情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-ARC-195 No.28
Vol.2011-OS-117 No.28
2011/4/14

Regular Paper

Inter-OS Communications for a Real-Time Dual-OS Monitor

Daniel Sangorrin,[†1] Shinya Honda[†1] and Hiroaki Takada[†1]

Virtualization solutions that support collaboration among its guest OS enable the development of safe complex software architectures. Most studies addressing inter-OS communications focus on throughput improvements for hypervisors targeted at server consolidation or cluster computing. However, research on inter-OS communications for hard real-time virtualization, which requires different communication patterns and predictable latencies rather than high throughput, is still scarce. We present a novel approach to inter-OS communications based on a globally scheduled real-time dual-OS monitor that enables specifying the priority and bandwidth of each communication channel or port. Our approach takes advantage of memory and time isolation asymmetries to implement preemptable low-overhead communications. Three real-time communication patterns are supported: lock-free unqueued messages, priority-based message queues and shared memory. The architecture was implemented and compared to previous approaches to demonstrate its advantages for hard real-time virtualized systems.

1. Introduction

The increasing sophistication and complexity of modern embedded systems has created a new requirements conflict. On one hand, the need to leverage existing general-purpose OS (GPOS) kernel and libraries to address all this complexity is already indisputable. On the other hand, most embedded systems still need to perform activities with special requirements (e.g., security, certifiability or timeliness) that only a low-scale real-time embedded OS (RTOS) can satisfy[1]. As an example, security holes could be exploited by local or remote attackers in order to obtain sensitive information or disturb the operation of the devices controlled by the compromised machine.

†1 Graduate School of Information Science, Nagoya University

In order to consolidate a GPOS and an RTOS on the same embedded system, two fundamental approaches exist. One of them consists of using separated hardware at the cost of additional hardware. A second approach is to virtualize existing hardware resources through a real-time dual-OS monitor[2)–5)] which allows better flexibility and reduced hardware costs. Although the main goal of real-time virtualization is to provide memory and time isolation, communication between both OS is still needed for many applications. For example, the GPOS is often used to offer an improved user interface to the status of sensors and activities managed by the RTOS. Inter-OS communication (IOC), which share many concepts from Inter-Process Communications (IPC), has been the subject of research of several works[7),?),6)]. However, most of them focus on enterprise virtualization, where the main goal is to increase the communication througput rather than providing real-time predictability. A recent state-of-the-art survey[7)] mentions the following main obstacles in current IOC mechanisms:

- Long communication overheads.
- Lack of communication awareness in the CPU scheduler.
- Absence of real-time inter-OS interactions.

Although low-overhead is considered a good property for any virtualization solution, communication awareness in the scheduler and real-time interactions are specially important for real-time dual-OS monitors. Regarding to that, we observed the following issues in current real-time dual-OS approaches:

- Current IOC mechanisms often require several copies of the same message before it is finally delivered[8)] causing an overhead that can be avoided if we take into account memory access asymmetries (i.e., the RTOS has access to all the GPOS physical memory while the inverse is forbidden).
- Most real-time dual-OS monitors schedule the GPOS as the RTOS lowest priority task[2),9),10)], and therefore it is hard to provide low IOC latency bounds when the RTOS workload increases.
- IOC is often implemented inside a non-preemptable monitor which is accessed through system calls[11)]. The non-preemptability of the monitor increases the worst latency of RTOS interrupt handlers and tasks. As a result, the size of the message being transmitted must be limited or otherwise the time isolation will be broken.

This paper addresses these issues through a novel approach to IOC that includes the following contributions:

- A mechanism that takes advantage of memory asymmetries to implement low-overhead (zero-copy in the best case) communications. Three real-time communication patterns are supported: lock-free unqueued messages, priority-based message queues and shared memory.
- A communication architecture that is preemptable by activities with higher priority. In particular, RTOS interrupts do not need to be disabled and priority-inversion is avoided through lock-free algorithms and priority ceiling.
- We leveraged a globally scheduled dual-OS monitor to enable specifying the priority and bandwidth of each communication channel to enhance the latency of messages with high priority. Execution overruns are controlled in order to protect the RTOS predictability from GPOS misbehaviors.

