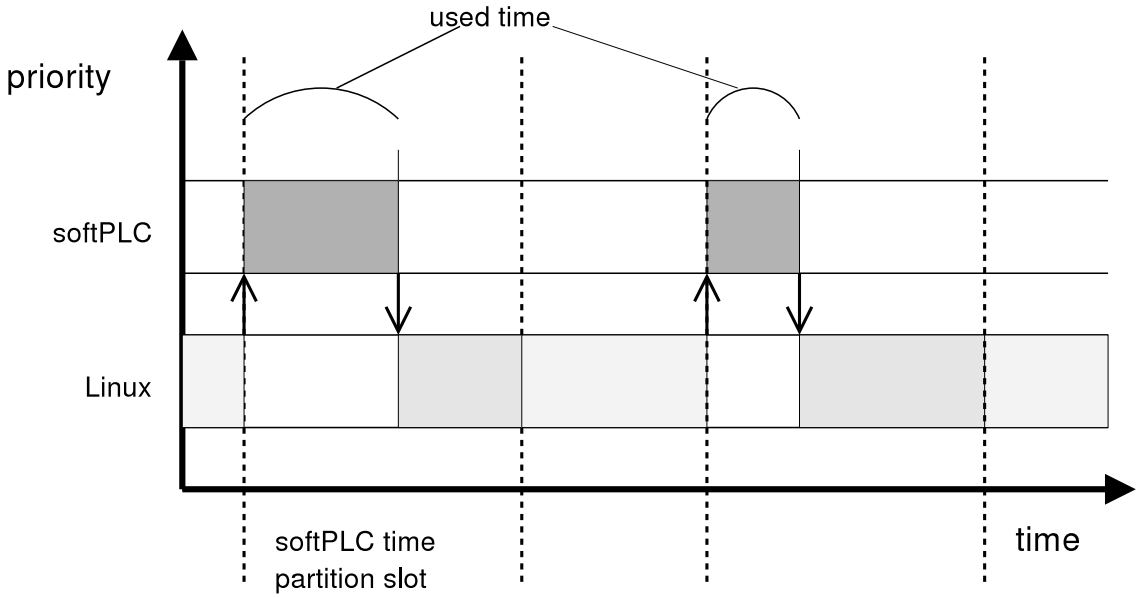


FIGURE 3: *A high priority softPLC in an own time partition and Linux in the background.*

The amount of time that the softPLC can consume is limited, so, even if the PLC made an attempt to monopolise the processor at high priority, it could still not starve Linux (as would be the case with RTAI or RTLinux). The softPLC always gets its guaranteed share of resources at predetermined points in time and also Linux has a guaranteed minimum amount of CPU time. Linux cannot disturb the PLC during its execution and the PLC can not detain Linux's share of CPU time: The softPLC as well as Linux need not trust each other any more.

4 Practical Implementation

In this section we describe some implementation details of the microkernel with a special focus of partitioning, CoDeSys, and the Linux kernel as guest operating systems.

4.1 PikeOS Microkernel

In the 1990s, Sysgo have been developing their own microkernel. Initially modeled after the L4 version 2.0 microkernel API as specified by Jochen Liedtke in [13], the kernel evolved from the generic microkernel approach to a specialized kernel for embedded systems with a focus on real-time and partitioning. The kernel is written almost entirely in C to facilitate porting. Currently supported architectures are: x86, PowerPC, MIPS, and ARM.

The general system design today is a static configuration approach to fit the microkernel concepts to practical embedded systems, which usually do not need complex dependencies and capabilities. The complexity depth was reduced to typical use-cases, like a Linux environment running in a partition on top of a System Software Layer, next to other services with a similar grade of requirements.

4.2 Basic Concepts

Like L4, the PikeOS Microkernel provides the concepts of tasks, threads and inter process communication (IPC). Unlike L4, it extends the task concept with the notion of "abilities", and it offers partitioning of resources as well as computation time.

A thread is the basic entity of execution. It has a unique identifier (UID) and is bound to a task. It is implemented by a register context which is scheduled at a given priority and time partition. Optionally, it can register a handler to receive exception messages. A thread can also register itself as interrupt handler, thus facilitating interrupt handling at user level.

Each thread is assigned to a task. A task is a container for threads, which all share the same virtual address space. Furthermore, it defines a set of "abilities", which restrict the allowed system calls of its threads. Tasks are organised in a hierarchical tree. All tasks have a parent task and may have child tasks. Unlike the clans and chiefs concept [12], the

microkernel manages task based communication and interrupt handling rights. These rights can be inherited from their direct parent by child tasks, where the parent is able to further restict them. Thus, a child can at most have the same rights as its parent, but never more. A task also restricts the priorities of its threads to an upper bound, the so-called "maximum controlled priority" (MCP). Again, a child's MCP can at most be the same as its parent's, but never higher.

Tasks are also grouped in "resource partitions": Tasks belonging to the same resource partition share a common kernel resource pool. Whenever a kernel call is made which causes the kernel to allocate a resource, that resource is taken from the caller's kernel resource pool. Thus, tasks belonging to the same resource partition can mount denial-of-service attacks against each other, but tasks with different resource partitions can not.

The kernel supports inter process communication (IPC) as standard communication mechanism. An IPC operation is synchronous and allows to transfer an arbitrary set of data or memory mappings between threads in different tasks.

Further extensions to the L4 concepts are an asynchronous notification mechanism and a special map system call. The notification mechanism has been added to simplify notification in a time partitioned environment. The map system call has been added to simplify process creation. This allows to establish all mappings before the first thread of a new task is activated.

4.3 System Software Layer

The System Software Layer distributes all available resources of the system according to a static configuration. Dynamical resource allocation is explicitly not possible.

In addition to this resource distribution , the system software layer also provides facilities for communication across partition boundaries. Probably the most basic of these facilities is shared memory: It is possible to define shared memory segments which are accessible to multiple guests in different partitions. To all communicating parties, such a segment appears as an external resource, i.e. neither of them "owns" it. This is necessary in order to maintain secure isolation between the parties. In addition to shared memory, the system software also provides two slightly more elaborate communication mechanisms: There are *queuing ports*, unidirectional buffered, message based communication channels similar to pipes, and *sampling ports*, unbuffered mechanisms roughly comparable to mailboxes. Both concepts were modeled after the ARINC 653 standard [1]. These ports can be accessed in a blocking and a non blocking mode. Again, resources used by these communication facilities are owned by the system software, a trusted component, so any operating system hosted in a partition can rely on them.

Also, the System Software Layer provides mechanisms for health monitoring and partition control. This makes it possible to define how to react on exceptions and errors, depending on the current system, partition, or process state.

The system software layer also provides a rudimentary file system and a a generic device driver infrastructure.

4.4 Porting CoDeSys

There already exist a large variety of different softPLC implementations, both as commercial products or as open source projects. Therefore, rather than re-invent yet another softPLC implementation which would surely have been an inferior to these mature products, we decided to port a readily available softPLC implementation to our virtual machine environment. We chose "CoDeSys" by 3s Smart Software Solutions because of its availability in source code form, its portability and –last but not least– because it is well accepted in the market.

PLC programming is typically done with an offline tool in one of the programming languages standardised in [10]. In the case of CoDeSys, the tool runs on a standard PC and connects to the softPLC via Ethernet and TCP/IP. The tool generates IEC compliant code which is downloaded to the softPLC for execution. The tool also supports remote debugging of PLC code via the same network link.

Internals

The softPLC itself runs several threads:

- The *IEC threads* execute the system's IEC function blocks. They can be time-triggered, in which case they are executed periodically at predefined points in time, or they can be "event-triggered", in which case their execution is controlled by an "IEC event" (basically a boolean variable). It should be noted that even "event triggered" IEC threads are actually time-triggered from the microkernel's point of view: Technically, they poll their IEC event periodically and, depending on the event's current value, they either execute IEC code or wait for the next cycle.

- The *control thread* triggers the PLC's IEC threads and it supervises them: If any of them

exceeds their time allocation, it stops the offending thread and signals a fault condition.

- The *communication threads* are responsible for communication via TCP/IP or –optionally– serial link. Unlike the other threads mentioned so far, these threads are event-triggered as they are activated by external events. Also unlike the other threads, these threads do not have to fulfill hard timing requirements.

The CoDeSys PLC has previously been adapted to a number of different real-time operating systems. It expects its run-time environment to support basic real-time, multithreading functionality. To port CoDeSys to our environment, we used the already existing implementation of the POSIX PSE51 ("Minimal Realtime System Profile") [11] API to provide this functionality. CoDeSys also needs TCP/IP sockets for communication. These were provided by a port of "lwIP", the "light weight IP" stack [7].

Hardware interfacing

In typical softPLC applications, the PLC interacts with the outside world via external I/O components which are connected to it either directly ("local I/O") or via various different fieldbuses. Thus, it is crucial for our PLC implementation to support a wide variety of fieldbuses. In the current prototype version, local I/O and Profibus are already supported and ongoing development aims at adding support for CAN, EtherCAT and Profinet. Unlike most other virtual machine environments, our system is able to partition the hardware's I/O resources, too, i.e. for every partition, it is possible to individually select the memory-mapped or port-mapped I/O registers that should accessible to it. In this way, the available selection of CoDeSys drivers for different fieldbus modules can be re-used in our environment with little to no change.

Communication with Linux

Communication between the softPLC and Linux is facilitated by the communication mechanisms provided by the system software layer. Queuing ports and sampling ports can be accessed from both sides like normal I/O devices. Either side can use this interface in ways to eliminate unwanted dependencies beween the two sides, for instance, the PLC can avoid being blocked by an attempt to send messages in case the Linux side does not read its end of a sampling port fast enough.

4.5 Implementing Linux as a Virtual Machine

Porting Linux to a virtual machine is not a new concept. The currently existing ports to IBM s390, L4 Microkernel [8] and XEN [3] are paravirtualization approaches and have proven that running Linux on a hypervisor is possible and efficient. With User Mode Linux (UML) [6], even a port to Linux itself exists, where the virtual machine is a Linux process on the host system.

Another method for running Linux (or in fact, *any* operating system) in a virtual machine is full system virtualization. Virtualizers like VMWare [17], Virtual PC or QEMU [4] execute an unmodified operating system environment, where the OS kernel is unaware of the virtualization. The technique typically used by these virtualizers is dynamic recompilation of the binary code at runtime, when the code cannot be executed directly. In most cases all access to hardware is emulated, it is almost impossible to enable direct access to the host system's hardware without compromising the entire system. Only simple, protocol based hardware like USB can be directly virtualized.

The emulation approaches require a set of drivers on the host system to be used by the virtualized hardware. For example, some kind of graphics driver is necessary to display the emulated framebuffer. However, this also increases the trusted code base of the host system, which makes this approach infeasible for our purposes.

We use the paravirtualization approach where the Linux kernel is aware of the microkernel environment. Then, Linux can properly use the assigned components of the underlying hardware *and* use specialized drivers e.g. to communicate with other virtual machines.

Our paravirtualized kernel, named *P4Linux*, was designed according to the following design goals:

- Inhibit any side effects on other virtual machines.
- Keep full ABI compatibility, don't rewrite applications.
- Keep overhead small for fast execution.
- Ease porting to other processor architectures.

Details on the Implementation

The Linux architecture layer is scalable enough to run on either a small embedded system, or a large multi processor server cluster. Therefore, we implemented the microkernel awareness like a new architecture.

To do so, the following concepts had to be adapted:

- Virtual Memory Management
- I/O resource management
- Exception Handling
- Interrupt handling

Since User Mode Linux (UML) was already already available, and since it uses similar abstractions, we used its "SKAS" approach as a starting point.

Memory Management The Linux memory management layer internally uses multilevel pagetables like on x86 to maintain the virtual address space of its userspace processes. After modifications to a pagetable, the Linux kernel has hooks to flush a CPU's Translation Lookaside Buffers (TLBs). Taken from the UML approach, we hooked these functions, too, and used the mechanisms provided by the microkernel to modify address spaces. The microkernel offers system calls to map a set of pages from one address space to another, and to revoke this mapping by unmapping pages from one's address space.

However, the mapping system call transfers pages from the virtual address space of one task to the virtual address space of another. This stands in contrast to a *normal* Linux implementation, where physical pages are mapped to virtual addresses in a process' address space.

From the Linux kernel's point of view, two different virtual memory views exist: the virtual memory of its own address space and the virtual memory of its userspace processes.

So, to keep the implementation easy, only one big chunk of physically contiguous memory can be used as Linux main memory. This eases calculation of DMA transfer addresses, as a fixed offset is used to calculate bus addresses.

Also, there is a restriction: as the P4Linux kernel and all user space processes run in their own address spaces, the Linux kernel cannot access the memory of its userspace processes directly. Taken from the UML approach, all copy in and out operations by the kernel must be translated back to the corresponding addresses in the Linux kernel address space.

Memory mapped I/O resources are no problem at all. The Linux kernel starts with all I/O resources it can access already mapped at start up. A driver calling *ioremap()* to get a virtual mapping of an I/O device now gets a pointer to these premapped memory areas.

Exceptions From the microkernel's point of view, all Linux userspace activities cause exceptions. Exceptions, like TLB misses or non-microkernel system calls, are propagated to a userspace exception handler. For Linux processes, the P4Linux kernel registers as this exception handler. On each exception, it receives an IPC containing the complete register context of the faulting process and tries to solve the exception by mapping a page in case of a pagefault, executing a Linux system call, or sending a signal.

Interrupts The microkernel's mechanisms to serve interrupts to userspace handlers need an active thread to wait for an interrupt. The P4Linux kernel offers a set of threads waiting for these interrupts and other asynchronous events, for example reception of data from other partitions. Finally, these threads call the Linux *do_IRQ()* implementation to invoke the registered handlers.

An interrupt blocking mechanism is implemented by priority: if the Linux kernel doesn't want to receive interrupts, it raises its priority above all asynchronous helper threads and it lowers its priority again after the critical section. This mechanism is quite fast, because the microkernel supports lazy priority switching.

Problems and Side-Effects

The isolation between partitions crucially depends on the microkernel's address spaces, which are implemented by means of memory management hardware (an MMU). But if there exist devices which are able to access the bus directly, bypassing the MMU, then this safety concept fails. The next generation machines are likely to have IOMMUs that will solve this problem once and for all, but in current hardware, there are several devices capable of mastering the bus while bypassing the MMU. Any guest operating system that has access to one of these busmasters is indirectly able to access any location in the system, i.e. it can cause any amount of damage.

One solution for this problem is to restrict hardware access only to polling devices or framebuffers and revoke any bus mastering capabilities from the devices.

Another solution is to divide a driver into a trusted and an untrusted section: only the trusted parts must access the hardware and are implemented outside the Linux kernel in a separate server, whereas the untrusted parts remain inside the Linux kernel and handle the software stacks.

Both solutions introduce additional communication overhead.

The current implementation is also restricted to uni processor execution, due to its simple but fast

locking approach via priority switching.

Performance

An approach to port Linux to a microkernel can never be as fast as a native Linux kernel running on the bare hardware.

We benchmarked the overhead of our approach with two different benchmarks: a microbenchmark calling performance critical functions in a loop and a real-world benchmark, compiling a Linux kernel. The target platform for the tests is a 600 MHz Intel Celeron / 256 MB RAM embedded industrial PC with Debian 3.1 installed. We chose this platform, as it is widely used in fanless industrial environment.

The Linux kernel used for this benchmark is taken from SYSGO's ELinOS distribution. It is a modified 2.6.15 kernel. We compare the time of a kernel build on a native i386 target against our microkernel environment, running P4Linux. When possible, both kernels utilize the same set of drivers and configuration options.

The microbenchmark calls performance critical functions like *getpid()*, a null system call, 1000 times in a loop and compares the CPU's time-stamp counter before and after the loop. The time-stamp counter on the x86 architecture is incremented every CPU cycle. The presented values are the normalized results of 1000 calls, also presented in CPU cycles.

Testcase	Native	P4Linux	Factor slower
getpid()	344	7004	20,4
fork()+_exit()	66428	388883	5,9
vfork()+_exit()	19199	54524	2,8
fork()+execve()	216041	1537256	7,12
vfork()+execve()	214620	1536807	7,16

TABLE 1: *Microbenchmark*

The *getpid()* benchmarks shows an overhead of about 6700 CPU cycles for all system calls compared to the native implementation, caused by two address space switches and IPC messages for one system call.

Especially performance critical are *fork()* and *execve()* operations. The table shows the differences in process creation between *fork()* and *vfork()* when the created child immediately terminates. The last two tests show the costs of address space space filling and immediately flushing (the forked child process terminates immediately).

Linux	Compilation time
Native Linux	361 s
P4Linux	441 s

TABLE 2: *Compiling the Linux Kernel*

The macrobenchmark shows the overall system performance impact. We compiled a Linux kernel (2.6.16 for i386, configuration "allnoconfig") on the test system and measured the overall time. The P4Linux system is about 22 percent slower. This is mainly caused by extensive use of the *fork()* system call by the build system.

5 Conclusion and Outlook

In this paper, we introduced a new approach to coexistence of a softPLC and Linux in a single machine. Unlike previous solutions, this one does not force the two components to trust each other. It thus makes it possible to apply the approach to safety-critical systems. The method does have an impact on system performance, however, we feel that this impact is acceptable: most of today's computer systems do not suffer from lack of performance, instead they have severe safety and security problems. With this background it seems sensible to sacrifice some performance while gaining significantly on the safety and security side. The softPLC is only one example of a real-time system that can be hosted by PikeOS, there are many others.

Current work on the PikeOS system aims at providing a large variety of real-time or non-real-time operating system interfaces to run on top of PikeOS (one of them being the CoDeSys softPLC). Work on the softPLC itself aims at increasing the number of fieldbuses that it supports.

In the next stage, the softPLC system is planned to be turmed into a networked, distributed system of PLCs which can share portions of their state accross a network by means of a publish/subscribe mechanism.

From the Linux point of view, the next goal is to approximate performance to the native implementation. One of the most important issues is to reduce process creation time. Furthermore, the overhead of processor mode switches can be decreased by batching system calls. Additionally, new TLB interfaces are verified, which support page table virtualisation.

Future improvements to the PikeOS Microkernel are support for multicore systems and current hardware virtualisation techniques for the x86 architecture.

References

[1] ARINC. Avionics Application Software Standard Interface. Technical Report ARINC Specification 653, Aeronautical Radio, Inc., 1997.

[2] M. Barabanov. A Linux-based RealTime Operating System, 1997.

[3] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization, 2003.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] E. Bianchi and L. Dozio. Some experiences in fast hard realtime control in user space with rtai-lxrt. 2nd Realtime Linux Workshop, Orlando, 2000.

[6] J. Dike. User-mode Linux. Online: user-mode-linux.sourceforge.net, 2001.

[7] A. Dunkels, L. Woestenberg, K. Mansley, and J. Monoses. lwIP embedded TCP/IP stack. http://savannah.nongnu.org/projects/lwip/, Accessed 2004.

[8] M. H. Hermann Haertig and J. Wolter. Taming linux, 1998.

[9] IEC. Functional Safety of Programmable Electronic Systems: Generic Aspects. IEC 65A (Secretariat) 123, International Electrotechnical Commission, February 1992. Technical Committee no. 65, Working Group 10 (WG10).

[10] IEC. IEC 61131-3: Programmable Controllers - programming languages. Technical report, International Electrotechnical Commision, 1993.

[11] IEEE. *1003.13-1998 IEEE Standard for Information Technology — Standardized Application Environment Profile (AEP) — POSIX® Realtime Application Support*. 1998.

[12] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 237–250, 1995.

[13] J. Liedtke. L4 Reference Manual - 486, Pentium, Pentium Pro, 1996.

[14] J. Liedtke. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proceedings of 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 5–6 1997.

[15] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux J.*, 2000(72es):10, 2000.

[16] R. Morley. *History of the PLC*. R. Morley Inc.

[17] VMware. VMware ESX Server Online Documentation, 2005.

[18] P. Wurmsdobler. Linux for real-time PLC control? Slower is easier. *InTech*, Industrial Computing:49–51, November 2001.

Energy Efficient Hard Real-Time DVS Algorithm

Cheng Yu, Wang Hua Yong, Ren Jie, and Yang Jian
Department of Computer Science and Technology
Tsinghua University, Beijing 100084, P.R.China
yuchen@tsinghua.edu.cn

Abstract

This paper proposed a novel hard real-time dynamic voltage scheduling (RT-DVS) algorithm for periodically hard real-time task set. Based on a mathematical system model which meets the real situation, the proposed RT-DVS algorithm fully exploited all the slack times, gave an optimal solution for the hard real-time task on CPUs with two adjustable frequencies and voltages. Finally, the optimality of RT-DVS algorithm is proven. The RT-DVS algorithm can be extended to used on CPUs with more adjustable frequencies, and is useful in multitask environments.

1 Introduction

Dynamic voltage scaling (DVS)[1] is one of the most effective approaches in reducing the power consumption of embedded systems. The supply voltage can be dynamically reduced to the lowest possible extent. In real-time systems, it still ensures a proper operation when the required performance of the system is lower than the maximum performance.

As we know, idle time always exists in real-time scheduling systems. There are three possible situations for these idle times: 1) the workload of real-time system is light. 2) the schedule algorithm cannot fully utilize CPU resources.For example, the ability of RM scheduling algorithm is: $U \leq n(2^{1/n}-1)$ U is the utilization ratio of CPU. 3) the execution time of tasks is usually less than the Worst Case Execution Time (WCET). In the past, various voltage scheduling algorithms have been proposed for hard real-time systems[1,2,3]. Given multiple tasks, these algorithms assign the proper speed to each task dynamically while guaranteeing all their deadlines. In real-time systems, since the real execution time of each task may be smaller than the worst-case execution time (WCET), workload variation slack times are generated at run time even though the worst-case utilization of the processor is 1. However, it is difficult to utilize the workload-variation slack times because the exact amount of slack time before the completion of a task cannot be known. Therefore, most DVS scheduling algorithms transfer the slack time to the following tasks that can utilize it. But these algorithms may not be effective in reducing the energy consumption [22]

In this paper, a novel RT-DVS algorithm is proposed. This algorithm utilizes idle times and reduces CPU frequency and voltage within real-time job, and it allow a slower execution of real-time task without break deadline. Therefore parts or all of the idle time would be eliminated, with assured real-time ability and energy savings. Finally, the proposed RT-DVS algorithm also has been proved to be optimality.a

2 Related Works

Since the dynamic energy consumption of CMOS circuits, which dominates the total power consumption, is proportional to the square of the supply voltage , a significant energy reduction is possible with the DVS scheme. The main dissipation of digital CMOS circuits comes from dynamic power [1,3]:

$$P_{dynamic} = C_L \times N_{SW} \times V_{DD}^2 \times f_{clk} \quad (1)$$

In formula (1),C_L is a physical quantity which depends on the characteristic of capacitance, $N_{SW}$is the reversal time of circuit per clock cycle, $V_{DD}$is the supply voltage of circuit, and $f_{clk}$is the clock frequency. Formula (1) suggests that the effective way to reduce power is to reduceV_{DD}, which would

increase the latency of CMOS circuit simultaneously and consequently reduce clock frequency of circuit[1,4,5]:

$$f_{clk} \approx k\frac{(V_{DD}-Vt)^2}{V_{DD}} \quad (2)$$

In formula (2), k is constant, $V_t$is the voltage threshold of CMOS circuit. Because the dynamic power of CMOS circuit $P_{dynamic}$has linear relation with clock frequency $f_{clk}$and the energy dissipated per cycle with CMOS circuitry scales quadratically to the supply voltage V_{DD}, the joint effect of simultaneously adjusting clock frequency and voltage to power is approximately cube relation. The frequency adjustment always accompanies voltage adjustment.