The paper is organized as follows. Section 2 reviews briefly the architecture of SafeG, the dual-OS monitor used by the implementation. Section 3 states the requirements for inter-OS communications. Section 4 describes the architecture and execution flow of the proposed approach. Finally, Section 5 compares this study with previous work and Section 6 draws conclusions and discusses future work.

2. SafeG: a real-time dual-OS monitor

2.1 Overview

SafeG (*Safety Gate*) is a high-reliability dual-OS monitor originally presented in 12), and designed to enable efficient concurrent execution of an RTOS (TOPPERS/ASP) and a GPOS (GNU/Linux) on top of a single embedded processor. SafeG takes advantage of ARM TrustZone security extensions[13),14)] which introduce the concept of *Trust* and *Non-Trust* states to provide a virtual environment for each OS. Trust state provides similar behavior to existing privileged and user-mode levels as in other ARM processors. On the other hand, code running under Non-Trust state (even in privileged mode) cannot access memory or devices which were allocated for Trust state usage, nor can it execute certain instructions that are considered critical. TrustZone state is controlled under a new mode called Secure Monitor mode. Switching between Trust and Non-Trust state is performed

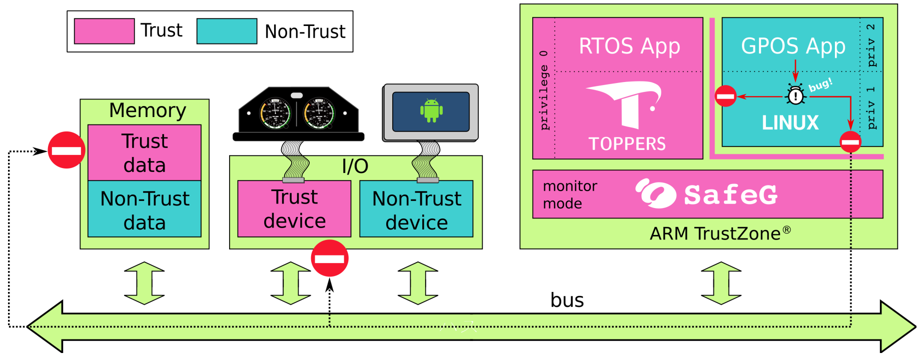

Fig. 1 SafeG: dual-OS monitor based on ARM TrustZone

under Security Monitor mode by SafeG with all interrupts disabled.

The overall organization of an embedded system based on SafeG is depicted in **Figure 1**. Memory and devices that need to be used in isolation by the RTOS are configured to be accessible only from Trust state. The remaining resources are configured to be accessible both from Trust and Non-Trust state. Time isolation of the RTOS is supported by carefully allocating the two existing types of interrupt. FIQ interrupts are assigned to the RTOS and IRQ interrupts are assigned to the GPOS. When the processor is running in Trust state, GPOS interrupts remain disabled to avoid them disturbing the execution of the RTOS. The GPOS can only execute once the RTOS sends an explicit request to SafeG using a Secure Monitor Call (SMC) instruction. On the other hand, when the processor is running in Non-Trust state, RTOS interrupts are always enabled. If an FIQ interrupt arrives (e.g., the RTOS system tick) SafeG traps it and returns the control of the CPU to the RTOS. ARM TrustZone is configured to prevent the GPOS from disabling RTOS interrupts.