For hard real-time systems where timing constraints must be strictly satisfied, a fundamental energy delay tradeoff makes it more challenging to adjust the supply voltage dynamically while minimizing the energy consumption and guaranteeing the timing requirements. For this reason, extensive studies have been recently carried out on the InterDVS problems[22].The recent researches have considered the above problems. Tohru Ishihara and Hiroto Yasuura [20] addressed the discrete voltage adjustment problem, but not the different characteristic of energy problem. Hakan Aydin [21] discussed the problem of tasks with different energy consumption characteristics, and analyzed the problem with continuous voltage adjustment and gave an optimal solution. Pedro Mejia-Alvarez [4] analyzed the discrete one and only gave an approximately optimal solution, and believed that different energy consumption characteristics in the status of discrete adjustable voltage equals to 0-1 knapsack problem, which is a NP problem and difficult to give optimal solution. However, this article believed that although different tasks have different energy consumption characteristics, if the energy consumption characteristic in the same task possesses uniformity (during executing time, task slices in different time region have the same energy consumption characteristic), the mathematic model could be modified from a 0-1 knapsack problem to a non 0-1 knapsack problem, which is easy to give an optimal solution. The real-time system model proposed in this paper supports tasks with different energy consumption characteristics, and considers the situation that CPU with two or more adjustable frequencies, and the cost of CPU frequency scaling. The performance evaluation section in this paper gives a comparison of these two models and related algorithm.

3 Hard Real-Time System Model

We notice that most recent CPUs only support finite discrete voltages and frequencies, and some common CPUs could only support two or three discrete voltages and frequencies. There are different effects on different tasks using DVS, for example, network communicating program spends more energy on pheripherials. The overhead of CPU frequency scaling includes time latency and energy consumption, most of which comes from the time latency and energy consumption of internal circuit initialization during CPU frequency scaling. The cost of one CPU frequency scaling is fixed in most cases.

From above situation, the hard real-time system model presumes that each task is periodically preempted and independent between each other. The tasks that running on real-time scheduling system form a task set: $\tau = (\tau_1, ..., \tau_2)$Any task consists of a sequence of jobs. Periodical task τ_i is described by a triple $\tau_i = (\phi_i, p_i, c_i)$. ϕ_i is the initial phase, which is the arrival time of the first job. p_i is the period of the task. c_i is the CPU clock cycle required to execute the task. If the CPU frequency is f, then the execution time of the task $t_i = c_i/f$. We assume the CPU has two set of adjustable frequency and voltage: (f_1, u_1) and (f_2, u_2) , in which $0 < f_1 < f_2$ and $0 < u_1 < u_2$. The objective of RT-DVS algorithm is to assign each task τ_i an appropriate frequency $\mathrm{F}(\tau_i)$ ($\mathrm{F}(\tau_i) \in (f_1, f_2)$) so that no task could exceed its deadline and the total energy consumption of the task set is minimal. The energy consumption of a job in the task τ_i under frequency $F(\tau_i)$ is $E(\tau_i, F(\tau_i))$. If there is a task which works under the altered frequency, then the frequency scaling time of all the other tasks would be affected.

Definition 1: The job frequency scaling time (shortened form: job scaling time) means the time required to scale the CPU's running frequency to a demanded one when a job starts to execute. If this job preempts another job and when this job is finished, the control returns to the previous preempted job, the frequency scaling time of the later job also includes the time required to scale the CPU's running frequency back to the one before execution. The scaling time of being preempted by another higher priority job is not included in the job's scaling time. The maximum of the all job scaling times is T_s **Definition 2:** The job frequency scaling energy consumption (shortened form: job scaling energy) means the CPU's energy consumption during job context switch. The maximum of all the job scaling

energies is E_s.
The execution time cost of a job in the task τ_i under frequency $F(\tau_i)$ is, after considering the switch cost, $t_i = \frac{c_i}{F(\tau_i)} + T_s$, and the energy consumption is $E(\tau_i, F(\tau_i)) + E_s$. The calculation below all considers the CPU status switch cost.
The schedulability of a periodically task set is decided by CPU's utilization ratio U_τ[6]

$$U_\tau = \sum_{i=1}^{n}(t_i/p_i) \leq U_{max} \quad (3)$$

U_{max} is the upper limit of the CPU's utilization ratio supported by the scheduling algorithm. If U_τ exceeds U_{max} , then task set τ is always non-schedulable. Different scheduling algorithms have different U_{max}. $U_{max} = n(2^{1/n} - 1)$ in RM algorithm, while $U_{max} = 1$ in EDF algorithm. For a static scheduling algorithm U_{max} is a constant. The utilization ratio of CPU for task τ_i under frequency $F(\tau_i)$ is:

$$U(\tau_i, F(\tau_i)) = \frac{t_i}{p_i} = \frac{\frac{c_i}{F(\tau_i)} + T_s}{p_i} \quad (4)$$

So the schedulability condition of task set τ is:

$$U_\tau = \sum_{i=1}^{n} U(\tau_i, F(\tau_i)) \leq U_{max} \quad (5)$$

For the scheduling problem of periodical tasks, only the things inside a hyperperiod (the lowest common multiple of all the task periods) need to be considered. The number of jobs in task τ_i during one hyperperiod is P/P_i, and the energy consumption is:

$$E_p(\tau_i, F(\tau_i)) = \frac{P}{P_i} \times (E_s + E(\tau_i, F(\tau_i))) \quad (6)$$

The energy consumption of the whole task set per hyperperiod is:

$$E = \sum_{i=1}^{n} E_p(\tau_i, F(\tau_i)) \quad (7)$$

Therefore the RT-DVS can be described as: Select an appropriate $F(\tau_i)(1 \leq i \leq n)$, minimize equation (7) under the confinement (5).

Definition 3: In the situation without reducing frequency, the CPU utilization ration and energy consumption in a hyperperiod for the whole task set τ is defined as,
With frequency scaling overhead:

$$U_{min} = \sum_{i=1}^{n} U(\tau_i, f_2),$$

$$E_{max} = \sum_{i=1}^{n} E_p(\tau_i, f_2)$$

Without frequency scaling overhead:

$$U_{min}^{`} = \sum_{i=1}^{n} \frac{c_i}{f_2 \times p_i}$$

$$E_{max}^{`} = \sum_{i=1}^{n} \frac{P}{P_i} \times E(\tau_i, f_2)$$

If $U_{min} > U_{max}^{`}$, then the task set τ is non-schedulable; otherwise it is impossible to dynamic adjust the voltage if $U_{min} \geq U_{max}$. The bellowing text only consider the situation when $U_{min} < U_{max}$. We define:

$$\triangle U_i = U(\tau_i, f_1) - U(\tau_i, f_2) \quad (8)$$

Because the execution time of the task and the utilization ratio of CPU would increase if the CPU frequency is reduced, then $\triangle U_i > 0$. Also we could define:

$$\triangle E_i = E_p(\tau_i, f_2) - E_p(\tau_i, f_1) \quad (9)$$

Only the situation when $\triangle E_i > 0$ is need to be considered. Because if$\triangle E_i \leq 0$, then there is no reduction of energy consumption for taskτ_i after reducing the frequency of CPU, which obviously dissatisfy the original intention of the RT-DVS algorithm. The RT-DVS algorithm would never consider reducing frequency for any of these tasks, therefore the situation on $\triangle E_i \leq 0$ is never considered. The equivalent form of condition (5) according to equation (8) is :

$$\sum_{i=1}^{n} \triangle U_i \times X_i \leq U_{max} - U_{min} \quad (10)$$

Then according to formula (9), condition (7) equals:

$$\sum_{i=1}^{n} \triangle E_i \times X_i = E_{max} - E \quad (11)$$

$$\text{in which } X_i = \begin{cases} 0 & F(\tau_i) = f_2 \\ 1 & F(\tau_i) = f_1 \end{cases}$$

Until now, the RT-DVS algorithm can be further described as: choose appropriate$(X_1, ..., X_n)$, maximize equation (11) under the confinement (10). So it is equivalent to 0-1 knapsack problem.

n order to avoid the occurrence of 0-1 knapsack problem, it is allowed to choose at most one main task $\tau_k(1 \leq k \leq n)$ in task set τ, and the RT-DVS algorithm setups a fixed length area for the main task and then reduce the frequency in that area. In figure 2(a), the front part of the main task is running in frequency f_1, while the rear in f_2. In figure 2(b),

the front part is in f_2 and the rear in f_1 . Because of the uniformity of energy consumption characteristics inside the task, figure 2 has no intrinsic differences. In practice, the partial reduce frequency method could be implemented by using OS timer.

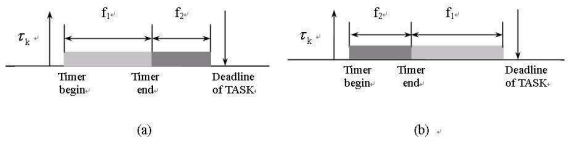


FIGURE 1: *Main Task and The Area of reducing frequency*

T-DVS algorithm could first fix the CPU utilization ratio U_k of τ_k after reduce frequency. U_k saticifies:

$$U(\tau_k, f_2) < U_k < U(\tau_k, f_1) \quad (12)$$

Then the RT-DVS algorithm calculates the length of area of reducing frequency. If the clock cycle of the area of reducing frequency is $c_{k1}(0 < c_{k1} < c_k)$, then the clock cycle of the non reducing frequency is $c_k - c_{k1}$, and the total execution time (includes one frequency scaling time) is:

$$t_k = \frac{c_{k1}}{f_1} + \frac{c_k - c_{k1}}{f_2} + T_s \quad (13)$$

There is one extra frequency scaling overhead inside the main task at the startup time. If the CPU is running under f_1 before the main task τ_k start, τ_k is scheduled according to figure 2(a), otherwise according to figure 2(b). Therefore the frequency scaling for startup can be eliminated to compensate for the extra frequency scaling cost in the middle of task execution. According to formula (13), the CPU utilization ratio of τ_k is:

$$U_k = t_k/p_k = \frac{\frac{c_{k1}}{f_1} + \frac{c_k - c_{k1}}{f_2} + T_s}{p_k} \quad (14)$$

Therefore,

$$c_{k1} = \frac{U_k \times P_k - T_s - \frac{c_k}{f_2}}{\frac{1}{f_1} - \frac{1}{f_2}} \quad (15)$$

IF U_k satisfy formula (12), then the c_{k1} from equation (15) satisfy the condition that $0 < c_{k1} < c_k$ The corresponding X_k for the main task τ_k is defined as:

$$X_k = \frac{U_k - U(\tau_k, f_2)}{\triangle U_k} \in (0, 1) \quad (16)$$

Therefore the RT-DVS algorithm equals an algorithm for non 0-1 knapsack problem. The steps of this RT-DVS algorithm is: choose an appropriate vector $(X_1, ..., X_n)$, of which at most one element $X_k \in (0, 1)$ and the other $X_i \in (0, 1)$, maximize equation (11) under the confinement (10). The vector $(X_1, ..., X_n)$ is the optimal solution of RT-DVS algorithm.

4 Optimal Solution of RT-DVS Algorithm

RT-DVS Algorithm is a greedy algorithm. For a non 0-1 knapsack problem, the greedy algorithm could get the optimal solution. The algorithm is shown below:

- Step 1: for all $\tau_i \in \tau$ and the corresponding parameters $\triangle E_i, \triangle U_i$ and $\triangle X_i$, sort them according to the descending order of $S_i = \triangle E_i / \triangle U_i$, and re-set subscripts of $\tau_i \triangle E_i, \triangle U_i$ and $\triangle X_i$.
- Step 2: set up variable $U_{free} = U_{max} - U_{min} > 0$
- Step 3: clear vector $(X_1, ..., X_n)$ to zero
- Step 4: select the first items in the sorted sequences as $\triangle E_i$ and $\triangle U_i$. IF $\triangle U_i \leq U_{free}$, then let $X_i = 1, U_{free} = U_{free} - \triangle U_i$, delelete $\triangle E_i$ and $\triangle U_i$. Then repeate Step 4.
- Step 5: if the sequence is not empty and $U_{free} > 0$, select the first item in the sorted sequence as $\triangle E_k$ and $\triangle U_k$. Let the task τ_k be the main task, $X_k = U_{free} / \triangle U_k$. Calculate the clock cycle in the area of reducing frequency according to formula (15).

The Sort procedure (Step 1) sorts the task set. The subscription of the task set and the corresponding task parameters follow the descending order of . Obviously the time complexity depends on the sort complexity, therefore the time complexity of this algorithm is O(nlogn) . The space complexity of this algorithm is O(n) which depends on the task parameters and the vector space $(X_1, ..., X_n)$

The following case explains the execution process and energy saving effect. Suppose the CPU has two adjustable frequency: 100Mhz and 50Mhz, frequency scaling time $T_s = 0.01s$, frequency scaling energy consumption $E_s == 0.01J$.The characteristic of task set $\tau = (\tau_1, \tau_2, \tau_3)$ is:

Table1: the basic parameters of task set

	Time clock	WCET 100MHz	WCET 50MHz	$E(\tau_i,100MHz)$	$E(\tau_i,50MHz)$	Period (p_i)	The num of Jobs in Hyperperiod (P/p_i)
τ_1	300M	3s	6s	0.3J	0.25J	12s	1
τ_2	200M	2s	4s	0.2J	0.1J	6s	2
τ_3	100M	1s	2s	0.15J	0.05J	6s	2

Table2: the energy related parameters used in RT-DVS algorithm

	$E(\tau_i,100MHz)+E_s$	$E(\tau_i,50MHz)+E_s$	$E_p(\tau_i,100MHz)$	$E_p(\tau_i,50MHz)$	ΔE_i
τ_1	0.31J	0.26J	0.31J	0.26J	0.05 J
τ_2	0.21J	0.11J	0.42J	0.22J	0.1J
τ_3	0.16J	0.06J	0.32J	0.12J	0.1J

Table3: the time related parameters used in DVS algorithm

	t_i 100MHz	t_i 50MHz	$U(\tau_i,100MHz)$	$U(\tau_i,50MHz)$	ΔU_i	$S_i(\Delta E_i/\Delta U_i)$
τ_1	3.01s	6.01s	3.01/12	6.01/12	1/4	0.2
τ_2	2.01s	4.01s	2.01/6	4.01/6	1/3	0.3
τ_3	1.01s	2.01s	1.01/6	2.01/6	1/6	0.6

If the EDF algorithm is selected, then $U_{max}=1$. When the CPU's frequency is 100M Hz, then $U_{min}=\sum_{i=1}^{3} t_i/p_i = 19/24 < U_{max}$. Therefore the RT-DVS algorithm can be used and $U_{free}=5/24$. Because of the sort requirement on S_i, the algorithm considers τ_3 first. Due to $\triangle U_3 < U_{free}$, $X_3=1$ and τ_3 is allocated with frequency 50MHz, $U_{free}=U_{free}-\triangle U_3=1/24$. Secondly τ_2 is considered. Because $\triangle U_2 > U_{free}$, then τ_2 is the main task, $X_2=U_{free}/\triangle U_2=1/8$, the utilization ratio of CPU is $U_2=U(\tau_2,100MHZ)+U_{free}=\frac{2.01}{6}+\frac{1}{24}=\frac{9.04}{24}$. According to formula (15) the clock cycle of area of reducing frequency is 25M. Therefore 1/8 of the program is running under frequency 50MHz and the energy consumption of τ_2's job is $\frac{1}{8}\times 0.1+\frac{7}{8}\times 0.2+0.01=0.1975J$. Task τ_1 emains under the frequency 100MHz and $X-1=0$.

In a hyperperiod (12s), the job number of τ_1,τ_2 and τ_3 is respectively 1,2 and 2. After the procedure of RT-DVS algorithm, the utilization ratio of CPU of the task is 100% and there is no idle time in system. The energy consumption of the task set in one hyperperiod is

$$E=E_p(\tau_1,100MHZ)+2\times 0.1975+E_p(\tau_3,50MHZ)=0.825J$$

If there is no procedure of RT-DVS algorithm, then the energy consumption of the task set in one hyperperiod is $E^{`}=\sum_{i=1}^{3}E_p(\tau_i,100MHZ)=1.05J$. If the algorithm[4] is adopted, which did not support partial reduce frequency method, then the RT-DVS algorithm can only choose either τ_1 or τ_3 to reduce frequency (τ_2 reduce frequency or both τ_1 and τ_3 reduce frequency will break the deadline), therefore under the best circumstances, the energy consumption can only reach 0.85J and there is no possibility to eliminate idle time. In this case, The optimal RT-DVS algorithm can save at least 21% energy compared with the one without RT-DVS algorithm.

5 Proof of the Optimality of RT-DVS Algorithm

Obviously the solution $X1,...,Xn$ of the above algorithm has the form $1,1,...,X_k,0,...,0$ and $\sum_{i=1}^{n}\triangle U_i\times X_i=U_{max}-U_{min}$ in which τ_k is the main task, $0\le X_k\le 1$. Now prove the optimality of the algorithm by negative approach. The proof is shown below:

We first prove the situation of first dimension X_1 must be as big as possible. That is: if $\triangle U_1\le U_{max}-U_{min}$, then $X_1=1$ or $X_1=(U_{max}-U_{min})/\triangle U_1$. By deductive approach:

First, statement is correct apparently when n=1

Second, we assume statement is correct when n<h. Now consider the condition when n=h. If the statement isn't correct, then the optimal solution is $X_1^{`},...,X_n^{`}$, which satisfy three condition:

$$X_1^{`}<X_1 \quad (17)$$
$$\sum_{i=1}^{n}\triangle U_i\times X_i=\sum_{i=1}^{n}\triangle U_i\times X_i^{`} \quad (18)$$
$$\sum_{i=1}^{n}\triangle E_i\times X_i=\sum_{i=1}^{n}\triangle E_i\times X_i^{`} \quad (19)$$

According to (18),$\sum_{i=1}^{n}\triangle U_i\times X_i^{`}=(X_1-X_1^{`})\times\triangle U_1+\sum_{i=2}^{n}\triangle U_i\ge(X_1-X_1^{`})\times\triangle U_1>0$. Therefore, $X_2^{`},...,...X_n^{`}$,cannot be all zero. Then a third solution $X_1^{`},...,...X_n^{`}$ ould be constructed as follows: let $X_1^{``}=X_1$

$$X_i^{``}=\begin{cases}0 & \text{if}X_i^{`}=0(2\le i\le n)\\ X_i^{`}-\triangle & \text{if}X_i^{`}>0\end{cases}$$

Here $\triangle$ is a small adjustment value. By adjusting $\triangle$, we could let $\sum_{i=1}^{n}\triangle U_i\times X_i^{``}=U_{max}-U_{min}$. Also, because $\triangle E_i/\triangle U_i$ is sorted decreasingly, $\sum_{i=1}^{n}\triangle E_i\times X_i^{`}<\sum_{i=1}^{n}\triangle E_i\times X_i^{``}$, which means $(X_1^{`},...,X_n^{`})$is not the optimal solution. Contradiction exists, so we can get the conclusion: $X_1^{`}=$

X_1.
IF X_1 must be as big as possible, then the proposal deducted to n-1 length knapsack problem. Then $X_2^{'} = X_2, ..., X_n^{'} = X_n$ can be proved using the same proof. Therefore the RT-DVS algorithm is optimal.

6 Conclusion

The article proposed an optimal RT-DVS algorithm to periodically hard real-time work set problem on CPUs with two adjustable frequency. The proposed RT-DVS algorithm automatically selects appropriate intra-job locations where the supply voltage can be changed to minimize the energy consumption, satisfying the timing constraint. The proposed RT-DVS algorithm also has been proved to be optimality. The proposed Work in this paper can be extended in several directions. The future work would further extend to CPUs with several adjustable frequencies. The execution time of an application depends on the run-time events as well as the control flow, so we can integrate such run-time information to define more accurate system model and design new RT-DVS algorithm.

References

[1] Anantha P. Chandrakasan, Samuel Sheng, Robert W. Brodersen, 1999,*Low-Power CMOS Digital Design*,IEEE Journal of Solid-State Circuits, 27(4): 473-484.

[2] Luca Benini, Alessandro Bogliolo and Giovanni De Micheli, Jun 2000, *A Survey of Design Techniques for System-Level Dynamic Power Management*,IEEE Transactions on VLSI Systems, Vol. 8, Issue 3.

[3] Osman S. Unsal and Israel Koren,July 2003 *System-Level Power-Aware Design Techniques in Real-Time Systems*,Proceedings of the IEEE, Vol. 91, Issue 7,pages 1055- 1069

[4] Pedro Mejia-Alvarez, Eugene Levner, Daniel Mosse, 2004 *Adaptive Scheduling Server for Power-Aware Real-Time Tasks*,ACM Transactions on Embedded Computing Systems (TECS), 3(2): 284-306

[5] Marcus T. Schmitz, Bashir M. Al-Hashimi, Petru Eles, February 2004*Iterative schedule optimization for voltage scalable distributed embedded systems*,ACM Transactions on Embedded Computing Systems (TECS), Volume: 3, Issue: 1, Page: 182 - 217

[6] Buttazzo G C, 2000,*Hard Real-Time Computing System: Predictable Scheduling Algorithms and Applications*,Massachusetts: Kluwer Academic Publishers,

[7] Lars S. Nielsen, Cees Niessen, Jens Spars and Kees van Berkel, Dec 1994 *Low-power operation using self-timed circuits and adaptive scaling of the supply voltage*,IEEE Transaction on VLSI Systems,Vol. 2, Issue 4

[8] Jui-Ming Chang and Massoud Pedram, 1997 *nergy Minimization Using Multiple Supply Voltages*,EE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 5, Issue 4, pages 436-443.

[9] Mark C. Johnson and Kaushik Roy, July 1997,*Datapath Scheduling with Multiple Supply Voltages and Level Converters*,ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 2, Issue 3, pages 227-248

[10] Mark Weiser, Brent Welch, Alan Demers, Scott Shenker, 1994, *Scheduling for Reduced CPU Energy*,In Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation. California: USENIX Association, 13-23

[11] Kinshuk Govil, Edwin Chan, Hal Wasserman,December 1995, *Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU*,In Proceedings of the 1st Annual International Conference on Mobile Computing and Networking, page13-25

[12] Trevor Pering, Tom Burd and Robert Broderson. 1998, *The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms*,In Proceedings of International Symposium on Low Power Electronics and Design, page76-81

[13] Frances Yao, Alan Demers, Scott Shenker, 1995, *A Scheduling Model for Reduced CPU Energy*,In Proceedings of 36th Annual Symposium on Foundations of Computer Science. California: IEEE Computer Society Press, 374-382

[14] Inki Hong, Miodrag Potkonjak and Mani B. Srivastava, 1998,*On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage CPU* ,In Proceedings of IEEE/ACM International Conference on Computer-Aided Design, pages 653-656

[15] Gang Quan, Xiaobo Hu. 2003 *Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage CPUs*,In Proceedings of Design Automation Conference. California: IEEE Computer Society Press, 828-833

[16] Inki Hong, Darko Kirovski, Gang Qu, Miodrag Potkonjak and Mani B. Srivastava. 1998 *Power optimization of variable voltage core-based systems* In Proceedings of Design Automation Conference, pages 176-181.

[17] Youngsoo Shim and Kiyoung Choi, 1999, *Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems*, In Proceedings of 36th Design Automation Conference, pages 134-139.

[18] Seongsoo Lee and Takayasu Sakurai, 2000, *Run-time Voltage Hopping for Low-power Real-time Systems*, In Proceedings of 37th Design Automation Conference, pages 806-809.

[19] *Intel PXA250 and PXA210 Application CPUs Developer's Manual*,Intel Corporation , Feb.2002

[20] Tohru Ishihara and Hiroto Yasuura, 1998, *Voltage Scheduling Problem for Dynamically Variable Voltage CPUs*,In Proceedings of International Symposium on Low Power Electronics and Design, page197-202

[21] Hakan Aydin, Rami Melhem, Daniel Mosse, Pedro Mejia-Alvarez, 2001, *Determining Optimal CPU Speeds for Periodic Real-Time Tasks with Different Power Characteristics*,In 13th Euromicro Conference on Real-Time Systems. California: IEEE Computer Society Press, 225-232

[22] Dongkun Shin, Jihong Kim, OCTOBER 2005 *Intra-Task Voltage Scheduling on DVS-Enabled Hard Real-Time Systems*,IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 24, NO. 10,

Dynamic Kernel Thread Scheduling for Real-Time Linux

Dongwook Kang, Woojoong Lee, and Chanik Park
Department of CSE/GSIT
Pohang University of Science and Technology, Kyungbuk, Republic of Korea
{dwkang,wjlee,cipark}@postech.ac.kr

Abstract

Real-time tasks are supported in Linux by separating the real-time task priorities from non real-time task priorities. However, this separation of priority ranges may no longer be effective when real-time tasks make the system calls for important system service like the I/O and the kernel threads that take care of these system calls. In that sense, Linux is known for supporting soft real-time requirements. The kernel threads in Linux systems are important as basic execution units providing kernel services for non real-time tasks as well as real-time tasks. In current Linux implementation, kernel threads are configured to have static priorities for better throughputs. However, the static assignment of priorities to kernel threads may cause trouble for real-time tasks when real-time tasks require kernel threads to be invoked for the kernel service via system calls. This is due to the fact that kernel threads do not discriminate real-time tasks from non real-time tasks. For example, the kernel thread called *pdflush* writes back to disks dirty memory pages. If real-time tasks have made several number of I/O requests of memory page updates and *pdflush* isn't scheduled sufficiently, then the responsiveness of real-time tasks may be prolonged significantly due to the shortage of memory cache.

In this paper, we present a dynamic kernel thread scheduling mechanism with *Weighted Average PIP*, a variation of the *Priority Inheritance Protocol* (*PIP*). The scheduling algorithms assign proper priorities to kernel threads at runtime by monitoring the activities of user-level real-time tasks. Through experiments, we have shown that the algorithms can greatly improve the unexpected execution latency of real-time tasks.

1 Introduction

With the advance of scheduling algorithms in Linux kernel, Linux 2.6 provides O(1) scheduling complexities and enhances soft real-time requirements thereafter. Linux nowadays have taken a wide interest in mobile terminals, automotives, robot controls, and factory automation, and all these areas require broad spectrum of real-time requirements. To promote Linux adoption in various real-time environments, much research has been conducted to improve real-time performance of Linux. Among them, Ingo Molnar's real-time preemption patch [1] is considered the most important.

One of the most important features provided by Ingo Molnar's patch is the threaded interrupt [2], an interrupt handling technique on process context rather than the interrupt context. Through threaded interrupt technique, the preemption latency caused by interrupt handling can be reduced remarkably, resulting in more deterministic behavior of real-time task. However, since we need to assign priorities to each interrupt statically, priority inversion problem is inevitable. For example shown in Figure 1, two real-time tasks are related to an *IRQ thread* [2] as marked by the black arrows. To execute the *IRQ thread* with highest priority among them, the priority 50 is assigned, while 50 and 40 are assigned to the real-time tasks respectively. In this situation, the real-time task 1 experiences execution delay by the *IRQ thread*, which handles interrupts of the real-time task 2, as marked by the red arrow. The reason of this problem is that the priorities of *IRQ threads* are static while the relations among *IRQ threads* and real-time tasks change dynamically, same as other kernel threads.

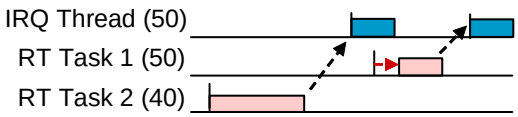


FIGURE 1: *The problem of IRQ threads in the real-time preemption patch*

Linux kernel threads are special tasks that provide specific kernel services requested by both non real-time and real-time tasks. They are different from user level application tasks in several points; they are automatically created by the kernel and are always executed in the kernel level. However, the kernel threads are treated as same scheduling entities as user level application tasks [4].

In current Linux implementation, kernel threads are configured to have static priorities for better throughputs. However, the static assignment of priorities to kernel threads may cause trouble for real-time tasks when real-time tasks require kernel threads to be invoked for the kernel service via system calls. This is due to the fact that kernel threads do not discriminate real-time tasks from non real-time tasks. For example, the kernel thread called *pdflush* writes back to disks dirty memory pages. If real-time tasks have made several number of I/O requests of memory page updates and *pdflush* isn't scheduled sufficiently, then the responsiveness of real-time tasks may be prolonged significantly due to shortage of memory cache.

The degree of performance impacts to real-time tasks by kernel threads varies due to different kernel threads' functionalities, thus recommending the different priority assignment policies for each kernel thread. In this paper, we show how they affect the real-time tasks through some experiments and how they propose a novel scheduling mechanism for the kernel thread in Linux, which monitors activities of real-time tasks and assigns proper priorities to kernel threads dynamically that take the relations between real-time tasks and kernel threads into consideration.

In Section 2, the related works are given. Section 3 describes the problem in kernel thread scheduling caused by Ingo Molnar's patch in current Linux implementation. In Section 4, we propose our kernel thread scheduling algorithm which dynamically adjusts the priorities of kernel threads by monitoring real-time tasks activities. The implementation of our algorithm is presented in Section 5 and the performance evaluation given in Section 6. Finally, conclusion and future works are described in Section 7.

2 Related Works

There were several research works to enhance the real-time performance of Linux. Firstly, Ingo Molnar developed the real-time preemption patch [1]. This patch adds three main technologies to enhance the real-time performance of Linux, such as *IRQ thread*, RT mutex, and high resolution timer [2][6]. The *IRQ thread* is a kernel thread handling top-halves of interrupts, which is waken up by ISRs when interrupts occur. This mechanism reduces preemption latency of vanilla kernel, which is caused by interrupt handling in interrupt context. This patch is widely used for real-time Linux area, and the performance is excellent [7]. However, the problem of the *IRQ thread* is that its priority is static. It is recommended to assign the priority based on the priorities of real-time tasks using the *IRQ thread*. However, the set of real-time tasks changes dynamically.
In [3], interrupts are also handled by interrupt service tasks whose role is the same as the *IRQ threads*. However, while real-time preemption patch allows all ISRs to start execution at the occurrence of IRQs, some IRQs are masked selectively and the corresponding ISRs are not executed in [3]. This is done by assigning priorities to IRQs and masking IRQs with the lower priorities. This reduces preemption latencies due to the waking up of ISTs or *IRQ threads* as well as interrupt handling in interrupt context. However, [3] only concentrates on prohibition of the preemption latency caused by interrupts, and does not suggest how to dynamically assign priorities to IRQs in detail.

3 Problems of Previous Kernel Thread Scheduling in Real-time Linux

Real-time tasks in Linux may suffer response time increase when associated kernel threads are not scheduled appropriately. In the following subsections, we present the examples of *IRQ thread*, *ksoftirqd*, and *pdflush*.

3.1 *IRQ thead* and *ksoftirqd*

In Linux as a general purpose operating system, top-halves of interrupts are handled in interrupt context with a higher priority than all tasks. And the bottom-halves are handled by a kernel thread *ksoftirqd*. However, when Ingo Molnar's real-time preemption patch is applied, top-halves of interrupts are handled by *IRQ thread* for real-time tasks to be able to preempt interrupt handlers.
To obtain the advantage to its fullest, the priorities of the *IRQ thread* and *ksoftirqd* have to be properly assigned by users. However, since the priorities are static, a problem may occur as mentioned in section 1.
This problem becomes more serious when the higher priority task rarely uses an external device while the lower priority task uses the device. This is because

the priorities are static but the priority of the real-time task related to *IRQ thread* changes dynamically.

3.2 *pdflush*

pdflush is a kernel thread which writes back dirty pages to disks to control dirty ratio of the system. When it is not scheduled for a long time, it can affect the response time of real-time tasks requesting disk writes or requiring additional free memory pages. To be more specific, when a real-time task performs disk read I/Os, the kernel tries to allocate multiple free pages to buffer read data. At that time, if the number of free pages is below specified threshold because of the starvation of *pdflush*, then the real-time task performs reclaiming used pages by itself to keep the number of free pages above the threshold. This is required because if all free memory has been used, then the kernel might easily get trapped in a deadly chain of memory requests that lead to a system crash [5].
The case of a disk write task is similar. For the same reason of maintaining dirty ratio below the specific threshold, the real-time task writes back dirty pages to disk by itself and its response time increases consequently when the dirty ratio crosses the threshold.

4 Dynamic Kernel Thread Scheduling for Real-time Systems

To solve the problems mentioned in section 3, we suggest a dynamic kernel thread scheduling algorithm, *weighted average PIP*. The *weighted average PIP* algorithm is a variation of PIP, the Priority Inheritance Protocol [10].
When *PIP* is applied to kernel thread scheduling, the priority of a kernel thread k_i is the maximum priority of R_i, a set of real-time tasks related to k_i, as shown in equation (1). In this case, the weak point is that the kernel thread can use CPU excessively and other real-time tasks can suffer unnecessary execution latency. Originally, the kernel threads are executed in background and perform their jobs minimizing interruption of normal tasks.
Thus, the alternative plan is *weighted average PIP* presented in equation (2). Let *default_prio(k_i)* be the default priority of the kernel thread k_i, and let *max_prio(R_i)* is the maximum priority of the real-time tasks in R_i. When there is no real-time tasks related with the kernel thread k_i, the default priority is assigned to k_i. Otherwise, the priority of a kernel thread is the average value of priorities of R_i multiplied by the weight values of k_i, w_i. Since the average value reflects the priorities of all real-time tasks in R_i and weight value reflects each kernel thread's own characteristics, *weighted average PIP* can make up the weak point of PIP. And this priority is bounded on the maximum priority of R_i not to disturb higher priority real-time tasks.

$$prio(k_i) = \begin{cases} default\_prio(k_i) & \text{if } |R_i| = 0, \\ max\_prio(R_i) & \text{if } |R_i| \neq 0. \end{cases} \quad (1)$$

$$sum\_prio = \frac{w_i}{R_i} \sum_{r_j \in R_i} prio(r_j) \quad (2)$$

$$prio(k_i) = \begin{cases} default\_prio(k_i) & \text{if } |R_i| = 0, \\ \min(sum\_prio, max\_prio(R_i)) & \text{if } |R_i| \neq 0. \end{cases} \quad (3)$$

4.1 *IRQ thread* and *ksoftirqd*

To apply the *weighted average PIP* to *IRQ thread* and *ksoftirqd*, the set of real-time tasks related to the two kernel threads, R_{irq}, is required. A real-time task has the relation since it requests an I/O job until the request is completed by *IRQ thread* and *ksoftirqd*. And whenever a relation is created or terminated, the priorities of the two kernel threads are recalculated using the Equation (2).

4.2 *pdflush*

An real-time task is maintained in $R_{pdflush}$ during the dirty pages, and its yields exist in the page cache in kernel. In other words. the relation is started when the real-time task writes to a file yielding dirty pages, and is terminated when the dirty pages are written back to disks.
To apply the relative importance of each real-time task to computation of the average priority, the number of files or inodes that the real-time task makes dirty is taken into consideration. For example, if a real-time task opens and writes to three files, the task is inserted to $R_{pdflush}$ three times. And when the dirty pages in one of the three files are written back, the task is removed from $R_{pdflush}$. This mechanism is efficient because the dirty pages are managed in the inode unit and written back in the inode unit as well.

5 Implementation in Linux 2.6

Whenever a change occurs in R_i, a set of real-time tasks related to a kernel thread k_i, the priority of

k_i is recalculated using Equation (2). To do this recalculation, some information is maintained for each kernel thread, such as the default priority of k_i, R_i, current average and maximum priority of R_i, and the number of real-time tasks in R_i. Thus we add this information as member variables in TCB, which is defined as a structure *task_struct* in Linux. In following sections, we explain how these new variables are maintained in each kernel thread.
However, managing the set R_i is a complex routine. R_i is needed to recalculate the maximum priority of R_i whenever a real-time task is added to or removed from it. In contrast, it is not required when obtaining average priority of R_i. The average priority can be calculated from previous average priority, the number of tasks in R_i, and the priority of the real-time tasks added to or removed from R_i.
If w_i is less than 1, the weighted average priority is always less than the maximum priority of R_i. Thus, in this case, it is not required to manage R_i. Among kernel threads we've mentioned, *pdflush* is in the case.
We implement our new kernel thread scheduling algorithm on Linux 2.6.15 with Ingo Molnar's real-time preemption patch.

5.1 *IRQ thread* and *ksoftirqd*

In section 4.2, it is explained that a real-time task has the relation with R_{irq} since it requests an I/O job to external devices and until the I/O job is completed by *IRQ thread* and *ksoftirqd*.
Thus, we add the currently running real-time task to R_{irq} right before this task adds the I/O job to the I/O request queue and is blocked, and we remove it right after the task is woken up.

5.2 *pdflush*

In our implementation, the weighted value of *pdflush* is less than 1, and we don't maintain $R_{pdflush}$.
In addition to the task_struct structure of *pdflush*, a few variables are added to the inode structure in Linux, such as the number of related real-time tasks and their average priority. This is because the pages become dirty in a unit of an inode or a file and are also maintained and written-back in a unit of the inode.
When a real-time tasks invoke write() system call to a file and pages of the file become dirty, the variables in the inode structure are updated and the priority of *pdflush* is recalculated based on the updated variables. If all dirty pages of the inode are written-back to the file, then the priority of *pdflush* is also recalculated with the variables of the inode structure.

6 Performance Evaluation

Through experiments, we compare the performance of our algorithm to that of Linux 2.6.15 + Ingo Molnar's real-time patch. The experiments are performed for three representative kernel threads, *IRQ thread*, *ksoftirqd*, and *pdflush*, on a 2GHz Intel Pentium 4 machine with 256MB of RAM. In each case, the weighted value is set to 1.2 and 0.8 respectively.

6.1 *IRQ thread* and *ksoftirqd*

To show the problem of static priorities of *IRQ thread* and kosftirqd, we measure the response time of a real-time task writing data to a file when another real-time task with lower priority reads data from another file. The first task writes 16 bytes in each period of 250 usec while the second task reads 100KB continuously. The two real-time tasks are related with the same *IRQ thread* because they use the same disk.
As shown in Figure 2, the response time on the existing Linux with real-time preemption patch is larger than that on our implementation because of the problem explained in section 3.1. As a result, it is proven that the problem can be solved by *weighted average PIP* algorithm.

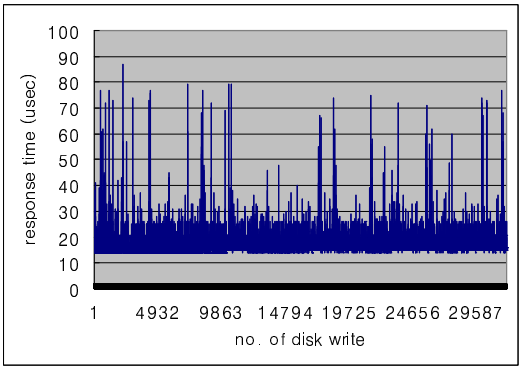


FIGURE 2: *Response time of disk write task on Linux with RT-Preempt patch*

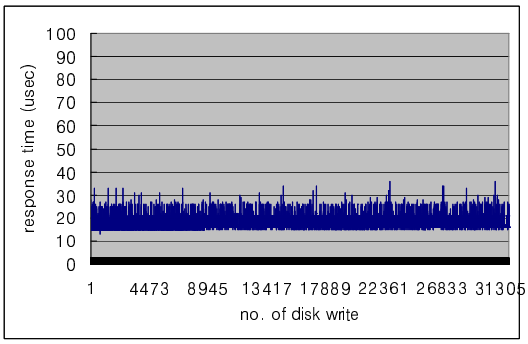


FIGURE 3: *Response time of disk write task on Linux with RT-Preempt patch and weighted average PIP*

6.2 *pdflush*

To show the problem caused by starvation of *pdflush*, we measure the response time of a real-time task writing of 800KB to a file every 200msec while a cpu-bound real-time task runs with lower priority. The lower priority cpu-bound task disturbs the execution of *pdflush*. In addition to the response time, the dirty page ratio of the system is also measured. Figure 4 and Figure 5 show the response times of the two cases respectively and Figure 6 shows how their dirty ratios change. The response time in the case of existing Linux with real-time preemption patch increases from the time when the dirty ratio reaches the threshold and the real-time task writes back dirty pages by itself. In contrast, in the case of our implementation, the response time is stable since the dirty ratio is controlled below the threshold.

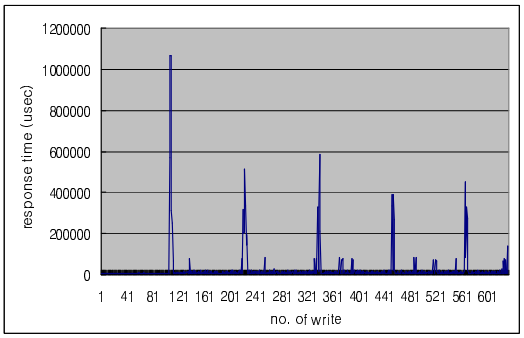


FIGURE 4: *Response time of disk write task executed with a cpu-bound task on Linux with RT-Preempt patch*

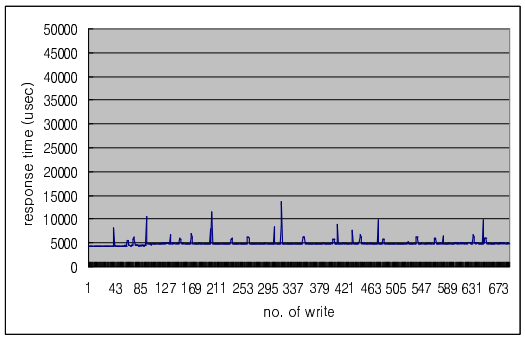


FIGURE 5: *Response time of disk write task executed with a cpu-bound task on Linux with RT-Preempt patch and weighted average PIP*

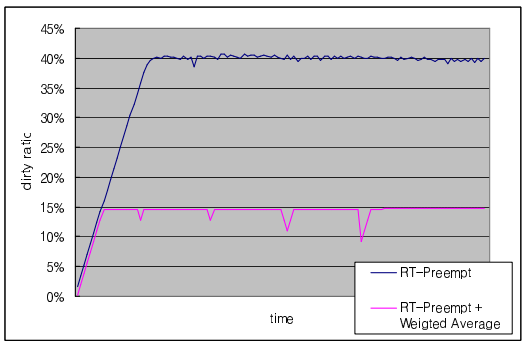


FIGURE 6: *Dirty ratio with and without applying weighted average PIP to pdflush*

7 Conclusion

This paper, we have proposed a new scheduling algorithm for kernel threads using *weighted average PIP* mechanism. The proposed method dynamically adjusts the priorities of kernel threads by monitoring activities of related real-time tasks. The priority computation is based on two factors: the average priority of real-time tasks related to the kernel thread and the weight value of each kernel thread representing its own characteristics. We have shown by experiments that, in the case of three kernel threads such as *IRQ thread*, *ksoftirqd*, and *pdflush*, the response time of real-time tasks greatly reduced when compared with current Linux system. There are many other kernel threads such as *kswapd*, worker threads, *kjournald*, etc. The future works include how these kernel threads are dynamically scheduled in our proposed scheduling algorithm.

References

[1] Ingo Molnar real-time preempt patch, http://people.redhat.com/mingo/realtime-preempt/

[2] Sven-Thorsten Dietrich, Daniel Walker, 2005, The Evolution of Real-Time Linux, Seventh Real-Time Linux Workshop

[3] Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz, 2006, Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware, Proceedings of the Twelfth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS06)

[4] Robert Love, 2005, Linux Kernel Development Second Edition, Novel Press, ISBN 0-672-32720-1

[5] Daniel P. Bovet, Marco cesati, 2005, Understanding the Linux Kernel, O'REILLY, ISBN 0-596-00565-2

[6] CE Linux Forum RealtimePreemption, http://tree.celinuxforum.org/CelfPubWiki/RealtimePreemption

[7] Patch: PREEMPT_RT and I-PIPE: the numbers, part 4, http://lwn.net/Articles/143414/

[8] REAL-TIME LINUX BENCHMARKS, http://www.mvista.com/products/realtime_benchmarks.html

[9] John Mehaffey, 2004, WHITE PAPER: MontaVista Linux Open Source Real Time Project, MontaVista Software

[10] L. Sha, R. Rajkumar, and J-P. Lehoczky, 1990, Priority inheritance protocols: An approach to real-time synchronization, In IEEE Transactions on Computers, pp1175-1185

A Pre-build Real-Time Scheduling Algorithm for Embedded System

Chen Tianzhou, Xie Bin and Hu Wei
College of Computer Science, Zhejiang University
Hangzhou, Zhejiang, China, 310027
{tzchen,xiebin,ehu}@zju.edu.cn

Abstract

This paper proposes a scheduling algorithm specially for the embedded system. To meet the requirement of the real-time system, we divide tasks into group based on their resource requirements. In this way, the switch can be done more effectively. The algorithm is static and pre-build because embedded systems have some remarkable features. Real time tasks and interrupts are important, so the algorithm takes them as a special part.

1 Introduction

Nowadays embedded systems play a more and more important part in people's everyday life. The hardware has become more powerful, and the tasks have become more complex. So embedded operating systems are used far more wildly then before.

Among all the embedded systems, real-time is a key feature. Almost all the system wants to finish works as soon as possible. The scheduling algorithm has been an important research field in the operating system for a long time, and there are many kinds of scheduling algorithm. But hard real-time operating system is really hard to design. One reason is that the system is complex, lots of kinds of tasks and resource. One operating system needs to manage all of these. So the performance of the general operating system cannot be improved greatly.

To provide embedded hard real-time operating systems, many techniques are proposed by researchers. Based on the traditional scheduling algorithms [1–3], many scheduling algorithms are proposed to solve the real-time problems. [4] presents an approach which is able to schedule based on an abstract graph representation. In this graph, both flow and the flow of control can be captured. But it is better for several processors and not very suitable for common embedded systems. Real-Time Petri Nets [5] are also used to synthesize real-time embedded software from a network. A branch-and-bound algorithm is presented in [6]. This algorithm performs an optimal pre-runtime schedule which is on a single processor for real-time process segments with release, deadlines, and arbitrary exclusion and precedence relations. But the performance is not evaluated in this paper and we can not compare this algorithm to the other algorithms. And readers can refer to more real-time scheduling algorithms in [7–12].

Embedded systems are special, so we can do a lot to modify these algorithm, to make them more powerful. To meet the requirement of embedded real-time systems, we simplify the classical scheduling algorithm based on limited tasks and resource. In the embedded operating system, the scheduling result can be small enough to store in a pre-build table. So all the scheduling structure can be changed a lot.