2.2 Global scheduling

The original implementation of SafeG[12)] scheduled the GPOS at the RTOS idle priority. Although idle scheduling preserves the RTOS determinism, at the cost of decreasing the GPOS responsiveness. In 15) a new approach that combines global scheduling with overrun control capabilities was proposed. As **Figure 2** shows, the approach allows mixing the priorities of RTOS and GPOS activities to

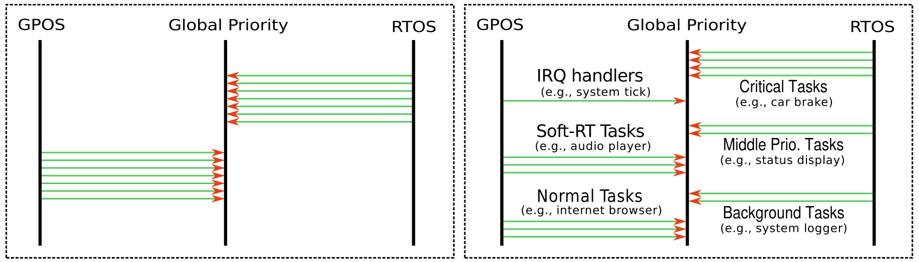

Fig. 2 Idle scheduling vs Global scheduling

provide better response times to the GPOS. For instance, the GPOS system tick could be configured with a global priority higher than non-critical RTOS task, such as the system logger. **Figure 3** shows that the RTOS application-level is divided in 2 parts: the user application and a *latency library*. This allows RTOS and GPOS applications to be implemented independently. The library consists of the following elements:

- *BTASK*: a single task running at RTOS background priority. When scheduled it calls SafeG to switch to the GPOS.
- *LTASK*: similar to the BTASK task, the main function of an LTASK task is starting a switch to Non-Trust state. The number of LTASK tasks can be configured by the user and each of them is mapped to a group of GPOS interrupt handlers and/or tasks. LTASK tasks can run at any RTOS priority and have an associated execution budget and period.
- *LTASK manager*: a manager task that decides which LTASK task should be suspended or resumed depending on the current GPOS execution priority and the availability of execution budget.
- *Latency handlers*: a set of RTOS handlers used to inform the LTASK manager that the GPOS execution priority has changed. This may happen after a GPOS task is switched or a GPOS interrupt arrives.
- *Budget handlers*: periodic time handlers to replenish the budget of each LTASK task. This replenishment strategy, commonly known as deferrable servers, allows for a simple and efficient implementation.
- *Overrun handler*: a handler executed when the running LTASK task exceeds the execution budget assigned to it.

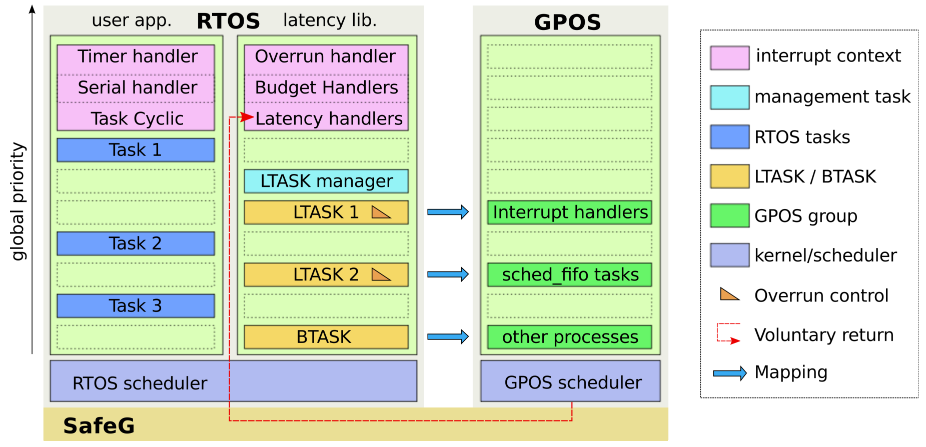

Fig. 3 Global scheduling architecture

The mentioned elements collaborate to accomplish global priority scheduling. It is important to note that the architecture does not modify the priority order at which tasks and interrupts are executed inside the GPOS. Consequently, activities assigned to a certain LTASK task are supposed to have higher priority than activities assigned to a lower priority LTASK task. In particular, if several GPOS interrupts are assigned to different LTASK tasks, then they must be configured with different hardware priorities as well. For the same reason, when the budget of a certain LTASK task expires, the budget of the next LTASK task in decreasing priority order is inherited. In other words, the neat mapping between GPOS activities and LTASK tasks depicted in Fig. 3 holds provided that the GPOS activities do not overrun the budget assigned to them. Otherwise a lower priority LTASK task is resumed to avoid disturbing the execution of higher priority RTOS tasks.