The remainder of this paper is organized as follows. Section 2 illustrates algorithm structure; section 3 describes the scheduling algorithm. Finally, section 4 provides conclusion and some future work.

2 Algorithm Structure

Commonly scheduling algorithm should be designed to look after both sides of throughout of this system and the response time. It is the core part of the entire embedded operating system. The algorithm proposed in this paper is designed to improve real-time of embedded operating systems. The algorithm procedure consists of three sub-procedure: Set division, scheduling inside one Set and scheduling among

Sets as shown in Figure 1.

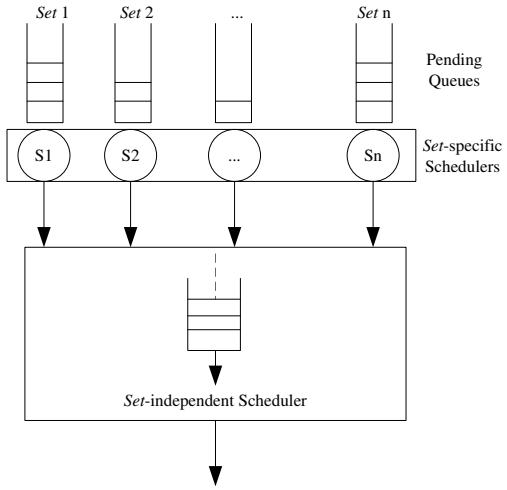


FIGURE 1: *Procedure of Scheduling*

2.1 Set Division

As the preparation for scheduling, our algorithm divides all tasks into different Sets according to the requirements for resource and real-time of tasks in order to manage the resources well. Specially, we design a Set named HRTSet for hard real-time tasks. System will reserve part of resources for HRTSet in order to ensure response time for the hard real-time tasks. Because to maintain the data structures for the Sets will seize a lot of system resources if the number of Sets is too big, and the efficiency will be low if the number of Sets is too small, our algorithm divides Sets in accordance with the most important resources of the system such as requirements for network, I/O, disk requests and so on. The common shared resources which are needed by every task will not be considered as the division criterion such as CPU. Each task will be signed a Set flag when the task is created.

2.2 Scheduling Inside One Set

Scheduling inside one Set means that the wellsuited task should be choose from one Set. In each Set, it will be selected a common scheduling algorithm to do so. Thus not all of the scheduling algorithms in Sets will be the same algorithm. Extremely, each Set perhaps selects a different algorithm from all other Sets. That each Set has its own scheduling algorithm will provide extra advantages: it can coordinate the requirements for resources in its Set and adjust scheduling according its own environment. For example, Round-Robin will be better to Network Set and FIFO is better to HRTSet. And if necessary, there is able to be more than one scheduling algorithms in one Set and it can use most suitable algorithm for the particular condition. But our algorithm assigns a simple and mature scheduling algorithm for each Set because complex algorithms will reduce the efficiency of the system.

2.3 Scheduling Among Sets

The inside set specific schedulers choose one task from each set. Now it is the time to schedule among them. Because the kind of tasks in a embedded system is limited. So we can enumerate all the possible case. For each case, we try to find the optimal scheduling result. Then we keep all of these results together. As the number of sets is small, these results won't be huge, and can be stored in a table. When the scheduler wants to choose one from different sets, it only needs to search the table, and gets the result. The system's performance can then be improved.

3 Algorithm Formulation

3.1 Notation and Problem Formulation

This section introduces the notation used throughout the article and gives the problem formulation.

Because all the tasks in the same set have nearly the same resource requirement, and the scheduling algorithm inside one set is simple. We take the among sets scheduling algorithm as the key part of the whole algorithm. So in this section when we refer tasks, that means the tasks came from different set.

3.1.1 Tasks and Jobs

Let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of N tasks. Each task consists of j^{th} jobs each of which is an instance of the task. Task $T_i = \{J_{i,j} | J \in j^{th}$ jobs of $T_i\}$ is a set of jobs. Each task T_i has a priority and this priority will be inherited by the jobs of T_i. These instances may be released either periodically or sporadically with a known minimal inter-arrival time. The period or minimal inter-arrival time of a task T_i is denoted as τ_i .

The basic scheduling entity we consider is the job abstraction. Thus, we use J to denote a job without being task specific, as seen by the scheduler at any scheduling event; J_k can be used to represent a job in the job scheduling queue. Jobs can be preempted at arbitrary times.

3.1.2 Resource

There are only finite resources in an embedded system. The types and amount of resources can be known as soon as the system is configured. And each resource is generally reusable and can be accessed by jobs. Resources can be shared and can be subject to mutual exclusion constraints. In our model, there may be multiple instances of a resource. Jobs should explicitly specify the resource that it wants to access. Thus resources are tetrad $R_i(t, c, b, n)$ in which t is the type of resource, c is the amount of resource type t, b is a busy link which consist of occupied resources and their owners and n is the amount of unoccupied resource type t. A job may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested, overlapped or disjoint. We assume that a job explicitly releases all granted resources before the end of its execution.

In our algorithm, we present the resources occupied by a given Job J_k, as a matrix as follows:

$$R_{J,k} = \begin{matrix} t_1 & r_1 & o_1 & i_1 \\ t_2 & r_2 & o_2 & i_2 \\ \dots & \dots & \dots & \dots \\ t_m & r_m & o_m & i_m \end{matrix}$$

t is the type of resource, r is the total requirements of type t from J_k, o is the obtained resources by J_k and i is the resources of type t still needed by J_k.

3.2 Scheduling Algorithm

In this section, we describe the scheduling algorithm presented in this paper.

3.2.1 Enumerated Scheduling Sequences

In embedded system, there are finite resources for tasks to use. And there are only finite tasks in embedded environments. For a set of jobs in a given embedded environment, the snapshot can be taken at time t. Thus we can get the resource status of time t including:

- The resources occupied by Jobs;
- The free resources.

The scheduling status of Jobs can be get from this snapshot too.

At any time we take the snapshot, the number of required resources and remainder resources is equal to the total number of resources in system.

Before the scheduling algorithm works, all the finite Jobs are enumerated and their requirements for resources are enumerated too as a matrix totally.

Theorem. The finite Jobs in a system can be enumerated and the scheduling sequence can be considered as a regular expression which can be computed by a finite automaton.

Proof. From our snapshot and the limited computation capacity of a embedded system, it can be concluded that there are only finite Jobs in the embedded system.

Let the set of Jobs $\sum = \{J_1, J_2, \dots, J_k\}$ is the alphabet. The language *LSA* based on $\sum$ is a regular language because of the following features:

1. *LSA* is based on $\sum \bigcup \{(,), \emptyset, \bigcup, *\}$ and the following string can be got;
2. $\emptyset$ and members of $\sum$ are regular expression;
3. if α and β are regular expression, $(\alpha\ \beta)$ is regular expression;
4. if α and β are regular expression, $(\alpha \bigcup \beta)$ is regular expression;
5. if α is regular expression, α^* is regular expression;
6. No other regular expression except for 2 to 5

Thus we can construct a finite automaton based on the enumerated Jobs from *LSA*.

LSA is the sequences of Jobs in $\sum$, and it is also the possible scheduling sequences of Jobs.

From theorem, all the scheduling sequences can be enumerated and thus we can continue the next step.

3.2.2 Scheduling Table

From the previous step, we get all the possible scheduling sequences. Nevertheless not all the sequences are usable and optimal for embedded system. One most optimal sequence should be selected from the *LSA*.

In our algorithm, there are two matrixes: R_i and $R_{J,k}$. To obtain the most optimal sequence, these two matrixes should be used to calculate.

Because there are penalties when Jobs have to be preempted from resources they occupy, an additional item P is added. $P_{J_k,R}$ means the penalty when we preempt resource R from J_k.

The basic computation of P is:

$$P = \sum_{k=0}^{n} P_{Jk,R} \tag{1}$$

P consists of two parts: P_{J_k}, this means the penalty on J_k; and $P_{\sum}$ which means the penalty on

all the following jobs in scheduling queue. Thus we get equation:

$$P = \sum_{k=0}^{n} (P_{Jk} + P_{\sum}) \tag{2}$$

Each penalty can be calculated by the preemption penalty which consists of context switch time. Assume there are n jobs in system, J_k is the kth job, and there still $n-k$ jobs in scheduling queue. And there are m types of resources. Then the P_{J_k} is:

$$P_{J_k} = \sum_{i=0}^{m} T_i \tag{3}$$

T_i means the penalty when R_i is preempted.

$$T_i = R_{i,c} \times P_{i,c} \tag{4}$$

$R_{i,c}$ is the possible preempted resource count of type i. $P_{i,c}$ is the penalty when $R_{i,c}$ is preempted.

The $P_{\sum}$ is:

$$P_{\sum} = \sum_{j=k+1}^{n} T_j \tag{5}$$

T_j means the penalty on the following jobs behind J_k when J_k is preempted.

$$T_j = \int_{j}^{n} R_{j,c} \times P_{j,c}\, dj \tag{6}$$

For each snapshot, there are certain jobs and all the status of system can be obtained. Thus we can select the scheduling sequence by calculating:

$$\min(\sum_{k=0}^{n}(\sum_{i=0}^{m} R_{i,c} \times P_{i,c} + \sum_{j=k+1}^{n} \int_{j}^{n} R_{j,c} \times P_{j,c}\, dj)) \tag{7}$$

of all the *LSA* sequences. The selected scheduling sequence is called *OSS* (Optimal Scheduling Sequence). Further, because there are finite Jobs in system, we will calculate different *OSSes* for different range of job number. An simple example is that we can calculate an *OSS* table for a embedded system as following: we can calculate an *OSS* when job number is less than ten, more than ten and less than twenty and so on. Thus an *OSS* table can be constructed.

3.2.3 Scheduling Algorithm

Our scheduling events include the arrival and completion of a job, a resource request, a resource release.

In the following we use the notation defined before. And new notation is used are explained as follows:

There are ε subsets in which there is fixed number of continuous θ jobs.

First we calculate the *OSS* table as shown in Algorithm 1.

Algorithm 1 : OSSOfflineComputing()
1 : input: ε, Job set Ψ, Resource set R;
2 : output: OSS;
3 : for every subset Ω ;
4 : $T_i = R_{i,c} \times P_{i,c}$;
5 : $P_{J_k} = \sum_{i=0}^{m} T_i$;
6 : $T_j = \int_{j}^{n} R_{j,c} \times P_{j,c}\, dj$;
7 : $P_{\sum} = \sum_{j=k+1}^{n} T_j$;
8 : $P = \sum_{k=0}^{n} (P_{J_k} + P_{\sum})$;
9 : $\mathrm{OSS}_{\Omega} = \min(\sum_{k=0}^{n}(\sum_{i=0}^{m} R_{i,c} \times P_{i,c} + \sum_{j=k+1}^{n} \int_{j}^{n} R_{j,c} \times P_{j,c}\, dj))$;
10 : OSSTable=$\bigcup \mathrm{OSS}_{\Omega}$

TABLE 1: *Algorithm 1*

The *OSS* table will be stored in memory of embedded system such as Flash on chip. When the system is startup, this table will be placed in main memory or SRAM.

A description of our algorithm at a high level of abstraction is shown in Algorithm 2.

Algorithm 2: OSS Scheduling Algorithm
1: input: Job set Φ ;
2: output: The selected Job J_k ;
3: $J_k = CheckOSSTable(\Phi)$;

TABLE 2: *Algorithm 2*

The $CheckOSSTable()$ is another function to search in OSSTable to find out the job need to be scheduled. The detail of $CheckOSSTable()$ is shown in the following table 3

As extension of this algorithm, the priority of the resources and jobs can be set according to the real using environments. And the *OSSTable* can be more simple to reduce the whole storage of *OSSTable*.

Algorithm 3: CheckOSSTable()
1: input: Job set Φ ;
2: output: The selected Job J_k ;
3: initialization: $s = size(\Phi)$;
//Get the number of Jobs in Φ
4: $R = Status(R)$;
//Get the status of Resource set R
5: $J_k = Compare(R, s, \Phi)$
//Get Jk by compared to the OSSTable in memory

TABLE 3: *Algorithm 3*

4 Conclusion and Future Work

In this paper, we presented a new hard real-time scheduling algorithm for embedded system. As the base of scheduling is the requirements for resources from tasks, all resources are divided into different types, and tasks relying on these resources belong to different Sets and to specific embedded applications. Because the resources and tasks in a system are able to be enumerated, the scheduling sequences can be calculated as preparation. Thus the system is only to search in pre-build table for scheduling when scheduling algorithm is needed. This algorithm needs more memory space for its large pre-build table to improve real-time response.

There are still much work to do in the future: how to improve the algorithm for power-efficient; how to predigest the pre-build table and how to enhance the performance of this algorithm. And it will be more efficient for embedded system.

References

[1] Butler W. Lampson, 1968, *A scheduling philosophy for multiprocessing systems*, Communications of the ACM, Volume 11, Issue 5, pp347–360.

[2] M.Ruschizka and R.S.Fabry, 1977, *A Unifying Approach to Scheduling*, Communication of the ACM, Volume 20, pp469–477.

[3] C. L. Liu and James W. Layland, 1973, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM (JACM), Volume 20, Issue 1, pp46–61.

[4] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli and P. Pop, 1998, *Scheduling of conditional process graphs for the synthesis of embedded systems*, Proceedings of the conference on Design, automation and test in Europe, pp132–139.

[5] Pao-Ann Hsiung and Cheng-Yi Lin, 2003, *Synthesis of real-time embedded software with local and global deadlines*, International Symposium on Systems Synthesis, Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp114–119.

[6] J. Xu and D. Parnas, 1990, *Scheduling processes with release times, deadlines, precedence, and exclusion relations*, IEEE Trans. Soft. Engineering, 16(3), pp360–369.

[7] S. K. Baruah, R. Howell, and L. Rosier, 1990, *Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor*. Real-Time Systems, 2, pp301–324.

[8] E. Bini and G. C. Buttazzo, 2004, *Schedulability analysis of periodic fixed priority systems*, IEEE Transactions on Computers, 53(11), pp1462–1473.

[9] I. Shin and I. Lee, 2003, *Periodic resource model for compositional real-time guarantees*, Proc. of the 24th RTSS.

[10] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg, 2003, *FAST: Frequency-aware static timing analysis*, Proc. of the 24th RTSS.

[11] G. Lipari and E. Bini, 2003, *Resource partitioning among realtime applications*, Proc. of the 15th Euromicro Conference on Real-Time Systems.

[12] J. P. Lehoczky, L. Sha, and Y. Ding, 1989, *The rate-monotonic scheduling algorithm: Exact characterization and average case behavior*, Proc. of the RTSS.

A New Dynamic Frequency Scaling Algorithm for Power-Aware And Real-Time System

Chen Tianzhou, Qian Jie, Shi Qingsong, and Huang Jiangwei
Zhejiang University
Intel-ZJU Embedded Technology Center, College of Computer Science And Technology
tzchen@zju.edu.cn, qjztc@sohu.com, {zjsqs,hjw}@zju.edu.cn

Abstract

Scalability of the core frequency is a common feature of low-power processor architectures. Many methods for frequency scaling have been proposed to find the best trade-off between energy efficiency and computational performance. Embedded hardware monitors in the form of event counters have proven to offer valuable information in the field of performance analysis. We can get some clues from that, this will be helpful for us to find the target frequency.

In this paper, we introduce a new algorithm to obtain the target frequency which benefit from the performance monitor counters. And once the system information changes, which means we can change the core frequency to perform at a lower power consumption, the core frequency will be changed through our method in the next time slice.

Experimental results, based on measurements on the platform with Intel pxa255 processor, show that significant energy savings are achieved with little performance degradation. With some good situations the energy savings can go up to about 10% (with little performance loss). The proposed algorithm is effective in the power-aware and real time system.

1 Introduction

Energy is the basic foundation of every movement in the world. Without energy, nothing can go well. This basic principle is still holding true in the field of computer. Nonetheless the measurement, both accounting and management of energy have been widely unattended in the field of systems research. But luckily, today, the problem of energy has become a problem which all the society are concerned. Every device providers hope that can advise the devices that has much more powerful while the consumed energy much lower. In this modern society, the energy has become more and more demanded, the meaning of reduce energy consuming is significant.

In the past few years, the minimum of the energy consumed by a computer is become a concernful point of the computer design. Because the energy consumed of the CMOS circuits depend more on the supplied voltage, lower voltage is one of the most effective way to save the energy. Both the DVFS which can dynamic change the supplied voltage and the DVFS which can dynamic change the frequency of the core CPU are concerned much in these years.

Though the circuit technology has long development, but the energy consumed is still very huge, besides, if the energy consumed by a device is high, it is much easier for it to become hotter, this problem need some way to resolve it, also the stability of that device will be worse. And another point, which is the most important, user can get more convenience if the energy consumed become lower, because they can spend less time to charge the batteries.

There exists a large body of work on both dynamic and static voltage scaling [1, 2, 3, 4]. Many of the ideas developed by Weiser et al. [4] and Govil et al. [2] form the basis of these algorithms: that the CPU idle (slack) time should be minimized by slowing the CPU core frequency [6, 7, 8, 9]. We will discuss questions all concerned about how to save energy and use it more effective. We can see from [10], the power consumption of CPU is still the largest component of one system. That's why we need to use DVFS to scaling down the frequency of CPU which can reduce the power consumption.

In this paper we focus on the energy-critical hardware component: the CPU, the use of this component can already be monitored by the performance

monitoring counters found in many contemporary processor architectures. We will discuss how to find a good frequency for your applications. This paper is organized as follows: Section 2 introduces related work and their disadvantages in detail. In Section 3, we provide a detailed description of our framework and describe our experimental result. We close with conclusions of this paper.

2 Related Work

Circuit-level techniques have been a mainstay of power reduction for some years, but recently much research attention has focused on system-level techniques. The benefits of approaching power reduction at the system-level are typically synergistic with circuit-level techniques, and higher design abstraction levels have more direct knowledge about the workload and can control large portions of the computer system accordingly. Yet for RT systems, performance should still be guaranteed even when power reduction techniques are in place. There are two well studied power reduction techniques that have impacts on system scheduling [11]: DVS (dynamic voltage scaling) and DPM (dynamic power management). In DVS, different computational tasks are run at different voltages and clock frequencies while still providing an adequate level of performance, we can also call this DVS and DFS separately. DPM aims to shut off system parts (inside or outside the CPU) that are not in use at any given time. Dynamic voltage scaling (DVS) has been proposed and widely deployed for microprocessors [12,13], uncovering significant power savings [13]. DVS in microprocessors exploit the variance in processor utilization, lowering the frequency (and voltage) when the processor is lightly loaded, and running at maximum frequency (and voltage) when the processor is heavily executing. As we know, the power consumption (also we can think it as the energy consumption) of a computer system is due to the equation:

$$p\_total = c * f * v^2 + p\_other \qquad (1)$$

where p_total denotes the total power consumption of the processor, while f is its operating frequency and v the supply voltage. c and p_other are architecture-dependent constants. According to the equation, the total power consumption is essentially the product of the operating frequency and the square of the supply voltage. If we can reduce the frequency f and the voltage v, then the total power consumption can reduce to a lower value.

In order to utilize DVFS, the CPU scheduler must cooperate closely with the DVFS algorithm. The two main questions that need to be answered by each DVFS approach are: (a) when to change the frequency and voltage and (b) how to change it. Somebody concern this problem combining tasks' scheduling attributes, utilizing all tasks' scheduling attributes, the DVS algorithm can compute the total CPU utilization, and consequently, the frequency and voltage combination that maximizes this utilization (i.e., 100%). However, inaccuracies in predicting tasks' run-times can cause system utilization to exceed 100%, thereby leading to missed deadlines and frequent frequency and voltage re-computations. Overestimating the utilization can result in inefficient energy management. One of the reasons for these inaccuracies is the frequent memory accesses of memory-bound applications. Even modern processors are able to hide these memory access latencies only partially. We therefore address this issue by monitoring the cache miss rates of all tasks to obtain an estimate of the memory latencies and to improve the frequency and voltage computations of our DVFS approach. There has been substantial prior work on energy management for mobile multimedia systems, including low-power modes for disks and networks [14, 15], energy-aware scheduling policies [16, 17, 18, 19], and energy management techniques for wireless communication [20, 21, 22, 23, 24]. The main contribution of this work is the implementation and evaluation of a feedback-based DVFS approach for real-time systems that adjusts predictions of future system utilizations based on monitored memory accesses. The resulting approach to DVFS can lead to higher energy savings, reduced deadline misses, or a reduced number of re-computations of frequencies and voltages. Our experimental results utilize the voltage and frequency scaling capabilities of an XScale-based evaluation board. In comparison, previous work on DVFS algorithms for real-time schedulers described in [25] and [16] relies on worst-case task execution times, whereas we use cache feedback data to approximate actual execution times. There has been substantial prior work that examines the effects of DVFS on cache performance. In [26] present an energy-aware scheduling policy for non-real-time operating systems. In our algorithm, we concern the frequency scaling during the run mode, as the time when system is running is very long. In our approach, CPU receives information through performance monitors in every period time of T, the kernel will change the frequency according to the given information. So we can sure that the frequency can be changed within a time slice. We will show this in the following sections.

3 The Implement of Our Algorithm

3.1 Architecture

Our approach to find the target frequency is base on the event-counters. For a specific architecture we have to find a set of countable events that characterize the behavior of a system concerning performance and energy consumption. The rates at which these events happen at a certain time can describe the information of system, and through that, we can find the proper clock frequency that minimizes the energy consumption for a given performance requirement. The new frequency is determined by a periodic evaluation of the event rates in the latest history of the system. Therefore the scheduler has to find the frequency domain that matches all the event rates of the system.

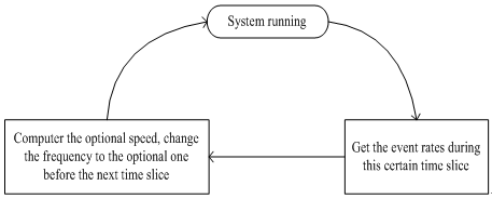


FIGURE 1: *The data flow relation in our system*

The data flow in figure 1 shows the relation in our system. The running system get the event rates during this certain time slice. And an optimal speed is made and applied to the application in the next time slice. Again, the new CPU speed is fed into the model along with newly measured event rates. To summarize, our approach can be outlined as follows:

1. Get the rates of different events during certain time.

2. Compute the lowest possible clock frequency (the optimal speed) for a certain combination of event rates and performance degradation.

3. Scale the clock frequency according to the precomputed optimal speed. Now the problem is how to get the optional frequency.

3.2 How to get the optional frequency

The hardware platform we use is an Intel development board EMH255. The PXA255 processor we use has a maximum clock frequency of 400MHz, two 32K caches (32K instruction cache, 32K data cache, 64M SDRAM, 32M boot ROM, 32M flash memory). This processor can run under several frequencies which determine by the register CCCR. We just take consider of the frequency 200, 300 and 400 in our algorithm. Now the problem is how to find the target frequency. If we can know the suitable running frequency for every application first, then we can solve these questions easily, but in fact it is difficult. You may get correlative information from running it forward, but I don't think this is a good way. This means you need to use more time to get that information and estimate the best frequency. So we need to find another way which can find a good frequency while taking not so muck time.

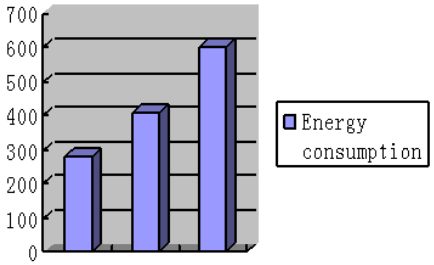


FIGURE 2: *Energy consumption in three different states*

Firstly, we think about when to reduce the frequency. During all tests the processor was operated at 300MHz. Without involving the main memory and caches the power consumption of the pxa255 processor is nearly at about 280mW. But the power consumption increases to about 410mW when caches and MMU are utilized. Further tests also involve the SD-RAM module and memory controller by reading from and writing to the main memory. The power consumption now rises up to 600mW. We can see this from figure 2. We can believe that this fact results from activating the memory controller and the memory module and using the memory-processor bus[8].

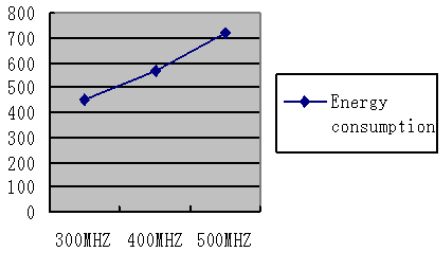


FIGURE 3: *Energy consumption at different frequency when working with memory*

Then, when we do tests which involve the SD-RAM module and memory controller by reading from and writing to the main memory at different frequency, we can see an interesting result as shown in figure 3. But the performance is not speed up clearly. This figure displays one important fact: applications intensive involving the main memory do not significantly increase their application performance when CPU is sped up. Of course, this fact results from the

constant memory access rate. Because if the memory access rate can change with the CPU speed, then the environment will be changed and there will be no benchmarks.

Then when we can say now the applications is intensive involving the main memory. We think the number of executed instructions is a good choice. If an application spends much time on exchange data with main memory or other functions, the number of executed instructions in a fixed time slice must be reduced. So we can say: if the number of executed instructions in a fixed time slice is much lower than the expected number at this frequency, then we can think we need to reduce the frequency. Characteristics for resource usage of processes can be obtained by performance monitoring counters. Most modern processors supply performance monitoring units (PMU) counting special events that occur in the CPU like cache misses or executed instructions. The Intel Xscale core can monitor either occurrence events or duration events. When counting occurrence events, a counter is incremented each time a specified event takes place and when measuring duration, a counter counts the number of processor clocks while a specified condition is true. If any of the counters overflow, an interrupt request will occur if it's enabled. Each counter has its own interrupt request enable. The counters continue to monitor events even after overflow occurs, until disabled by software [13]. Each of these counters can be programmed to monitor any one of various events. Just take the pxa255 at XSC2 for example. There are four performance monitoring counters PMN0 to PMN3. We can use them to monitor events that have been defined, such as instruction cache miss requires fetch form external, data TLB miss, instruction executed and so on. If a special interrupt happen, you can define what you want to do and let the system go to execute your defined function. Also, you can use a timer to achieve this. Besides, we need to take care of the performance loss, we think the cache efficiency can describe the system performance, because the system is much busier when the cache efficiency is higher than the system when the cache efficiency is lower. And we assume there is a linear relation between the cache efficiency and the system performance.

We need to make a table where we can find a good frequency under a close number of executed instructions. This means we need to build a referenced table first. With this table, we select information which are need and compare with the referenced value, getting the target information, and then we set our CPU frequency to the optional value. Of course we need to think about the executing time. We denote a relative number which is called experimental value, and we use this value to control the executing time. Because the time that spend on tasks is in inverse proportion to the executed instructions in certain time slice. Also, we use the performance loss value to make sure the system performance loss is not high. This is an example, we have three training example of the form (frequency, instructions executed as the fixed time slot, data cache efficiency): A: (200, 1200, 80%) B: (300, 1800, 85%) C: (400, 2400, 86%) D: (200, 800, 60%) E: (300, 1200, 40%) F: (400, 1500, 45%), we can computer the data cache efficiency as follows: we can use a pmn0 (performance monitor unit 0) to count the number of instruction executed, and another one pmn1 to count the number of instruction cache miss requests, and then we can get the instruction cache miss rate as pmn0/pmn1. Then if now the number we get from the monitoring counter is 1100, the data cache efficiency is 90%, the frequency is 400, then we may choose 200 as our target frequency at the next time slot. Because absolute value of data cache efficiency minus 80% is less than 12%, and the number of executed instructions is 1100, $|1200 - 1100| < 300$, then we can think this frequency 200MHZ is enough which means we can finish the tasks of the system and save more energy also. We register an IRQ interrupt, when the clock counts overflow, the CPU will run our function to read the counts which are used to count the number of executed instructions. And according to the information we get from the performance monitor counters, we compute the optional frequency. Such as in the above example, we will set the frequency of pxa255 to the optional value 200MHZ.

We can summarize as follows (in our system, we set the experimental value to 300, and set the performance loss value to 12%):

```
Get the information in this time slot: the number
of executed instructions N1, now frequency F1,
date cache efficiency is P1;
Compare with the training examples;
Assume we get one (F2, N2, P2);
While (we still have new training example);
If (|N1 - N2| < experienced value);
If (|P1 - P2| < performance loss value);
Change it with the expected value Set relative
registers;
Go to next time slice;
```

4 Experimental Result And Discussion

We test our algorithm at several different situations, and the schedules of tasks are shown as below (figure 4, figure 5, and figure 6). In situation a, the tasks on scheduler are sustained, they work almost under the

same workload, which means the degree of intensive involving the main memory is almost at the same level. In situation b and c, the tasks on scheduler are obviously different, but the interval in b is much longer than that in c, which means the degree of intensive involving the main memory changes rapidly. The power consumption of these three different situations is shown in figure 7. The left colonnades denote the power consumption when tasks are executed on the system without our algorithm, and the right ones denote the power consumption when tasks are executed on the system within our algorithm.

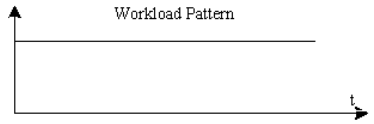


FIGURE 4: *the degree of intensive involving the main memory is almost at the same level*

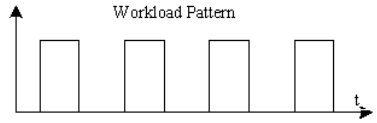


FIGURE 5: *the degree of intensive involving the main memory changes soon*

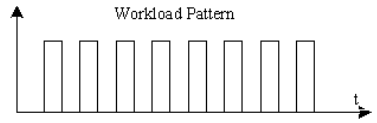


FIGURE 6: *the degree of intensive involving the main memory changes rapidly*

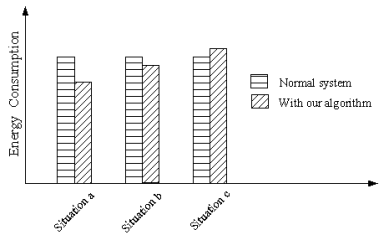


FIGURE 7: *the power consumption of different situations*

We can find that within our algorithm, when the degree of intensive involving the main memory is almost at the same level during certain time, the result is much better, and we can reduce the energy consumption much more. That's because we get the target frequency based on the former system running information, and if the state of system changes rapidly in the next time slice, the result would not very suit the changed system. The result also has relation with many other aspects, such as the measuring device, different temperature, the situation of the system, especially the data capability.

5 Conclusion

We want to represent a feasible way to get an good frequency to achieve the aim that saving more energy. Before finding any methods a target system had to be selected upon requirements for this approach: The system had to provide performance monitoring capabilities as well as functions to scale processor core clock. Then, values which are collected from the performance monitoring counters of the processor were investigated for indicating the state of memory accesses. Then we get the target frequency form the table we defined forward. Also we need to make sure that the response time is limited.

The next work we need to do is make a date library with more training examples. Also, our future work will extend this approach to I/O-bound applications (memory and disk accesses), thereby addressing the needs of energy-aware and real time communication-intensive applications.

References

[1] K. Flautner, S. K. Reinhardt, and T. N. Mudge. 2001, Automatic performance setting for dynamic voltage scaling. In Mobile Computing and Networking, pages 260-271, 2001.

[2] K. Govil, E. Chan, and H.Wasserman. 1995, Comparing algorithm for dynamic speed-setting of a low-power CPU. In Mobile Computing and Networking, pages 13-25.

[3] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, and M. Neufeld. 2000. Policies for dynamic clock scheduling. In Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI), pages 73-86.

[4] M.Weiser, B.Welch, A. J. Demers, and S. Shenker. 1994. Scheduling for reduced CPU energy. In Operating Systems Design and Implementation, pages 13-23.

[5] Andreas Mull, July 2002, Optimizing energy-consumption by event-driven frequency scaling. Master's thesis.

[6] Intel. Performance profiling techniques on Intel Xscale micro architecture processors, 2002. Application Note.

[7] Weiser, B.Welch, A. Demers, and S. Shenker, 1994, Scheduling for reduced cpu energy. OSDI.

[8] P. Pillai and K. Shin, 2001, Real-time dynamic voltage scaling for low-power embedded operating systems. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01).

[9] A. Weissel and F.Bellosa, 2002, 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.

[10] X. Feng, Rong Ge, Kirk Cameron, 2005, Power and Energy Profiling of Scientific Applications on Distributed Systems, presented at 19th International Parallel and Distributed Processing Symposium (IPDPS 05), Denver, CO.

[11] N. K. Jha, 2001, Low power system scheduling and synthesis. In IEEE Int. Conf. on Computer-Aided Design

[12] T. Burd and R. Brodersen, 2000, Design issues for dynamic voltage scaling In Proc. International Symposium on Low Power Electronics and Design, pages 9-14.

[13] Intel XScale microarchitecture. http://developer.intel.com/design/ intelxscale/

[14] D. Helmbold, D. Long, and B. Sherrod, 1996, A Dynamic Disk Spin-down Technique for Mobile Computing. In Proc. Of the Intl. Conference on Mobile Computing and Networking.

[15] S.Chandra and A. Vahdat. 2002, Application-Specific Network Management for Energy-Aware Streaming of Popular Multimedia Formats. In Proc. of the USENIX Annual Technical Conference.

[16] S. Saewong and R. Rajkumar, 2003, Practical Voltage-Scaling for Fixed-Priority RT-Systems. In Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).

[17] P.Mejia-Alvarez, E. Levner, and D. Mosse, 2002, Power-Optimized Scheduling Server for Real-Time Tasks. In Proc of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium.

[18] W. Kim, D. Shin, H. Yun, J. Kim, and S. Min, 2002, Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. In Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium.

[19] C. Poellabauer and K. Schwan, 2002, Power-Aware Video Decoding using Real-Time Event Handlers. In Proc. of the 5, International Workshop on Wireless Mobile Multimedia, Atlanta, GA.

[20] S. Agrawal and S. Singh, 2001, An Experimental Study of TCP's Energy Consumption over a Wireless Link. In Proc. of the 4th European Personal Mobile Communications Conference.

[21] J. Monks, V. Bharghavan, and W. Hwu, 2001, A Power Controlled Multiple Access Protocol for Wireless Packet Networks. In Proc. of IEEE Infocom.

[22] C. Poellabauer and K. Schwan, 2004, Energy-Aware Media Transcoding in Wireless Systems. In Proc. of the Second IEEE Intl.Conference on Pervasive Computing and Communications (PerCom 2004).

[23] C. Poellabauer and K. Schwan, 2004, Energy-Aware Traffic Shaping for Wireless Real-Time Applications. In Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium.

[24] D. Qiao, S. Choi, A.Jain, and K. Shin. MiSer, 2003, An Optimal Low-Energy Transmission Strategy for IEEE 801.11a/h. In Proc. of the ACM/IEEE Intl. Conference on Mobile Computing and Networking.

[25] P.Pillai and K. Shin, 2001, Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In Proc. Of the 18th ACM Symposium on Operating Systems Principles.

[26] Andreas Weissel, Frank Bellosa. 2002, Process Cruise Control Event-Driven Clock Scaling for Dynamic Power Management. CASES 2002, Grenoble, France.

Worst Case Behavior of CPU Caches

Tobias John and Robert Baumgartl
Chemnitz University of Technology
09107 Chemnitz, Germany
{tobias.john,robert.baumgartl}@cs.tu-chemnitz.de

Abstract

CPU caches reduce main memory access times and thereby speed up execution timing and significantly improve performance. Various cache architectures with different strategies and mechanisms have been developed. However, caches have a negative impact on execution timing predictability which is crucial in real-time systems. A precise understanding of available cache architectures is therefore essential. Although IA32 processors share a common instruction set, their CPU cache architectures differ significantly. Therefore, we describe in this paper a methodology for the construction of realistic worst cases for CPU caching. We compare and evaluate several IA32 processors with respect to efficiency and predictability of execution timing. Our methodology incorporates RTAI and the usage of Performance Monitoring Registers and generates very precise results in comparison to indirectly measuring execution times by counting clock cycles. Nonetheless our micro benchmarks are easily extendable and adaptable.

1 Introduction

For satisfying complex and compute-intensive tasks, real-time systems must more and more rely on features of modern processor architectures. By analyzing microprocessor evolution it is obvious that processors used in today's state-of-the-art personal computers can soon be expected to make their way into future embedded systems. As an example, memory management units (MMU) and caches have been introduced into real-time systems. Unfortunately, these features have a tendency to complicate the estimation of precise worst-case execution times, which is essential for real-time systems. Therefore, it is necessary to analyze and understand real-time behavior of modern processor architectures. Although processor caches in general have been the focus of several real-time research projects, this is not the case for the caches of the Intel IA32 architecture, because details of their inner workings have not been published. Therefore, this paper describes how to construct realistic worst cases of cache access operations for this specific processor architecture. We present a measurement methodology for obtaining very accurate access time measurements and apply it to different IA32 processors.

The remainder of this paper is structured as follows: Section 2 describes basic caching principles and defines our terminology. Some specific worst cases for different cache configurations are discussed in section 3. The following section 4 describes how these worst cases can be constructed and measured. The results we obtained during our experiments are presented in section 5. The paper closes with some conclusions and a short outlook onto future work.

2 Caching Principles

This section gives a short introduction into recent CPU cache architectures and thereby provides an overview of the terms and definitions used in this paper.

Data is indexed in caches by its address in main memory. Because caches are much smaller than main memory, only a part of the address is necessary as index. Hence, different main memory addresses point to the same cache location. In a *direct-mapped* cache, every cacheable datum can be stored at exactly one location within the cache. Usually, this leads to many replacement operations which lowers caching efficiency. Therefore, *n-way-set-associative* caches have been introduced which provide n potential locations for a datum to be cached (cf. figure 1). Now, the least significant part of the address indexes a *set* of the cache (a row in figure 1), in which a datum can be stored in different locations (the columns in figure 1).

The basic cache data unit is the *cache line* which comprises a block of bytes for efficiency reasons. We denote cache lines by uppercase letters and an individual datum within a line with a lowercase letter.

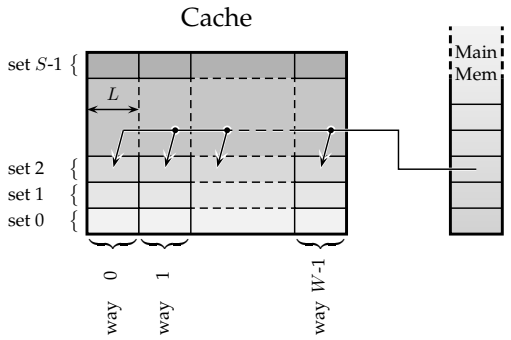


FIGURE 1: *Simplified Structure of a Cache*

Depending on whether a modified datum in the cache is written back immediately or when the cache line is replaced, two different cache write policies (*write-through* and *write-back*, respectively) are distinguished. If data in the cache is frequently modified, the latter usually performs better.

To further improve cache efficiency, cache hierarchies exist which consist of several caches between CPU and main memory. The nearer a cache is located to the CPU the smaller and the faster it is. We denote the cache level by *L1, L2, ...* If every line of Ln is also present in $Ln+1$ the hierarchy is called *inclusive*. On the other hand, if Ln and $Ln+1$ cache different data the hierarchy is *exclusive*. This paper solely focuses on two-level hierarchies but the principles can be applied to higher hierarchies as well.

3 Worst Case Configurations

3.1 Single Cache

Figure 2 illustrates the worst case for reading a direct-mapped cache with write-back policy.

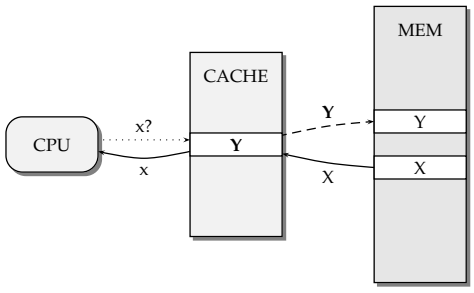


FIGURE 2: *Worst Case for Single Write Back Cache*

The CPU requests datum x which is not yet in the cache and therefore has to be fetched from main memory. If, however, the possible location where X can be stored in the cache is already occupied by line Y, then Y has to be written back to memory to free up space before X can be loaded. If the cache is 2-way-associative, both possible locations must be occupied (and one of them is written back).

A write-through policy is easier to understand. No data has to be written when a cache miss occurs, because cache contents and associated main memory is always consistent.

3.2 Inclusive Cache Hierarchy

The worst case for a cache hierarchy of two levels is more complex and worse than for a single cache.

Löser et al [1] describe a worst case for a two-level inclusive caching hierarchy where L1 uses a write-back strategy and is at least two-way-associative. Its structure is shown in figure 3 by means of a simplified one-way cache.

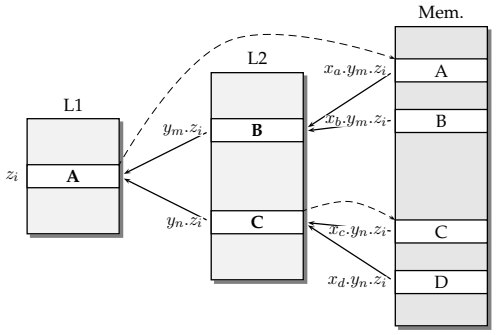


FIGURE 3: *Worst Case for Inclusive Two Level Hierarchy*

The CPU accesses d which is neither stored in L1 nor in L2 and has to be fetched from memory. Cache line D that holds d maps to entry z_i of L1 which is already occupied by line A. Therefore, for caching D in L1, line A has to be written to L2. Unfortunately, the modified line B occupies the corresponding entry in L2, hence A is written through to memory.

To load D into L2, entry $y_n.z_i$ is the only one that can be used. The modified line C has to be written back to memory to free the necessary space. Hence, a single read miss causes two write operations to main memory. This worst case is sometimes called *double-purge configuration.*

The characteristic property of this configuration is that although an inclusive caching architecture is used, it is possible to fill the cache levels in a way that data of L1 is *not* in L2. Both cache contents have

to be modified so that lines can not simply be overwritten and have to be written back to main memory before being overwritten. These conditions are met by the Intel Pentium PII and PIII processors but not by the P4. The latter features a write-through L1 cache which makes it impossible to store modified data in L1 only.

3.3 Exclusive Cache Hierarchy

The AMD Athlon64 uses an exclusive caching architecture. That means data is cached either in L1 *or* in L2. Uncached data can be loaded directly into L1 without being transferred first into L2. Hence, the worst case is accessing data that is not cached when L1 and L2 are already filled with modified ("dirty") lines. To load a new datum into L1 one cache line has to be moved into L2 to free up space. Because L2 is completely occupied with modified data, the line is written back to memory. A double-purge configuration does not exist. Interestingly, the Athlon64 provides another cache memory between L1 and L2, the so-called *victim buffer*, which stores a small number (e. g. eight) of lines that have been evicted from L1 to L2. If the victim buffer is not completely filled, it prevents data from being moved into L2 and lines from being written back from L2 into main memory. However, if the victim buffer is full, the next line evicted from L1 causes a full flush of the buffer. *All* victim buffer entries are transferred into L2. If no free or invalid lines are available in L2 an equal number of lines are written to main memory. That means, in the worst case, if the victim buffer can hold up to l_{VB} lines, the cache miss of datum x might result in one read from and l_{VB} write operations to main memory.

Obviously, different caching architectures with varying strategies and number of cache levels have different worst cases. Even for processors of the same architecture family the timing of the worst case memory access may significantly differ.

4 Constructing Worst Cases

4.1 Experimental Platform

The algorithm described in [1] to achieve a double-purge configuration is quite complex, can not easily be modified and has some limitations. Because our aim is to compare worst cases for different IA32 architectures we needed a more general and less complicated method wherefore we are accessing memory by consecutive addresses to fill the cache line by line. In doing so, address calculation and thereby the complexity is reduced to a minimum. This improvement makes the algorithms easy to understand and thereby well adaptable.

An ideal experimental platform for our purposes is the Linux Real-Time Application Interface (RTAI). By implementing our benchmarks as RTAI modules we are able to eliminate any timing interferences by user-space applications and the Linux kernel itself. Additionally, because RTAI modules reside in kernel space, no memory access restrictions exist which eases implementing arbitrary access patterns to manipulate cache contents. For drawing precise conclusions and for verifying the assumptions made on the underlying architecture we utilize the performance monitoring capabilities of modern IA32 microprocessors.

The discussion of several worst case configurations demonstrated clearly that a detailed knowledge of the underlying caching architecture is necessary and that beside structural parameters (e. g. cache size or associativity) functional behavior such as the replacement strategy or the caching policy need to be known in advance. Unfortunately, many of these parameters are not well-documented and often must be inferred from further experiments. This analysis has been done in a similar manner as the worst case experiments described here. The methodology basically consists of making guesses on properties of the cache to be analyzed and testing these guesses one after another by crafting a well-defined cache contents, performing specific access patterns and analyzing the resulting cache state. A discussion of these tests is beyond the scope of this paper. We refer the interested reader to [2] instead.

4.2 Implementation Details

The software to measure the worst case cache access offers the possibility to differentiate between three different cache configurations:

1. untouched,
2. "filled",
3. "flooded".

The first configuration leaves the caches untouched and therefore reflects the best case. The second configuration simply fills both cache levels with modified data, leaving the cache in a state that is likely to occur in everyday scenarios. The third configuration constructs the true, processor-specific, worst case. To quantify the influence of the branch prediction, all configurations allow to define whether the data references are performed in a loop or in a sequence without a branch. Furthermore, it is possible to execute a chunk of instructions prior to the test

to fill the instruction cache. Those instructions do not have any influence onto the measurement.

Figure 4 presents the structure of the benchmark, using assembly instructions common to the x86 architecture.

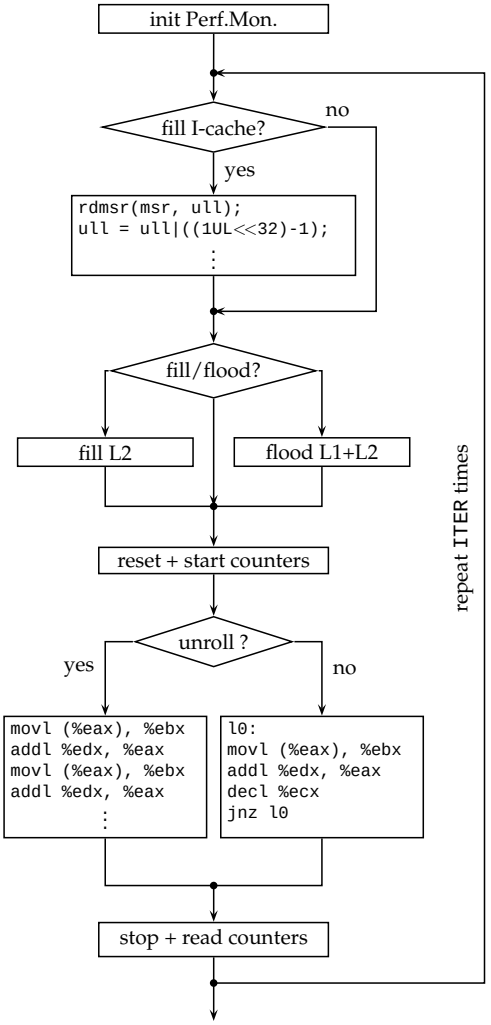


FIGURE 4: *Structure of the Benchmark*

The *"filled"* state of the caches is achieved by reading *and* writing as many cache lines as fit into L2 cache. The reading is necessary to allocate lines even if the cache uses a write-no-allocate policy and the writing modifies the data to mark the line "dirty".