3. SafeG Inter-OS communications requirements

This section defines the requirements that the Inter-OS communications system must satisfy, through a formal list of statements that are feasible, testable and consistent with each other. They are not only applicable to SafeG but to any other real-time dual-OS monitor as mentioned in 16).

(1) It must provide mechanisms and appropriate interfaces for the most common real-time communication patterns.
(2) Ability to specify a global priority for the transmission of messages. Communications must be predictable and deterministic.
(3) Memory isolation and the integrity of the control structures used during the inter-OS communications must be preserved.
(4) The worst-case response time of RTOS interrupt handlers and tasks must remain bounded even if the GPOS misbehaves.
(5) The overhead incurred must be as low as possible.
(6) Modifications to the RTOS and GPOS must be small and maintainable.
(7) SafeG monitor modifications must be minimized.

Requirement (1) is the main functional requirement and consists of providing capabilities to communicate RTOS and GPOS tasks using communication patterns that are suitable for real-time interactions.

Requirement (2) refers to the need of a bounded message latency and the ability to distinguish messages with higher priority from those which are less critical. Missing a deadline may affect the quality of the system but is not catastrophic.

Requirement (3) is defined to make sure that sharing data with the GPOS (a Non-Trusted domain) does not compromise the security of the RTOS.

Requirement (4) is necessary to preserve the hard real-time properties of the RTOS even when code running in Non-Trust state does not collaborate, is defective or is even trying deliberately to disturb the execution of the RTOS.

Requirement (5) means that the IOC overhead must be kept as low as possible not to affect the overall performance of the system. Otherwise the practical applicability of the proposed method would be negatively affected.

Requirement (6) is extremely important to reduce maintenance costs. In especial, a GPOS kernel is large and evolves very rapidly, and therefore keeping the implementation independent from the GPOS kernel is vital for maintainability across different versions of the kernel.

Finally, requirement (7) refers to the fact that the SafeG monitor is the cornerstone of the system's overall reliability. Its code must be very small, easy to verify and runs with all interrupts disabled. Therefore, a solution that does not require extensions to SafeG monitor is the most suitable.

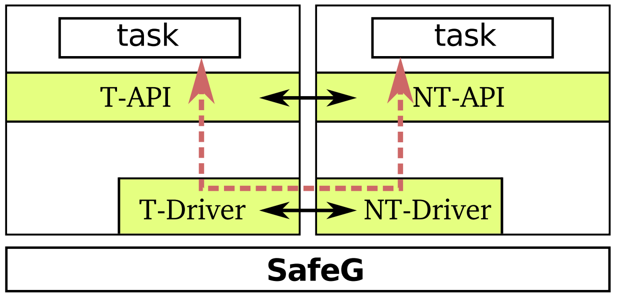

Fig. 4 Inter-OS communications APIs and drivers

4. SafeG Inter-OS communications

4.1 Inter-OS interfaces

The proposed IOC architecture offers two asymmetric application interfaces (see **Figure 4**) which provide support for the following patterns:

- *Unqueued messages*: these are unidirectional messages representing information sent by a publisher task and received by several subscriber tasks. Since they are unqueued, they are appropriate for situations where only the last status or value of the message information is important.
- *Priority message queues*: they represent bidirectional channels where messages can be queued and carry a certain priority. They are useful for event-triggered communications and synchronization.
- *Shared memory (shm)*: these are blocks of Non-Trust memory shared by both OS. The minimal size of one block is 1 memory page. Shared memory is useful for transmitting a bigger amount of data and can be used in combination with lock-free synchronization or the mentioned message queues.