The *"flooded"* state depends on the underlying architecture but is described on the basis of a two-level inclusive architecture as shown in figure 3. As pointed out before, the key is to have *different* modified data in L1 and L2. To this aim, L1 is filled once (cf. figure 5) and afterwards only one of its cache ways is used to load new data into L2. The other ways are continuously "touched" to keep their data in L1. (Therefore it is necessary to have an L1-associativity greater one.) This procedure is repeated until L2 is completely filled. The resulting configuration is shown in figure 6. The rightmost way of L1 has been used to load new data whilst the leftmost three ways still hold their original content. Until now the content of L1 is still available in L2, but if we continue to load new data into L2 by using only the *one* L1 cache way, the oldest lines in L2—those that we are continuously touching to keep them in L1—will be replaced. Finally, L1 and L2 contain different, modified data except the one L1-way used to fill L2 (cf. figure 7). But because this way is the most recent, it will be replaced last and until that point the other ways of L1 will have been replaced, leading to the described worst case. Figures 5 to 7 rest upon the Intel Pentium PII cache architecture.

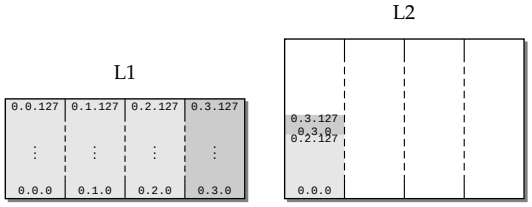


FIGURE 5: *Filled L1 Cache*

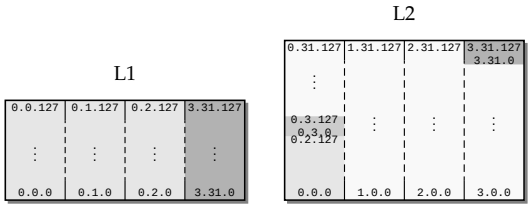


FIGURE 6: *Filled L2 Cache*

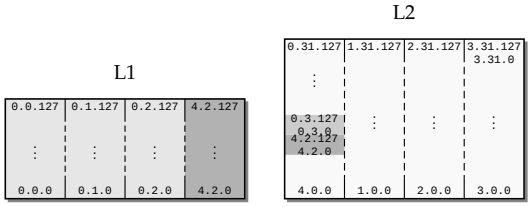


FIGURE 7: *"Flooded" State of both Caches*

5 Experimental Results

The described measurement benchmark was applied to different processors of the IA32 architecture. Tests were carried out with Intel Pentium II, Pentium III and Pentium 4. Because we still do not fully understand the AMD Athlon64 victim buffer we omitted it from the tests. For each processor the cache configurations "clean", "filled" and "flooded" (if available, cf. section 4.2) are compared in diagrams. Each cache configuration is further differentiated by loop or sequential execution ("unrolled") and whether the instruction cache has initially been filled ("dosth") or not.

5.1 Pentium II and III

Both, the Pentium II and III belong to P6 Family and share the same microarchitecture with the following characteristics:

- two-level inclusive cache hierarchy,
- write-back policy for L1 and L2 (configurable for L2),
- a non-random replacement strategy applied to L1,
- set-associativity,
- at least a two-way L1.

It is possible to construct a double-purge configuration (cf. figure 3). Both processors have a 4-way 16 KB L1 data cache and a 4-way 512 KB unified L2 cache. A cache line is 32 B long. L1 realizes a write-back policy and both caches follow a "pseudo Least Recently Used with tree-based fill (pLRUt)" replacement strategy. Therefore, the caches can be "flooded" as explained in the previous section. Figure 8 presents the results for the Pentium II.

cycles represents the average number of processor clock cycles read from the Time Stamp Counter register. The parameter mem-ref is the number of memory references by the CPU. This value should always be 100 because 100 cache lines were accessed during the program run. The most interesting parameter is the mem-trans value, which represents the number of memory transactions. Ideally, if the cache is "clean" , there should be no memory accesses because all data is found in the cache.

If the caches are "filled", then there should be two memory accesses per referenced date: one for the fetching of the requested datum and one for the evicted cache line. 100 cache lines were referenced and, as figure 8 shows, the memory transaction count in the "filled" diagram is always about 200. There are only minor differences between the diverse benchmark configurations "filled I-Cache (dosth)", "unrolled loops" and their combinations, therefore their complete discussion is omitted here. Details can be found in [2].

The diagram for "flooded" caches with its 300 memory transactions clearly shows that three fetches from memory were necessary to satisfy the CPU requests. Nevertheless, this worst case needs less processor cycles and therefore less time than the "filled" case. We do not have a explanation for that phenomenon so far. The PIII (cf. figurefig:piii) behaves identically to its predecessor except for the fact that accessing the "flooded" caches really takes longer than accessing the "filled" ones. Furthermore, this processor needs about twice as much clock cycles for the same operations. Comparing best and worst case, it can be seen that caches can speed up execution timing by about 27 times: almost 450 processor cycles are necessary if the data is completely cached, whereas about 12.500 cycles are needed in the worst case.

5.2 Pentium 4

The Pentium 4 as representative of the Netburst microarchitecture features a write-through first level cache which makes it impossible to have modified data in L1 only. Data is always written-through to L2 and therefore the worst case is an L2 completely filled with modified data. If a cache miss occurs, the datum is read from memory and can be stored in L1 without any effort because the content of L1 is either not modified or is modified but has been written-through to L2 and can be overwritten anyway. Only one of the modified lines in L2 has to be written back to main memory to free a line to store the new datum. This worst case situation is very similar to that depicted in figure 2. That means that "filled" caches already represent the worst case on the Pentium 4, flooding is not possible. Hence, figure 10 only consists of two diagrams: results obtained for clean and filled caches.

The Pentium 4 provides different performance monitoring events than its predecessors. That's why loads is used, representing the average number of memory loads and IOQ alloc as average number of read and write bus transactions. The latter parameter shows that two memory accesses per memory reference are necessary when "filled" caches are used.

Furthermore, figure 10 indicates that the P4 needs much more clock cycles to execute the measurement, although it has a cache line length twice as long as the PII/PIII and although the number of bits that can be transferred per clock cycle has been doubled, too.

5.3 Athlon64

The exclusive cache of the Athlon64 and its possibility to load data directly into L1 has a completely different worst case in cache access than the architectures already described. Under certain conditions, the flush of its victim buffer causes eight memory writes. This is really critical for real-time systems because the occurrence of these writes is hard ti predict. Theoretically, if the referenced data is not cached then for 8 out of 9 references only one memory access happens but every 9th time 9 memory accesses occur. The average is 2 memory accesses, but the best and worst case differ by 8 memory accesses. Due to

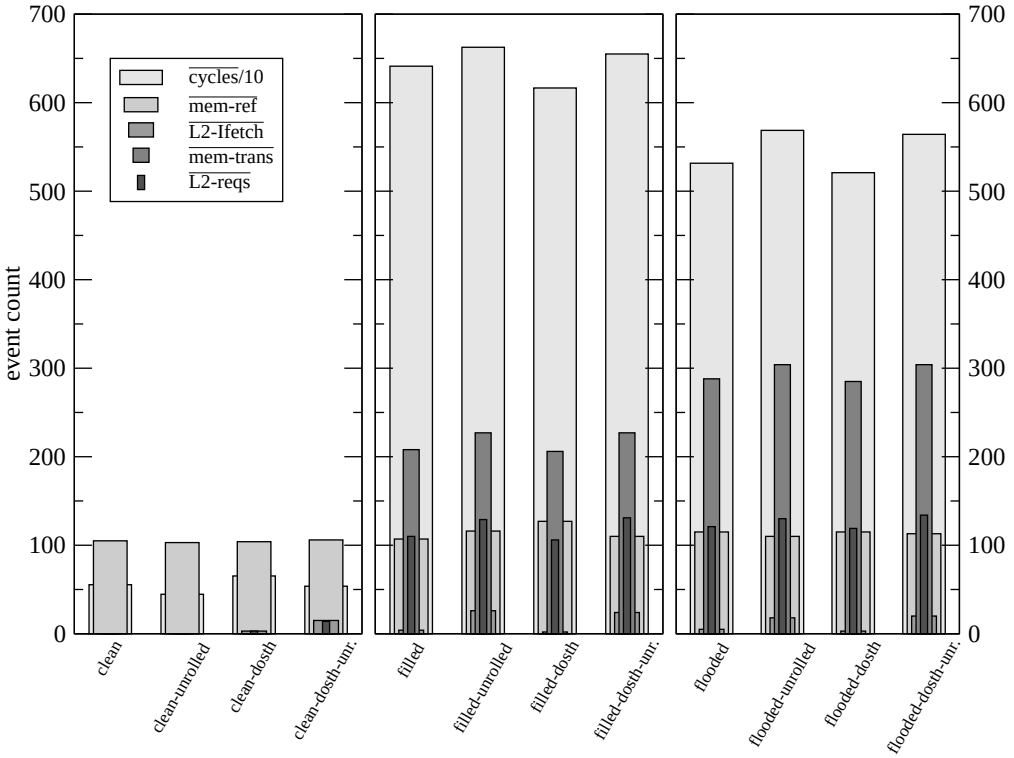


FIGURE 8: *Measurement Results for Intel Pentium II*

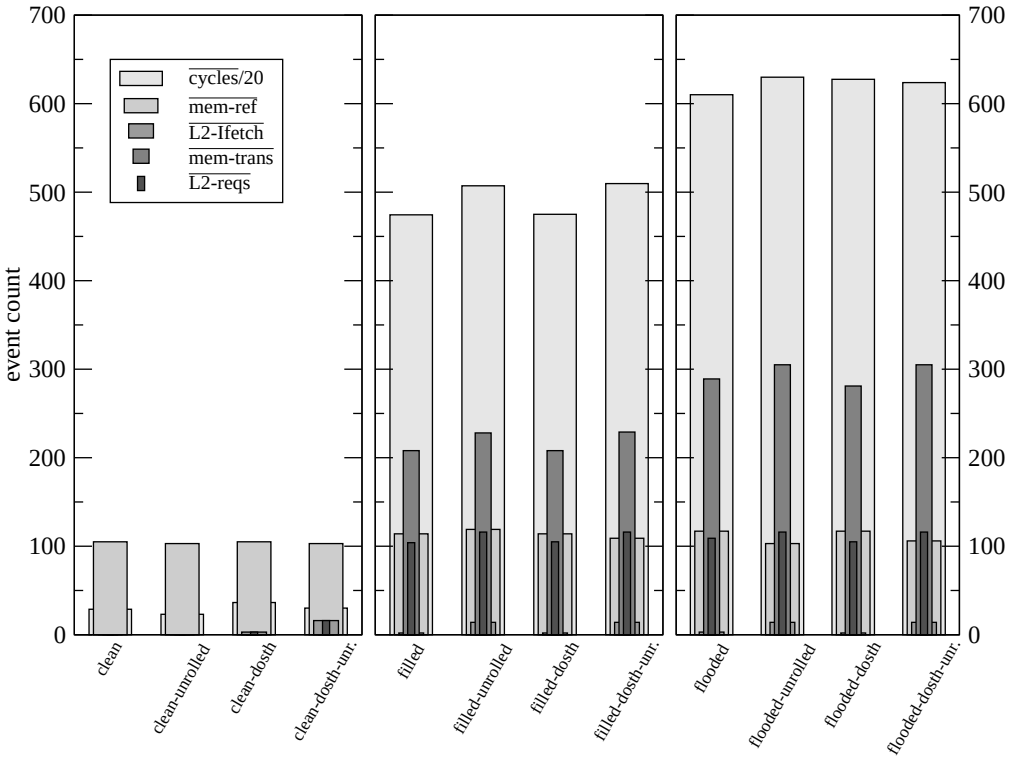


FIGURE 9: *Measurement Results for Intel Pentium III*

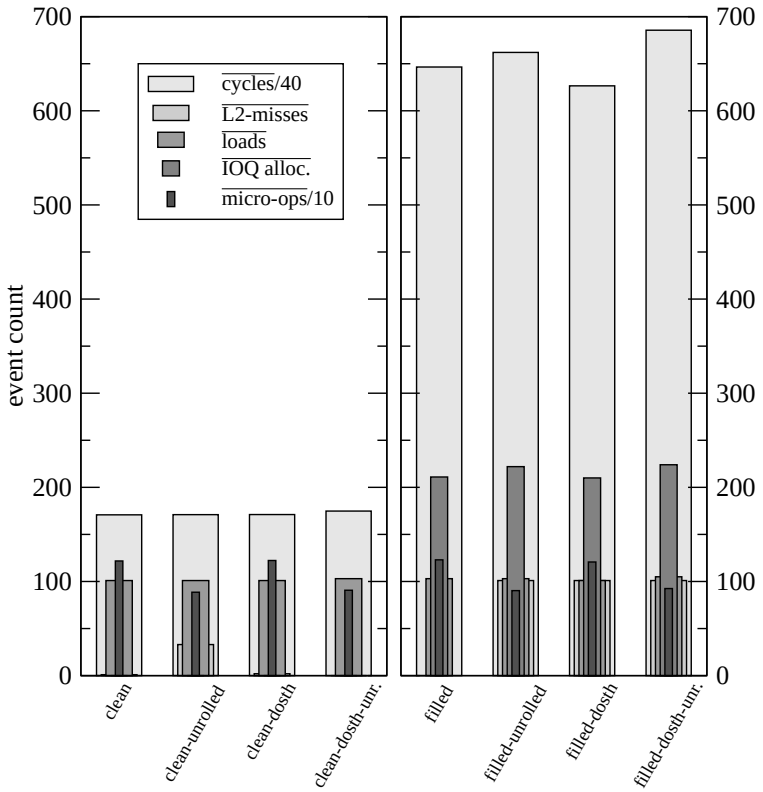


FIGURE 10: *Measurement Results for Intel Pentium 4*

poor documentation, we have not implemented special tests to provoke that worst case so far.

6 Conclusion and Future Work

We have illustrated that realistic worst case cache access times heavily depend on the underlying cache architecture. Even processors with seemingly similar architectures differ significantly with respect to cache behavior and access timing. We have implemented measurement benchmarks to semi-automatically analyze the worst cases for different configurations. To further verify our methodology and to establish a cache access time data base we plan to incorporate typical embedded and mobile processor architectures into our project. Furthermore, a thorough analysis of the victim buffer will be performed soon. Finally, our worst cases should be contrasted with real-world applications to quantify the gap between theoretical and realistic worst case scenarios.

References

[1] Jork Löser and Herman Härtig, 2002, *Cache Influence on Worst Case Execution Time of Network Stacks*, Technical Report TUD-FI02-07, Dresden University of Technology

[2] Tobias John, 2005, *Instruction Timing Analysis for Linux/x86 based Embedded and Desktop Systems*, Diploma Thesis, Chemnitz University of Technology,

[3] Intel Corporation, 2004, *IA32 Intel Architecture Software Developer's Manual. Volume 3*

[4] AMD, *Software Optimization Guide for AMD64 Processors*, 2005

[5] AMD, *BIOS and Kernel Developer's Guide for the AMD Athlon64 and AMD Opteron Processors*, 2006

Research on Real-Time Linux in Telecom Operations Supporting Systems

Ning Ting, Zhang Rui-sheng*, Fan Xiao-liang, Liu Long-guang, Wang Hai-long
School of Information Science and Engineering,
Lanzhou University
Lanzhou, 730000, China
zhangrs@lzu.edu.cn
ningt05@lzu.cn

Abstract

Telecom Industry encounters rapid changing operations environment, which makes it urgent to innovate in its original Operations Supporting Systems. In order to attract more customers and provide higher quality of services, Service Providers rollout all kinds of new services such as digital audio and video services which result in different usage data record formats increasing very quickly. How to bill for these new records in accurate has become a hot problem which the original Telecom Billing System is facing. Telecom operators are finding new billing solutions to suit these changes. Furthermore, the performance of real time is also taken into account. This paper describes a simulative platform based on RT-Linux that aims to use OSS/J IP Billing API to integrate an open source billing system and solve the problem of different usage data record formats billing. We present the framework of the platform, build a trial network of VoIP and design a real time billing module to collect usage data records for the billing system.

1 Introduction

Telecom Billing System plays a very important role in Telecom Operations Supporting Systems. Telecom companies need an effective and accurate billing system to be able to assure their revenue. Billing system processes the usage of network equipment that is used during the service usage into a single Call Detail Record (CDR). The billing procedure involves receiving billing records from various networks, determining the billing rates associated with the billing records, calculating the cost for each billing record, aggregating these records periodically or real time to generate invoices, informing invoices to customers, and collecting payments received from the customer[1].

Nowadays, as the market requirements and customer demands change rapidly in telecom industry, all kinds of new services quickly rollout and correspondingly the formats of usage records vary greatly including IPDR, M-CDR, S-CDR and even more. How to effectively bill for them has become a great challenge to the billing system. The original billing system is always limited to bill for specific services that have existed. Modifying the internal architecture of original billing system to suit new services will be a low-efficiency and high-cost integration nightmare for telecom operators. Defining an all encompassing billing solution is emergent. Furthermore, with the concept of customer-centered deep into the market, the performance of real time is also becoming a measure standard to the billing system. In real telecom industry, the billing system is always running on commercial operating systems such as Unix operating system on off-the-shelf hardware. Actually, a commercial OS offers many attractive features such as developing tools and after-sales services, but in academic research field, the use of a "free" operating system such as RT Linux may be a good alternative.

In this paper, we describe a simulative billing platform based on RT-Linux to introduce the billing

*Corresponding author

solution using OSS/J [2] Internet Protocol (IP) Billing API[3] to convert different usage data formats for the billing system. The platform is comprised of open source software AstBill[4] that is the testing billing system running on RT-Linux, trial network of Voice-over-Internet Procotol(VoIP)[5]. The network elements in the trial network generate usage records and related information for the billing system. OSS/J IP Billing API is implemented as interfaces between AstBill and usage data records.

The organization of this paper is as follows. Section 2 covers the basics of telecom billing system and a brief introduction of an open source billing system AstBill. Section 3 discusses the building of the simulative platform and designing of a real time billing module. Section 4 presents some concerns on RT-Linux. Section 5 contains the conclusion of the paper.Section 6 is the acknowledgement.

2 Background

2.1 Brief Introduction of Telecom Billing System

The process of telecom billing system is complicated, starting from network elements that generate usage data to the billing system to usage collection, mediation, rating, and invoicing. The following Figure 1 shows a standard billing process. In this diagram, the customer calls customer care or works with an activation agent to establish a new account. As the customer makes calls, the connections made by the network (such as switches) create records of their activities. These records include the identification of the customer and other relevant information that are passed onto the billing system. The billing system also receives records from other carriers (such as a long distance service provider, or a roaming partner). The billing system now guides and updates these call detail records (CDRs) to their correct customer and rating information.

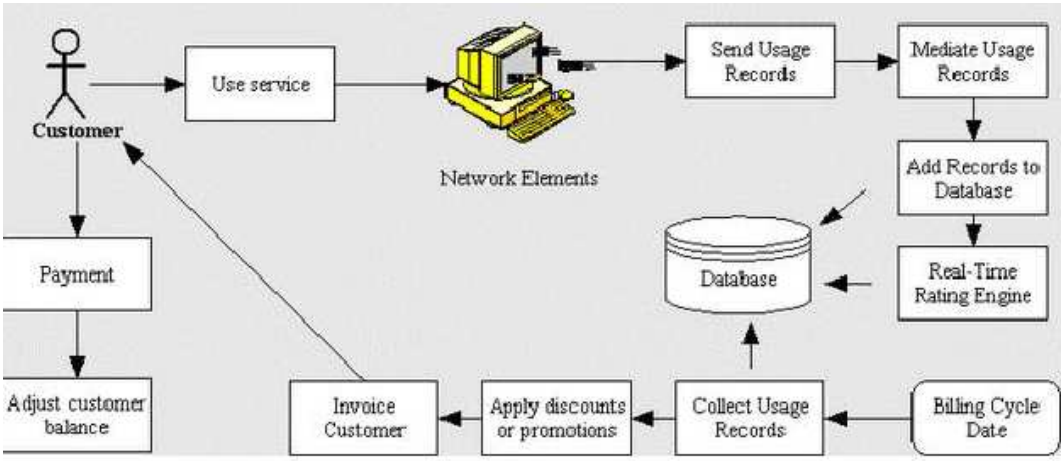


FIGURE 1: *Process of Billing System*

As information about the customer is discovered (e.g. rate plan), the updated billing records are placed in a billing pool so that they may be combined into a single invoice that is sent to the customer. The customer then sends his payment to the telecom service provider. Payments are recorded in the billing system.

In the billing system, there are two types of billing, One is postpaid billing, the other is prepaid billing. In post paid billing the customer may pay an insurance payment in advance, and he may pay the installation or setup fees, and in each billing cycle he will be invoiced (receive a bill) to pay for his usage of the service.In prepaid billing the customer buys a given amount of credits (duration, volume, number of events) and is then allowed to use the corresponding network resources as long as their account is in credit. Billing system receives customer usage records from the network elements and adjusts the customer credits. When their credit has been used up, network usage will be restricted. Prepaid corresponds to a real-time process, because transactions are only allowed if the user account is in credit, and this has to be checked in real-time[1].

We need to choose a billing system for the simulative platform and the best way is to use an open source product. Unfortunately, during the stages of research it became apparent that no existing open

source billing systems provided the ideal functionality, and as such a similar solution would have to be chosen and modified to meet the requirements. The choice for the billing system would be based on the conceptual similarity of the product in its current state to the real scenario just as described above, as well as the level of user support and product documentation to simplify the modification process.

Having analysed several open source possible alternatives, like Nexus Billing System[6], *starShop-Call Shop Billing Software[7], and AstBill-Asterisk[8] Billing and Management , it was decided that AstBill-Asterisk Billing and Management was the most suitable billing system. Despite not offering many of the features that are required, it currently appears to be the most viable product given the goals and scale of the platform.

2.2 Open Source AstBill System

AstBill is a Web Based Billing, Routing and Management Software for Asterisk and VoIP running on Linux. AstBill provides pre and post paid billing services that are just described above in telecom industry. AstBill is suitable to simulate the real telecom billing operations and has a calling card module. It is an open source software licensed under the GPL, and is maintained and developed by a community of users and developers. Users and administrators interact with the system through the GUI that is used to invoke the underlying modules. Figure 2 shows the GUI of AstBill.

FIGURE 2: *AstBill Interface*

In AstBill, the user end web interfaces give access to a range of functionality such as time based forwarding and billing, Call Data Records(CDR) including cost of each call and time based billing , switchboard (displays live status of users phones and ongoing calls), and rate table in currency of choice. The administrator end interfaces can show balance, expenditure, payments and number of calls on each account, include CDR cost and sales on each call and manage prepaid and postpaid customers.