Table 1 shows the main functions of the proposed IOC interface for both OS. The interface for TOPPERS/ASP follows the μITRON[17] function interface style (operation + object) with the addition of the `safeg_` prefix. The interface for Linux tasks just reuses current POSIX IPC interfaces (POSIX message queues and shared memory). Creation of communication objects (i.e., memory buffers and queues) is performed through a static API in the case of ASP, and by the Linux IOC driver at initialization time in the case of Linux. Each queued channel

Table 1 TOPPERS/ASP vs Linux interfaces

Pattern	ASP	Linux	Explanation
Unqueued	`SAFEG_CRE_UNQ`	`mq_open("/safeg_unq_0"..)`	create a message port
Unqueued	`safeg_wri_unq`	`mq_send`	write an unqueued message
Unqueued	`safeg_rea_unq`	`mq_receive`	read unqueued message
Queued	`SAFEG_CRE_PMQ`	`mq_open("/safeg_que_0"..)`	create message queue
Queued	`safeg_snd_pmq`	`mq_send`	send a message
Queued	`safeg_rcv_pmq`	`mq_receive`	receive a message
Shm	`SAFEG_CRE_SHM`	-	create a shared memory block
Shm	`get_shm`	`mmap`	get a shared memory block

or unqueued port is identified by a string consisting of three elements: the `safeg_` prefix; `que` or `unq`; and a natural number. The internal drivers are in charge of making sure they refer to the same object. Parameters for message communication (sending and receiving) functions are the same for both OS; except Linux's priority parameter for unqueued messages, which is left unused. This and the fact that messages can be overwritten make the behavior of unqueued message queues significantly different from the standard POSIX message queues. Since queueing message functions can block, each of them has a timeout and a polling version (e.g., `mq_timedreceive`). Shared memory resides in Non-Trust memory and is reserved by the kernel at initialization. ASP tasks just need to obtain the physical address of the block for a given identifier. In case of Linux tasks, they have to memory-map the corresponding block to the process virtual address space. A shared memory block is identified by an offset though we provide a wrapper function `safeg_map_shmregion(id,&va)` to simplify its usage. The implementation must make sure that shared memory blocks are cache coherent.

4.2 Architecture elements

IOC interfaces are implemented through a driver module inside of each kernel, as we see in Fig. 4. **Figure 5** shows an overview of the architecture for bidirectional inter-OS channels. We see that a message queue is defined for message reception at each OS. Queues for sending messages are not required since we will write directly into the reception message queues. Each OS also counts with a software module capable of sending and waiting for inter-OS asynchronous events. They are necessary to implement the blocking interface of message queues and work in a similar way as software interrupts.

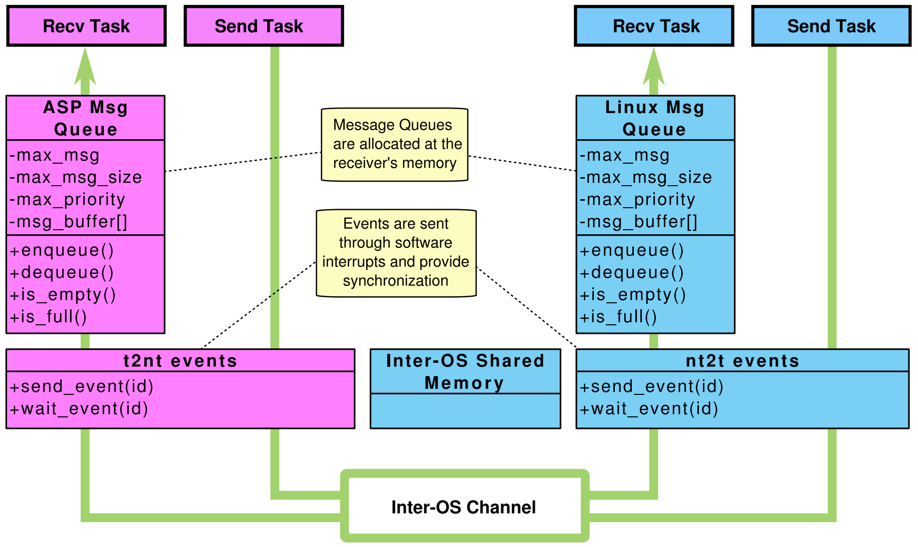

Fig. 5 Inter-OS communication channels

Between both OS there is an inter-OS shared memory area used to store the communication objects (i.e., unqueued ports, message queues and shm objects) and a few data structures. These structures are depicted in **Figure 6** and include:

- *Global scheduling data*: these are two variables (`nt_prio` and `ltask`) required for the implementation of global scheduling.
- *Shmem block*: shared memory block size and permissions.
- *Unqueued buffer variables*: apart from the size of each buffer, we have variables (i.e., `timestamp`, `current_buffer` and `shadow_buffer`) required for lock-free synchronization (see 4.4).
- *Event-related data*: these are variables to indicate that a channel has a pending event or a task is waiting.
- *nt2t message*: this variable stores a pointer to a Linux task message being sent to ASP. Since the Linux kernel cannot access ASP's memory, a pointer is placed in shared memory and an ASP handler is raised to perform the copy instead. If a task is waiting at the queue, the message can be copied directly into the task's buffer to reduce the overhead (zero-copy).

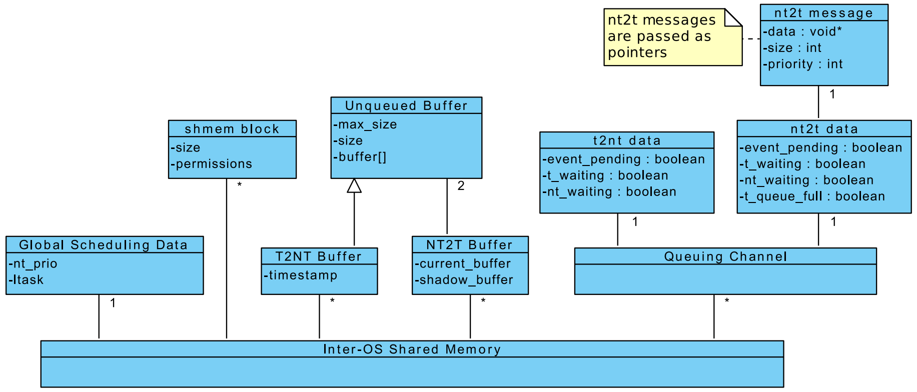

Fig. 6 Inter-OS communications shared data structures

The configuration of all these elements is performed statically at initialization and cannot be changed dynamically. Since inter-OS shared memory can be filled with garbage at any time by a compromised Linux system, the range of each variable must be checked by ASP before its use.

4.3 Queued messages flow

The key to understanding the flow of IOC queued messages is the fact that we are just trying to emulate the flow of a local IPC message as close as possible. The main obstacles that need to be solved are mutually exclusion (e.g., Linux cannot disable RTOS interrupts); asymmetric memory access permissions (e.g., Linux cannot access to the RTOS memory queues directly); and synchronization.

Figure 7 shows the communication flow of a queuing message sent by an ASP task to a Linux task. Transmission from several ASP tasks to the same channel must be serialized through a local mutual exclusion mechanism. First, the status of the reception buffer is checked (e.g., by inspecting, and range-checking, its head and tail variables). If the buffer is full the task blocks and waits for a new event (i.e., a message being dequeued) or alternatively, until a timeout expires. Additionally a shared variable (i.e., `t_waiting`) is set to true in order to indicate that an ASP task is waiting. In case the queue was not full, the message is enqueued in the Linux buffer. The priority message queue is composed of several FIFOs and is accessed by only one writer and one reader at a time. Therefore,

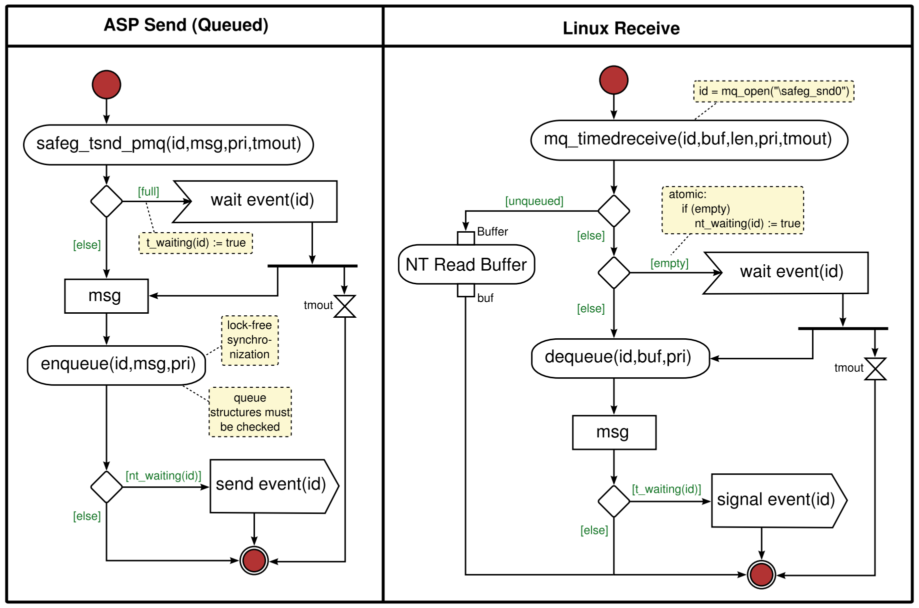

Fig. 7 Queuing communication flow from ASP to Linux tasks

a lock-free algorithm can be used to insert/extract messages instead of more complex mutual exclusion algorithms. Once the message is inserted, the shared variable `nt_waiting` of that channel is consulted. If it is true, we send an event (i.e., indicate that a message was enqueued) to Linux. On the receiving end, when a Linux task tries to receive a message we first need to evaluate whether it is accessing an unqueued port or a message channel. This happens because the interface is the same for both unqueued messages, which are explained in 4.4, and queued ones. After that, we check whether the queue is empty or not. If it is empty we need to wait for an event from ASP (i.e., a message being enqueued) and set the shared variable `nt_waiting` to true. The variable must be set atomically to avoid racing conditions. Alternatively, the wait can be based on kernel semaphores. If the queue was not empty, we just have to read the message into the Linux task's buffer. After the message is dequeued, the last operation is waking up any possible task waiting in the RTOS.

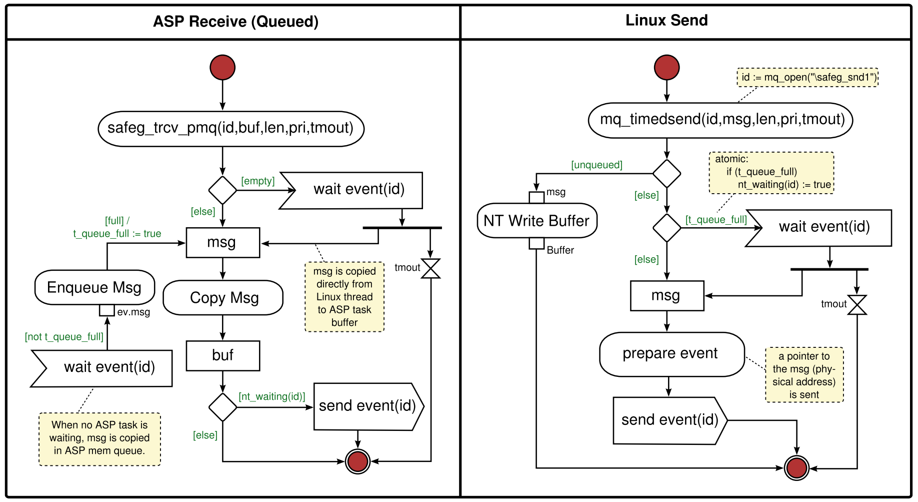
ASP Receive (Queued)
Linux Send
safeg_trcv_pmq(id,buf,len,pri,tmout)
[empty]
wait event(id)
[else]
[full] / t_queue_full := true
msg
Enqueue Msg
ev.msg
Copy Msg
msg is copied directly from Linux thread to ASP task buffer
tmout
[not t_queue_full]
wait event(id)
buf
[nt_waiting(id)]
send event(id)
[else]
When no ASP task is waiting, msg is copied in ASP mem queue.
id := mq_open("\safeg_snd1")
mq_timedsend(id,msg,len,pri,tmout)
[unqueued]
atomic: if (t_queue_full) nt_waiting(id) := true
msg
NT Write Buffer
[else]
Buffer
[t_queue_full]
wait event(id)
[else]
msg
tmout
prepare event
a pointer to the msg (physical address) is sent
send event(id)