The AstBill GUI will be modified as a non-real-time task using Free Software Foundation's General Tool-Kit (GTK). This GUI is used to send commands or parameters to interfaces implemented by OSS/J IP Billing API.

3 Simulative Billing Platform

3.1 System Overview

The purpose of the simulative billing platform is to implement a real time working module using OSS/J IP Billing API and provide uniform interfaces for AstBill system. RT-Linux was chosen as the RTOS for the platform, primarily because of the flexibility that comes with source code availability, and the large software base that comes with Linux itself and cost constraints.

Figure 3 shows the architecture of the simulative platform. There are two applications running on the RT Linux. One is the real time application that is implemented by OSS IP Billing API as a real time task running in kernel space. It collects usage data from network elements in trial network and passes them to AstBill system in real time. The other is the non-real-time application namely the AstBill system running in user space, its function is to accomplish the process of billing like mediation, rating, and invoicing. The real time and non-real-time applications communicate with each other through a modified AstBill GUI. We modified the source code of AstBill under the GPL to make it send command or parameters to the real time module via RT-FIFO[9]. The trial network of VoIP provides usage data records that are collected by the real time application.The protocol for data transforming is SNMP, while we can also choose other protocols because the OSS IP Billing API is protocol neutral.

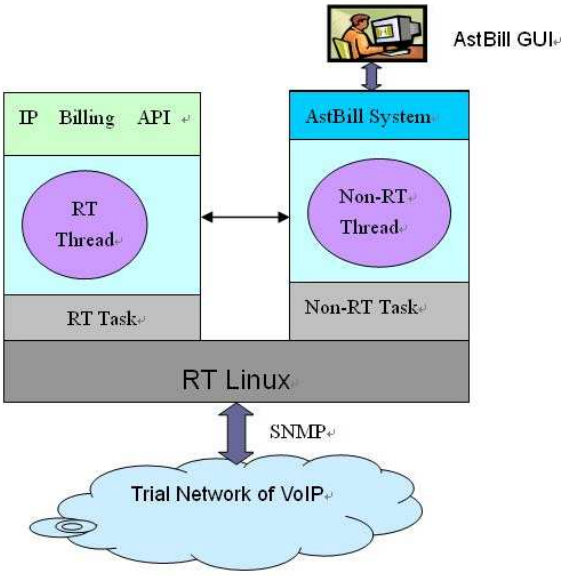


FIGURE 3: *Architecture of Simulative Platform*

3.2 OSS/J IP Billing API

In traditional telecom billing models, network elements perform usage measurement, collection, and record generation. The generated information is sent to supervisory systems and adjunct operation support systems for measurement aggregation, mediation, and correlation. The supervisory and adjunct systems generate call detail records (CDRs) that are passed to yet a third layer of system for operations such as rating, discounting, tariff accounting, debiting and invoicing.

The billing model for a telephony network built on top of an IP network is significantly more complicated than that of a traditional circuit switched voice telephony network. The causes of this increase in complexity include, but are not limited to:

- packet based transport
- a combination of connection and connectionless services
- signaling and bearer traffic (for the same call) may take radically different paths
- mixed-media calls, e.g., voice and video
- multicast calls

OSS through Java Initiative (OSS/J) IP Billing API is Billing Record Type agnostic, and can thus be used for collecting many kinds of records including IPDR, M-CDR, S-CDR and more. The usefulness of the API also is not just limited to specific service or network technology (such as one of IP, 2G, 2.5G, 3G). The API is defined to be service definition independent to handle advancements occurring in the technology, without requiring changes to the API. For example, the new Billing Data Formats (such as CDRs for new services) and multiple versions of Billing Record formats can be supported without requiring changes to the parameters passed to the methods in the API. This can be achieved by using XML based parameters and defining interfaces to identify data formats.

3.3 Trial Network of VoIP

The trial network of VoIP is to provide services usage data for the billing system. Figure 4 illustrates the simple Voice over IP service provider scenario we targeted. We envisaged that a VoIP service provider would have a number of geographically dispersed nodes interconnected through an outsourced wholesale IP service. Each node would have a SIP router or gateway and would be connected either directly to customer clients or through an outsourced access network.

The VoIP service provider would have their own network elements such as switches that create usage records depending on the activities of customers. These records include the identification of the customer and other relevant information that are passed onto the billing system.

3.4 Design of Real-Time Billing Module

The real time application described in this platform was designed as modules embedded into RT-Linux kernel. It was decided that the most effective implementation would require three basic layers. One to interact directly with the usage data and related information collected from trial network, one to directly access AstBill system, and an intermediary layer that will perform all the necessary conversions to allow seamless interaction between the components[10].The OSS IP Billing API provides standard interfaces that solve the problem of varied usage records formats transforming [11]. These three components will then be wrapped up in a module and deployed as a real time task in RT Linux kernel. Figure 5 is the schematic of the billing module.

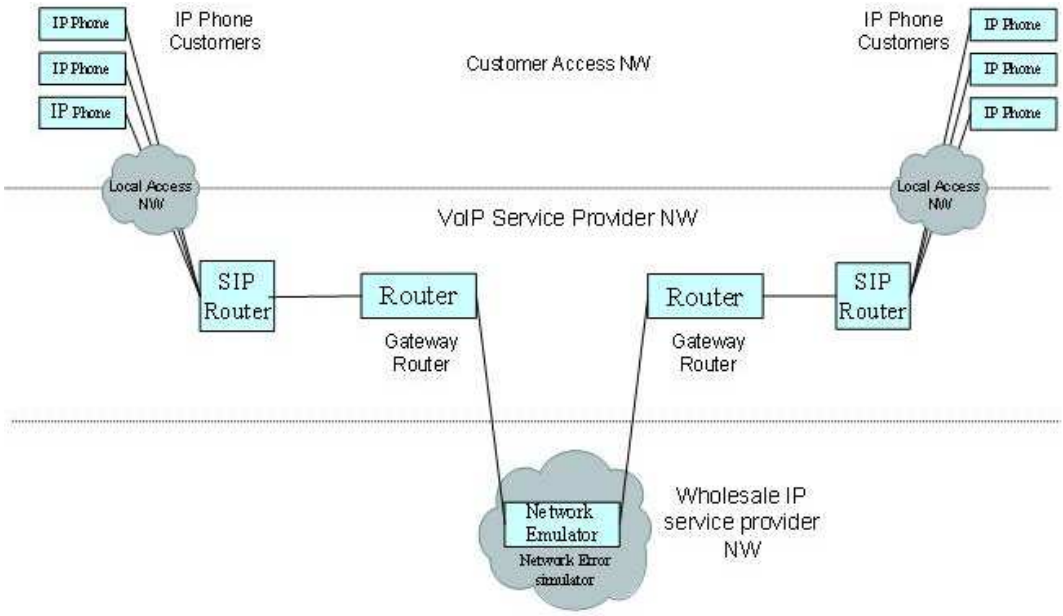


FIGURE 4: *Trial Network of VoIP*

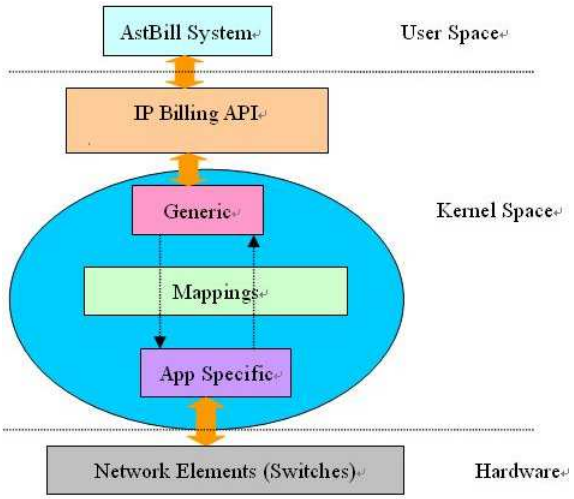


FIGURE 5: *Schematic of the Billing Module*

The procedure of kernel modules are programmed according to multiply thread mode. RT-Linux provides corresponding API to multiple thread programming. There are two main function methods to implement real time tasks in RT-Linux kernel:

int init_module

void cleanup_module

The billing modules are called to set the real time parameters like priority, interrupt period when they are loaded into kernel at the first time. In the billing modules, they can start many tasks. Here we just describe one RT task (task1) which collects usage data records from network elements such as switches. Before starting the real time task it should be initialized. The function of init_module() is initialization. There are three parts:

- Create three RT-FIFO to communicate information between RT thread in kernel and non-RT thread in user space. RT-FIFO1 is used to transform data between RT thread and non-RT thread. The data include CDRs and other related information. RT-FIFO3 is used to send parameters or commands from the non-RT tasks to RT tasks. RT-FIFO2 is used to save parameters or commands passed from RT-FIFO3. The API rtf_create() completes this work.
- Create corresponding threads for RT task1, and set the priority of scheduling. The API pthread_create() completes this work.
- When non-RT processes pass control commands through RT-FIFO3, RT processes give responses in time. Consequently, a subroutine is given to trace the change of RT-FIFO3 in RT programs.

There are two ways to implement main programs in kernel mode: interrupt handling and periodic real time thread mode. The billing modules adopt periodic real time thread mode that means collecting and transforming usage data records periodically with non RT thread in user space.

4 Discussion

RT-Linux is attractive for those real-time applications that require a mix of hard and soft real-time tasks. However, there are many concerns when using RT-Linux in the manner that we have chosen- that is:

IPC The RT-Linux executive gives the RT tasks direct communication between themselves and the user-level tasks via RT-FIFOs. However, it does not give much help in the communicating between the RT-tasks. There was concern that the user might have to write specialized routines an libraries to circumvent this problem.

Debugging Given that real-time tasks are placed directly in kernel space, the controls people wondered in there was little that a person would do to debug code in kernel space other than using *printk* statements.

It is interesting that the source code availability in combination with placing real-time tasks directly into kernel space caused certain problems. Because the RT tasks are in the kernel, it might appear attractive for a real-time programmer to search the source code for candidate functions without knowing the exact details of how each function works. There's little way for the user to realize what function can and cannot be called by a user-defined real-time task [12].

5 Conclusion

This paper describes the use of RT-Linux to build a simulative billing platform that implements the OSS/J IP Billing API and provides real time collection of usage data. This platform validates the solution of using OSS/J IP Billing API to solve the problem of billing for varied usage data formats and facilitates the integration of billing system in telecom industry. To build different trial network and provide different services of usage data format for billing will be the future work.

6 Acknowledgement

This work was supported by National Natural Science Foundation of China through projects: Research on Computational Chemistry E-SCIENCE and its Applications (Grant no. 90612016). And this work is also supported by China National Technology Platform.

References

[1] Hatem Mostafa, *Billing System: Introduction*

[2] *Telemanagement Forum. OSS/J Overview*, `http://www.tmforum.org/browse.aspx?catid=2896`, 2006

[3] *OSS/J IP Billing*, `http://www.tmforum.org /browse.aspx?catID=2983&linkID=31863`, 2006

[4] *AstBill*,`http://astbill.com/`, 2006

[5] *VoIP*, `http://www.fcc.gov/voip/`, 2006

[6] *Nexus Billing System*, `http://sourceforge.net /projects/billing-nexus`

[7] **starShop - Call Shop Billing Software*, `http://sourceforge.net/projects/starshop`

[8] *Asterisk*, `http://www.voip-info.org/wiki-Asterisk`

[9] *Linux-HA, Linux-HA project*, `http://www.linux-ha.org/`

[10] *Open OSS Trouble Ticket integration with OSS/J Initial Investigation Project Report*

[11] *Design of the OSS IP Billing API Reference Implementation*

[12] Marty Humphrey, Edgar Hilton, Paul Allaire,*Experiences Using RT-Linux to Implement a Controller for a High Speed Magnetic Bearing System*

Visual Event-Condition-Action Rules with Temporal Events *

Ying Qiao, Hongan Wang, Kang Zhong and Xiang Li
Intelligence Engineering Lab Institute of Software,
Chinese Academy of Sciences
Beijing 100080,P.R. China
qiaoying@@ios.cn

Abstract

To support higher level of intelligence in real-time applications, the real-time active database (RTADB) should have powerful reasoning capability to solve complicated problems, especially the problems with temporal properties. This consequently requires the event-condition-action (ECA) rules in RTADB to have strong expressiveness for complicated temporal information in the system. However, traditional ECA rules cannot meet this requirement since they only describe basic temporal elements such as quantitative temporal constraints. To solve this problem, we present visual ECA rules with a set of novel temporal events to describe complicated quantitative temporal information. Furthermore, smart home applications are used to validate the proposed rule.

1 Introduction

The real-time active database (RTADB) serves as a key infrastructure in many real time systems such as medical care systems and telecom systems. For example, in a value-added telecom service, called "location tracking service", which is usually used to prevent the loss of elder people, the real-time active database is responsible to collect the data about the location of people that are tracked by the GPS phone and generate notifications to subscribers according to predefined conditions in real-time. Meanwhile, the demands on higher level of intelligence in above systems have been expanding. In the case of location tracking service for the elder people, the subscribers may want to know not only current location of the elder person but also if there is any immediate danger in the circumstance on that location. In this case, the RTADB with powerful reasoning capability for complicated temporal problems is necessary. Therefore, being the basis for reasoning in RTADB, Event-condition-action (ECA) rules [8] are required to have strong expressiveness for these complicated temporal problems. Unfortunately, existing ECA rules are unable to meet above requirements. Many ECA rules in traditional rule systems such as Starburst [19], POSTGRES [17], Ariel [10], SAMOS [7], HiPAC [6], and EXACT [12] only support basic temporal events, e.g., absolute timing events, relative timing events, periodic timing events and sequence events etc [4]. These events only express qualitative temporal information such as the order of events and very limited quantitative temporal information such as the time point. In recent years, some works have been done on enhancing the expressive of ECA rules for temporal information. Some of them focus on extending the basic temporal event set. Besides instaneous events, they also support the events that last for a period of time [9]. Others attempt to incorporate temporal logic into the condition part of an ECA rule. For example, Sistla [16] incorporates the past temporal logic and future temporal logic with aggregate functions into the condition part. In addition, Muller [11]describes temporal information in the action part of an ECA rule by imposing timing constraints on the action. However, these extended ECA rules are still unable to express complicated quantitative temporal information such as real-time constraints, i.e., timeouts and duration. To solve above problems, we present visual event-condition-action rules with a set of novel temporal events. Compared to existing ECA rules, our visual ECA rules contribute to following aspects: (1) they support various novel temporal events to describe complicated

*This work is supported by France Telecom (Grant No. 46135653)

quantitative temporal information. This enhances the capabilities of ECA rules to express complicated temporal problems whereas to provide a sound basis for RTADB to reason these problems; (2) they provide a natural way for users to understand and define ECA rules. This greatly enhances users' involvement in the reasoning process, thereby improving the flexibility of the reasoning. The rest of this paper is organized as follows: section 2 presents the novel temporal events in ECA rules; section 3 addresses the visual model for ECA rules; section 4 describes the graphical user interface for editing visual ECA rules; section 5 addresses a case study based on smart home applications to validate the ECA rules with proposed novel temporal events; conclusions and future works are given in section 6.

2 Temporal Events in ECA rules

Existing ECA rules are unable to express complicated temporal behaviours in the system since they only support very limited temporal events that lack capability to specify complicated quantitative temporal constraints. Thus, we present a set of novel temporal events to support timing constraints so as to enhance the expressiveness of ECA rules for complicated temporal problems.

2.1 Quantitative Temporal Constraints and Metric Temporal Logic

Quantitative temporal constraints describe critical behaviours of a real-time system. They actually refer to the timing constraints such as timeouts and duration serving as exact measure for events and among events. The quantitative temporal constraints allow the definition of quantitative temporal relationships - such as distance among events and durations of events in time units [5].

Metric temporal logic (MTL) is a good tool to express quantitative temporal constraints. It extends liner temporal logic (LTL) by time-bounded temporal operators to support the specification of relative- and real-time constrains [18, 14]. In general, there are two time-bounded operators, denoted as *during* and *within* respectively.

The *during* operator is used to express the concept of duration. It is represented as an interval shown as follows:

$$during : [x, y]$$

Here, x is the starting time and y is the ending time for the interval. They are either relative time units or absolute time instants. The within operator describes the concept of timeout. It is represented as a time restrictor shown as follows:

$$within :\leq x \text{ or } within :< x$$

Here, x is the upper bound of the time restrictor. It is either a relative time unit or an absolute time instant. In MTL, four LTL operators (Sometime $\Diamond$, Always $\Box$, Until U, Next $\bigcirc$) can be characterized by relative- and real-time constraints specifying the duration of the temporal operator [18, 14]. Thus, assuming A and B are two propositions, the meanings of following typical MTL representations are shown as below:

$\Box_{[3,6]}$**A:** A is always true from 3 to 6 time instant.

$\Box_{\leq 6}$**A:** A is always true within 6 time units.

$\Diamond_{[3,6]}$ **A:** A is eventually true from 3 to 6 time instant.

$\Diamond_{\leq 6}$**A:** A is eventually true within 6 time units.

$\bigcirc_{[3,6]}$**A:** A is true in the next instant from 3 to 6 time instant.

$\bigcirc_{\leq 6}$**A:** A is true in the next instant within 6 time units.

A $\mathbf{U}_{[3,6]}$ **B:** B is become true and before then A is always true from 3 to 6 time instant.

A $\mathbf{U}_{\leq 6}$ **B:** B is become true and before then A is always true within 6 time units.

The novel temporal events presented in this paper are derived by applying time-bounded operators, i.e., during operator and within operator, to basic temporal events supported by current ECA rules. They are either primitive or composite. We will address these novel temporal events in following sections.

2.2 Primitive Temporal Events

- *Durative events:* A durative event is based on imposing during operator on a regular event. It indicates a specific event occurring in a specified interval limited by two time instants.

2.3 Composite Temporal Events

A novel composite temporal event is the sequence of various primitive temporal events or regular primitive events connected by novel temporal event operators. These novel event operators are derived by imposing real time constraints on regular event operators. They are addressed as follows:

- *Time constrained sequence:* The time constrained sequence is derived by applying within operator to regular sequence operator, i.e., by imposing timing constraints on the distance between two events. It is denoted as Seq-Within[X], where X represents a time restrictor. The time constrained sequence of event E1 and E2, E1 Seq-within[10 minutes] E2, occurs when both E1 and E2 have occurred in that order within 10 minutes. Other events may occur between E1 and E2.

- *Durative sequence:* The durative sequence, denoted as Seq-During[X,Y], is derived by applying during operator to sequence operator, where X and Y represent the starting time and ending time for a specific interval. The durative sequence of events E1 and E2, E1 Seq-During[10:00AM, 12:00PM] E2, occurs when both E1 and E2 have occurred in that order in a specific interval from 10:00AM to 12:00PM. Other events may occur between E1 and E2.

- *Durative conjunction:* The durative conjunction, denoted as AND-During[X,Y], is derived by incorporating during operator into conjunction operator AND, where X and Y are the starting time and ending time for a specific interval. The durative conjunction of event E1 and E2, E1 AND-During[10:00AM, 12:00PM] E2, occurs when both E1 and E2 have occurred in any order during a specific interval from 10:00AM to 12:00PM with possible other events in between.

- *Durative disconjunction:* The *durative disconjunction*, denoted as OR-During[X, Y], combines the disjunction operator OR with during operator. Here, the interval is defined by the starting time X and ending time Y. The *durative disconjunction* of events E1 and E2, E1 OR-During[10:00AM, 12:00PM] E2, occurs when either E1 or E2 occurs or when both E1 and E2 occur during a specific interval from 10:00Am to 12:00PM.

- *Durative between:* The durative between applies during operator to regular between operator. The durative between of Event E1 and E2, denoted as Between-During (E1, E2)[X,Y], occurs when there are events occur between the initiating event E1 and terminating event E2 in a given interval limited by starting time X and ending time Y, ignoring the relative order of their occurrences.

Besides above novel temporal events, the proposed ECA rules still support traditional temporal or regular events, i.e., absolute timing events, relative timing events, periodic events, sequence events, conjunction events, disconjunction events, Between events, Any events, Count events and negation events. The details of these traditional temporal events can be found in [19, 17, 10, 7, 6, 12, 4] and are out of the scope of this paper.

2.4 Formalization and Analysis

In this section, we will use temporal logics to formalize proposed novel temporal events and traditional temporal events to analyse their capabilities for expressing complicated temporal problems, i.e., quantitative temporal constraints. In this context, an event E (event E is either primitive or composite) can be considered as a formula which has a value of True or False. Here, when event E occurs, the value of this formula is True; otherwise its value is False.

Thus, the novel temporal events are formally represented by Metric temporal logic as follows:

- *Durative events:*a durative event, 'event E occurs during the interval from 10:00am to 11:00am ', can be formally represented as follows:

$$\Diamond_{[10,11]}\mathrm{E}$$

- *Timing constraint sequence:*The time constrained sequence of event E1 and E2, E1 Seq-within[10 minutes] E2, can be formally represented by as follows:

$$\mathrm{E}_1 \rightarrow \Diamond_{\leq 10}\mathrm{E}_2$$

- *Durative sequence:* The durative sequence of events E1 and E2, E1 Seq-During[10:00AM, 12:00PM] E2, can be formally represented as follows:

$$\Diamond_{[10,12]}(\mathrm{E}_1 \rightarrow \Diamond\mathrm{E}_2)$$

- *Durative conjunction:* The durative conjunction of event E1 and E2, E1 AND-During[10:00AM, 12:00PM] E2, can be formally represented as follows:

$$\Diamond_{[10,12]}(\mathrm{E}_1 \wedge \mathrm{E}_2)$$

- *Durative disconjunction:* The durative disconjunction of events E1 and E2, E1 OR-During[10:00AM, 12:00PM] E2, can be formally represented as follows

$$\Diamond_{[10,12]}(\mathrm{E}_1 \vee \mathrm{E}_2)$$

- *Durative Between:* The composite event, Between-During(E1, E2) [9:00AM, 11:00AM] can be formally represented as follows: (Assume E is an event)

$$\Diamond_{[9,11]}((\mathrm{E}_1 \rightarrow \Diamond \mathrm{E}) \wedge (\mathrm{E} \rightarrow \Diamond \mathrm{E}_2)$$

Meanwhile the traditional temporal events can be formalized by linear temporal logic (LTL) as follows:

- *Sequence:* The sequence of event E1 and E2, denoted as E1 Seq E2, is formally represented as follows:

$$\mathrm{E}_1 \rightarrow \Diamond \mathrm{E}_2$$

- *Conjunction:* The conjunction of event E1 and E2, denoted as E1 AND E2, is formally represented as follows:

$$\Diamond(\mathrm{E}_1 \wedge \mathrm{E}_2)$$

- *Disconjunction:* The disconjunction of events E1 and E2, denoted as E1 OR E2 can be formally represented as follows:

$$\Diamond(\mathrm{E}_1 \vee \mathrm{E}_2)$$

- *Between:* The composite event, Between (E1, E2) can be formally represented as follows: (Assume E is an event)

$$\Diamond((\mathrm{E}_1 \rightarrow \Diamond \mathrm{E}) \wedge (\mathrm{E} \rightarrow \Diamond \mathrm{E}_2)$$

Based on above formalization, we found that the LTL which can only specify qualitative temporal constraints is powerful enough to express most of traditional temporal events. However, the proposed novel temporal events have gone beyond the expressiveness of LTL. They need to use more expressive temporal logic - MTL, which can specify both qualitative and quantitative temporal constraints to formalize. This situation demonstrates that the proposed temporal events have powerful expressiveness for quantitative temporal constraints while the traditional temporal events do not have. Consequently, the proposed ECA rules in this paper have more powerful expressiveness for complicated temporal problems than traditional ECA rules.

3 Visual modelling of ECA rules

To facilitate the understanding of end users, we will give the visual model for the proposed ECA rules.

3.1 Framework

The overall model of a proposed ECA rule, denoted as M, can be represented as a triple:

$$M = (E, C, A)$$

E is a set of temporal events, denoted as $(e_1, e_2, ..., e_n)$, where$e_i (1 \leq i \leq n)$ is an individual event. It is either primitive or composite; is a set of conditions, denoted as $(c_1, c_2, ..., c_m)$, where $c_i (1 \leq i \leq m)$ is a condition which is either primitive or composite; A is a set of actions. It is denoted as $(a_1, a_2, ..., c_p)$, where $a_i (1 \leq i \leq p)$ is an action that is either primitive or composite. To visualize above model, we introduce a set of graphical notations to model the events, conditions, actions, and the connections between these components. The framework of the visual model for an ECA rules is shown in Figure 1.

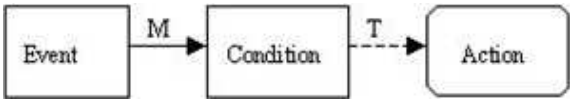


FIGURE 1: *Framework of visual model for an ECA rule*

Connection type	Visual model	Explanations
Connections between an event and a condition/action	M	N/A
Connections between a condition and an action	T	It means the rule will be fired when the condition is True.
	F	It means the rule will be fired when the condition if False
Connections between two ECA rules		N/A

TABLE 1: *Connections for the components in ECA rules*

In Figure 1, an event is modelled by a square associated with a set of attributes that imply critical characteristics of the event. Similarly, a condition is modelled by a square with associated attributes describing critical characteristics of the condition. An action is visually modelled as a rounded rectangle labelled with name of the action and content of the action depending on the specific application. Meanwhile, three types of connectors are provided to model connections between components in ECA rules, including the connections between an event

and a condition/action, the connections between a condition and an action and the connections between two ECA rules. The visual models for these connectors are shown in Table 1.

3.2 Visual modelling of temporal events

Primitive events

A single square associated with corresponding attributes is used to model a primitive event. The square is labeled with name of the event and an icon indicating the type of primitive event. Table 2 shows the icon for each type of primitive events.

Event Type	Icon
Durative event	
Periodic event	
Relative timing event	
Absolute timing event	

TABLE 2: *Icons for primitive events*

Table 3 listed the associated attributes for each type of primitive events.

Event type	Attributes	Meaning
Durative event	Starting time	The starting time to limit an interval
	Ending time	The ending time to limit an interval
Relative timing event	Time offset	A duration offsetting from a specific time point
	Time reference	A time point used for reference
Periodic event	Interval	The duration between two consecutive occurrences
Absolute timing event	Specified time point	The fix time point

TABLE 3: *Associated attributes for primitive events*

Table 4 shows the visual models of primitive events.

Event type	Event name/ content	Visual Model	Attributes/ vale
Durative event	DE: during [10:00am, 12:00pm]	DE	Starting time: 10:00am
			Ending time: 12:00pm
Relative timing event	RE: 10 minutes after occurrence of event B	RE	Time offset: 10 minutes
			Time reference: Event B
periodic event	DPE: every 10 minutes	DPE	Interval: 10 minutes
Absolute timing event	AE: At 10:00am	AE	Specified time point: 10:00am

TABLE 4: *Associated attributes for primitive events*

Composite events

A composite event is modelled by nested squares with associated attributes. Specifically, a square, called framework square, is labelled with an icon and name of the event to identify the specific event. The icon indicates which type of the event operators is used to form the composite event. The framework square serves as a container which contains squares representing constituent primitive events. Table 5 gives the icons for each type of event operators.

Event operator	Icon
Time constrained sequence	
Durative sequence	
Durative conjunction	
Durative disconjunction	
Durative Between	
Any	
Count	
Negation	

TABLE 5: *Icons for event operators*

Table 6 list the associated attributes for the composite temporal events based on above temporal event operators.

Event types	Attributes	Meanings
Composite event based on Time constrained sequence	Constrained time	The time restrictor for the interval between occurrence of two events
	Consecutiveness	Indicates if constituent events occur continuously ('True' for continuous; 'False' for discontinuous)
Composite event based on Durative sequence	Starting time	The starting time of an interval
	Ending time	The ending time of an interval
	Consecutiveness	Same to above
Composite event based on Durative conjunction	Starting time	The starting time of an interval
	Ending time	The ending time of an interval
Composite event based on Durative disconjunction	Starting time	The starting time of an interval
	Ending time	The ending time of an interval
Composite event based on Durative between	Starting time	The starting time of an interval
	Ending time	The ending time of an interval
Composite event based on any	Number of event	How many events need to occur among specified n distinct events
Composite event based on count	Number of occurrence	How many times the specified event needs to occur
	Consecutiveness	Indicates if specified event occurs continuously
Composite event based on Negation	Starting time	The starting time of an interval
	Ending time	The ending time of an interval

TABLE 6: *Attributes for composite events*

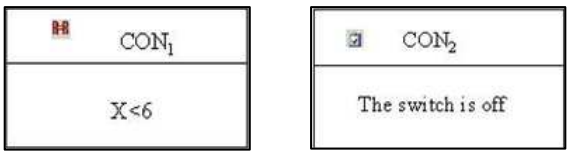
Table 7 shows visual models of composite events.

Event types	Event name/ Content	Visual Model	Attributes/values
Time constrained sequence	TCSE: E_1 *Seq-within*[10 minutes] E_2	TCSE E_1 E_2	Constrained time: 10 minutes Consecutive-ness: True
Durative sequence	DSE: E_1 *Seq-During* [10:00am, 12:00pm] E_2	DSE E_1 E_2	Staring time: 10:00am Ending time: 12:00pm Consecutiveness: True
Durative conjunction	DCE: E_1 *AND-During* [10:00am, 12:00pm] E_2	DCE E_1 E_2	Starting time:10:00am Ending time: 12:00pm
Durative disconjunction	DDE: E_1 *OR-During* [10:00am, 12:00pm] E_2	DDE E_1 E_2	Starting time: 10:00am Ending time: 12:00pm
Durative between	DBE: *Between-During* (E_1,E_2)[9:00am, 11:00am]	DBE E_1 E_2	Starting time: 9:00am Ending time: 11:00am
Any	AE: *Any-During* (2,E_1,E_2,E_3)	AE E_1 E_2 E_3	Number of event: 2
Durative count	DCE: *Count-During* (E_1,2)	DCE E_1	Number of occurrence: 2 Consecutiveness: True
Negative	NE: *NOT* E_1 [10:00am, 12:00am]	NE E_1	Starting time: 10:00am Ending time: 12:00pm

TABLE 7: *Visual Models of Composite Events*

3.3 Visual modelling of conditions

The conditions supported in proposed ECA rules include primitive conditions and composite conditions. A primitive condition is atomic. It is modelled by a single square associated with a set of attributes. The square is labeled with a specific icon indicating the type of the primitive condition and the content of the condition. There are two types of primitive conditions, i.e., relational conditions and customized conditions. For a relational condition, the content of the condition is a relational expression. The associated attributes of the square are called logic method, variable, operator and value. Here, variable, operator and value are three parameters for a relational expression. Logic method indicates how the evaluation result of the condition impacts fire of rules. If the value of logic method is true, it means the rule will be fired when the relational expression is true; otherwise, the rule will be fired when the relational expression is false. For example, a relational condition CON1, 'X¡6', can be modelled in Figure 2. Here, is the icon for the relational condition. 'CON1' is the name of this condition. The value of attribute logic method is true. The values of attributes variable, operator and value are X, ¡ and 6 respectively. For a customized condition, the content of this condition depends on the specific application. The attribute associated with the square is logic method. Assume a customized condition CON2 is described as 'The switch is off'. Thus, its visual model is shown in Figure 2. Here, is the icon for the customized condition. 'CON2' is the name of this condition. The content of the condition is 'The switch is off'. The value of attribute logic method is True.

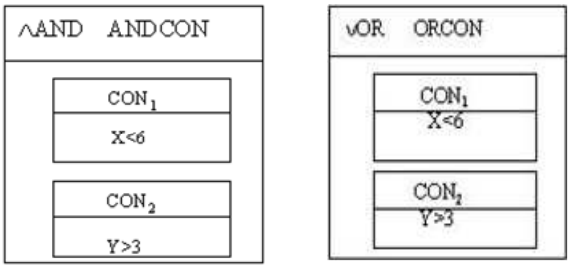


(a) Relational condition (b) Customized condition

FIGURE 2: *Visual models of primitive conditions*

A composite condition is derived by combining primitive conditions via logic operators. It is modelled by nested squares associated with corresponding attributes. The framework square is labelled with name of the condition and a specific icon identifying the type of composite condition. It serves as a container encapsulating squares representing the constituent conditions. The attribute associated with the framework square is logic method. There are

two types of composite conditions, i.e., AND conditions and OR conditions. The former are formed by connecting constituent conditions via AND operator while the latter are formed by connecting constituent conditions via OR operator. Thus, an AND composite condition, described as 'X¡6 and Y¿3', is modelled as Figure 3. (AND is the icon for the AND condition. 'ANDCON' is the name of this condition. The squares labelled with CON1 and CON2 represent two primitive conditions 'X¡6' and 'Y¿3'.) An OR composite condition, described as 'X¡6 or Y¿3', is modelled as Figure 3. (OR is the icon for the OR condition. 'ORCON' is the name of this condition.)



(a) AND condition (b) OR condition

FIGURE 3: *Visual models of composite conditions*

3.4 Visual modelling of actions

An action is modelled by a rounded rectangle labelled with name of the action and content of the action which depends on the specific application. Assume an action named ACT1 is described as 'raise an alarm'. It is modelled as Figure 4.

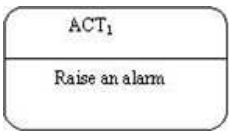


FIGURE 4: *Visual model of action*

3.5 Example

Assume there are two rules R1 and R2 shown below. Execution of R1 can trigger R2.

```
R1:On "Every 30 minutes"
    If "The air pressure of the tank is higher
    than 200"
    Do "Open the air pressure valve"
R2:On "15 minutes after the air pressure valve
    is open"
    If "The air pressure of the tank is lower
    than 100"
    Do " Close the air pressure valve"
```

Assume the variable for the air pressure of the tank is PRE. EVENT1 represents the external event "the air pressure valve is open". The periodic event "Every 30 minutes" is denoted as Per_Event1. The relative timing event "15 minutes after the air pressure valve is open" is denoted as Rel_Event1. The action "Open the air pressure valve" is denoted as ACT1 while the action "close the air pressure valve" is denoted as ACT2. Thus, the visual models of R1 and R2 are shown in Figure 5.

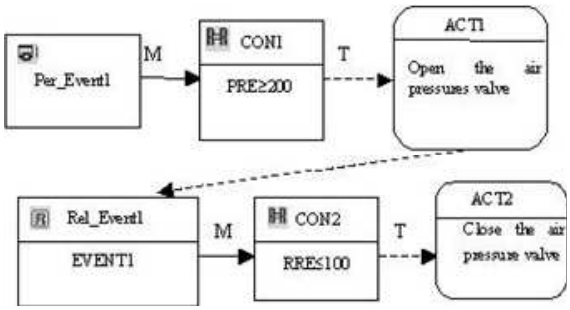


FIGURE 5: *Visual models of ECA rules*

3.6 Graphical user interface for editing visual ECA rules

To facilitate user involvement in the reasoning process, we design and implement a graphical user interface (GUI) to provide an edit environment for visual ECA rules that materialize the model addressed in section 3. Figure 6 shows this graphical user interface. This GUI includes four areas, i.e., navigator area, outline area, rule edit area, and property define area. In the navigator area, the name of project and the files contained in this project are listed. When the user defines a group of rules for a specific application, a project contained several files is created. These associated rules are saved into corresponding files under this project. The outline area is responsible to display the overall picture of selected group of ECA rules defined by the user. It has two views: one view is textual view which lists all components in selected ECA rules by texts; another view is graphical view which displays the overall imagine of selected ECA rules.

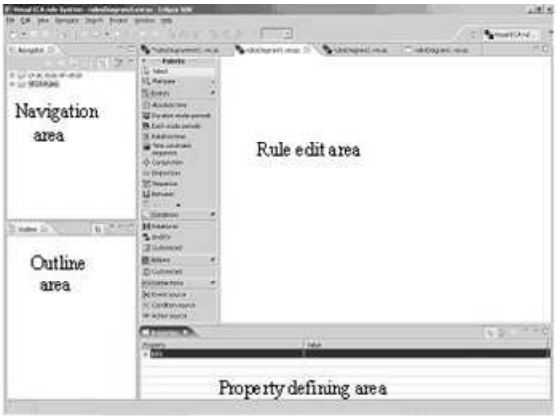


FIGURE 6: *Graphical User Interface for defining visual ECA rules*

In the rule edit area, a set of graphical components are provided. Users could drag corresponding graphical components into the edit area to draw visual ECA rules as needed. There are four types of graphical components, i.e., event components, condition components, action components and connector components. In this case, users are allowed to depict a group of visual ECA rules that have independencies at one time. This facilitates the rule checking in the background and user understanding for whole rule system. Furthermore, in the property defining area, the user can define the attributes associated with each graphical block in visual ECA rules drew in the rule edit area.

4 Case study

Smart home is the home environment that can proactively change to provide services that promote independent living. In smart home system, each apartment has the ability to detect the movement of a person throughout the house and in addition provides a number of interfaces to monitor the person's interaction with various home appliances. All of the apartments are linked to a central monitoring facility (CMF) to which alarms can be relayed when additional assistance is required [2, 3]. The core of a computer-based smart home system is a real-time active database system (RTADB) where the data collected by sensors are put in. With ECA rules, the RTADB has the enhanced capabilities to detect complex events and contexts in order to distinguish between several situations of interest. In this context, ECA rules in the system may cause the system to change its own perception about the state the house is in, for example from normal context to potentially hazardous context. Thus, the RTADB system is able to anticipate potential or actual hazardous situations and intelligently discern how to best advise carers to increase safety and living standards for a person inside the monitored house. In addition, by executing ECA rules, the related persons (medical staffs and relatives) will be notified via their special devices or value-added service for their cell phones. Figure 7 shows the architecture of computer-based smart home system.

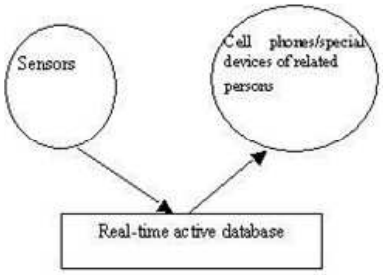


FIGURE 7: *Architecture of computer-based smart home system*

5.1 Requirements in smart home system

The information generated from the sensors within the house and the interfaces embedded into common domestic appliances can be processed by the RTADB to provide assistance to the person in a number of different ways: prevention of dangerous situations, comfort, security and health [2, 3]. In this paper, we focus on requirements in the category of 'prevention of dangerous situations' and 'health' since applications in these two categories have special requirements on the timing issues. Furthermore, these requirements contain a lot of complicated quantitative temporal information. For example, nobody passes after the door is open means that the door is unexpectedly opened. This is potentially dangerous during the sleeping time, i.e., from 10:00PM to 7:00AM, and will trigger an alarm to the person. Another example for complicated quantitative temporal information is that the medical staffs may care about the relationship between the change of heart beats and the blood pressure. Specifically, they may be interested in if the abnormal change of the blood pressure occurs after the abnormal change of the heart beats within certain time. The detailed requirements for ECA rules are shown in Table 8. (We label the requirements from Req1 to Req9.)

Prevention of Dangerous Situation
Req1: On "1 hour after the cooker is on" If "the cooker is still on" Do "Trigger alarm"
Req2: On " 15 minutes after the door is open" If "the door is still not locked" Do "Trigger alarm"
Req3: On "1 minutes after leaving the bath room" If "the light leaving on" or "the water leaving on" Do "Create a reminder"
Req4: On " the person does not pass" after "the door is open" during [10:00pm, 7:00Am] Do " Create reminder"
Health
Req5: On "At 10:00AM" If "the person's blood pressure is higher than 200/175" Do "Notify the medical staff"
Req6: On " Monitor blood pressure every 2 hours" If " the Blood pressure higher than 200/175 for more than two times in past 8 hours" Do "Notify the medical staff"
Req7: On "the person is in bed during [T1, T2]" If " the person is active during [D1, D2]" and "[T1, T2] is included in [D1, D2] and "Duration [T1, T2] is longer than 2 hours" Do "Report the context abnormal"
Req8: On " Monitor blood pressure every 2 hours" and "blood pressure higher than 200/175 for more than two successive samples" Do " Notify the medical staff"
Req9: On " Blood pressure higher than 200/175" after "the heart beat is higher than 150" within 5 minutes Do " Notify the medical staff"

TABLE 8: *Requirements for ECA rules in smart home system*

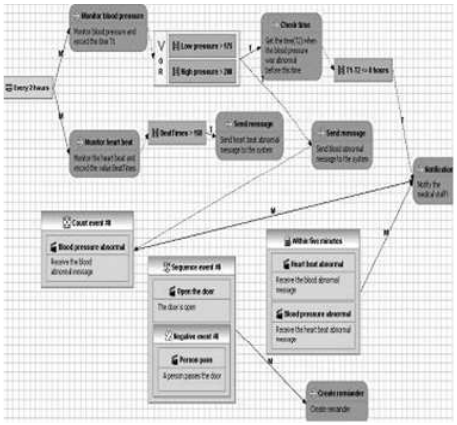
4.1 Visual ECA rules in smart home system

Based on requirements mentioned above, we can design ECA rules for smart home systems. In this context, each requirement mentioned above is decomposed into several detailed requirements that are further mapped into the event part, condition part and action part of the ECA rule. Table 9 describes the design of ECA rules in the system.

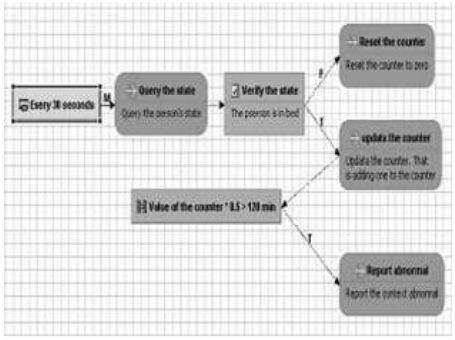
Requirements	Temporal Events	Conditions
Req1	Relative timing event	Relational condition
Req2	Relative timing event	Relational condition
Req3	Relative timing event	OR condition
Req4	Durative Sequence event	N/A
Req5	Absolute timing event	Relational condition
Req6	Periodic event	AND condition
Req7	Durative event	AND condition
Req8	Conjunction event	N/A
Req9	Time constrained sequence event	N/A

TABLE 9: *Design of ECA rules*

The table 9 shows that the proposed ECA rules have strong capabilities to express complicated temporal problems. For example, for Req4, the composite event based on durative sequence is capable of describing the complicated quantitative temporal information stating that " the person does not pass" after "the door is open" during [10:00pm, 7:00Am]. For Req9, the composite event based on time constrained sequence can describe the complicated quantitative temporal information stating that the abnormal change of blood pressure occurs after the abnormal change of heart beats within specific time. However, existing ECA rules are unable to express above complicated temporal information since they only support regular sequence of events and do not consider the timing constrains on the distance between two events and the bounded duration for the event sequence. Based on the design in Table 9, we can define these ECA rules via the graphical user interface. The visual ECA rules are shown in Figure 8.



(a) ECA rules for Req1-Req6, Req8-Req9



(b) ECA rules for Req7

FIGURE 8: *Visual ECA rules in smart home system*

5 Conclusions and future works

Enhancing expressiveness of traditional ECA rules for complicated temporal information is essential for improving reasoning capability of the real-time active database for complicated temporal problems. Thus, we present visual ECA rules with a set of novel

temporal events to describe complicated quantitative temporal constraints. Based on this, we design and implement a graphical user interface (GUI) for editing visual ECA rules, which greatly facilitate user's involvement in the reasoning process. Furthermore, we use smart home applications to validate our work.

References

[1] T. Bouaziz and A. Wolski.,1998, *Fuzzy triggers: Incorporating imprecise reasoning into active databases.*, In Proceedings of 14th International Conference on Data Engineering, pages 108–115.

[2] J. Augusto and C. Nugent., 2004 , *A new architecture for smart homes based on adb and temporal reasoning.* In Proceedings of 2nd International Conference on Smart Homes and Health Telematic (ICOST2004), pages 106–113

[3] J. Augusto and C. Nugent., 2004, *The use of temporal reasoning and management of complex events in smart homes.* In Proceedings of European Conference on Artificial Intelligence (ECAI 2004), pages 778–782

[4] J. Bailey and K. Rarmamohanarao., 1995, *Issues in active database.*, In *Proceedings of 6th Australasian Database Conference*, pages 27–35

[5] P. Bellin, R. Mattolini, and P. Nesi., 2000, *Temporal logics for real-time system specification.* ACM Computing Surveys, 32(1)

[6] U. Dayas, A. Buchmann, and D. McCarthy., 1988, *Rules are objects too.*, In Proceedings of the 2nd International Workshop on Object-oriented Databases, pages 129–143.

[7] S. Gatziu, A. Geppert, and K. Dittrich. Integrating active concepts into an object-oriented database system. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, pages 399–415, Naphlion, Greece, aug 1991.

[8] D. Goldin, S. Srinivasa, and V. Srikanti.,2004, *Active databases as information systems.* In Proceedings of 8th International Database Engineering and Applications Symposium (IDEAS 2004), pages 123–130.

[9] R. Gómez and J. Augusto., 2004, *Durative events in active databases.*, In Proceedings of 6th International Conference on Enterprise Information Systems, pages 306–311.

[10] E. Hanson., 1992, *Rule condition testing and action execution in ariel.*, In Proceedings of SIGMOD International Conference on Management of Data, pages 49–58.

[11] R. Muller, U. Greiner, and E. Rahm., 2004 *Agentwork: A workflow system supporting rule-based workflow adaptation.* Data and Knowledge Engineering, 51(2):223–256.

[12] A. Jaime O. Diaz., 1997, *Exact: An extensible approach to active object-oriented databases.*, International Journal on Very Large Database, 6(4):282–295.

[13] A. Pesonen and A. Wolski.,2000, *Quantified and temporal fuzzy reasoning for active monitoring in rapidbase.*, In Proceedings of International Symposium on Tool Environments and Development Methods for Intelligent Systems, pages 227–242.

[14] T. Henzinger and R. Alur., 1990, *Real-time logics: Complexity and expressiveness.*, In Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science, pages 390–401.

[15] Y. Saygin, O. Ulusoy, and A. Yazici., 1999, *Dealing with fuzziness in active mobile database systems.*, In Information Sciences, volume 120, pages 23–44.

[16] A. Sistla and O. Wolfson., 1995, *Temporal conditions and integrity constraints in active database system.*, In roceedings of ACM SIGMOD International Conference on Management of Data, pages 269–280.

[17] M. Stonebraker and G. Kemnita. The postgres next-generation database management system. *Communications of the ACM*, 34(10):78–92.

[18] P. Thati and G. Rosu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, March 2004.

[19] J. Widom and S. Finkelstein., 1990, *Set oriented production rules in relational database systems.*, In Proceedings of SIGMOD International Conference on Management of Data, pages 259–270.