Fig. 8 Queuing communication flow from Linux to ASP tasks

Figure 8 shows the communication flow of a queuing message sent by a Linux task to an ASP task. The flow is quite similar to the one described above except for the following fact: since a Linux sender cannot write to ASP's buffer directly, a pointer to the user's message (physcal address) is placed in shared memory and an event is sent to ASP. An ASP handler or task is in charge of copying the message either to ASP's message queue or directly to the buffer of an ASP waiting task. Notice that in the best case, the transmission will occur with zero-copy overhead, which paradoxically may result in certain IOC calls having less overhead than local IPC calls.

4.4 Unqueued messages flow

Unqueued messages consist of a shared buffer where the latest value of a certain variable or information is stored. They are unidirectional, either from trust to Non-Trust (t2nt) or viceversa (nt2t). A typical application could consist of a task sharing periodically the value of a sensor controlled by the RTOS and another task displaying it through a graphical interface in the GPOS. Our proposal takes advante of memory and time isolation asymmetries to provide lock-free algorithms for both directions.

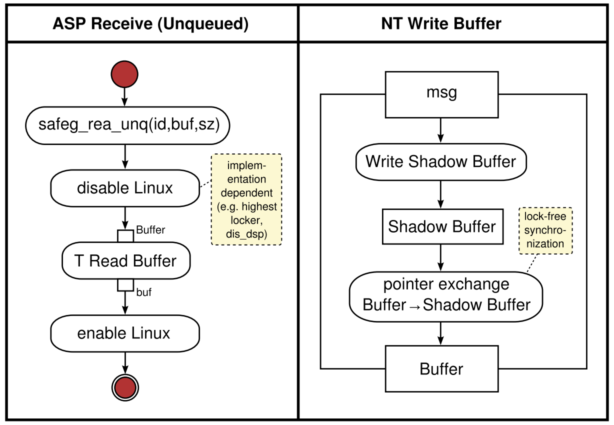

Fig. 9 Unqueued communication flow from Linux to ASP tasks

Figure 9 shows the flow for a Non-Trust to Trust unqueued message. When a Linux task needs to write a new value in the message buffer, it does it first on a temporary ”shadow” one to avoid coherence issues (e.g., in case it was preempted by a reader when the message were half-written). Once the message is complete the pointers to both buffers are exchanged (this operation is not required to be atomic). On the ASP side, when a task wants to read a message, it first must disable the execution of Linux (for example, by rising the priority of the task). This is necessary to avoid its preemption by a Linux task with a higher global priority. The main advantage of this mechanism is that we do not need to disable all interrupts, and therefore the latency of RTOS interrupt handlers and high priority tasks is not affected.

Figure 10 shows the flow for a Trust to Non-Trust unqueued message. Since Linux cannot disable the operation of the RTOS before reading the buffer, a lock-free mechanism based on timestamps (e.g., a value that always increases) has been designed. When a Linux task starts reading an unqueued message, it first records the value of the initial timestamp. Once the read is finished, it compares it with the current timestamp. In case they differ, then it means that an RTOS writer task preempted Linux overwriting the message and it has to retry. The algorithm is not wait-free for Linux tasks but that should not be a problem

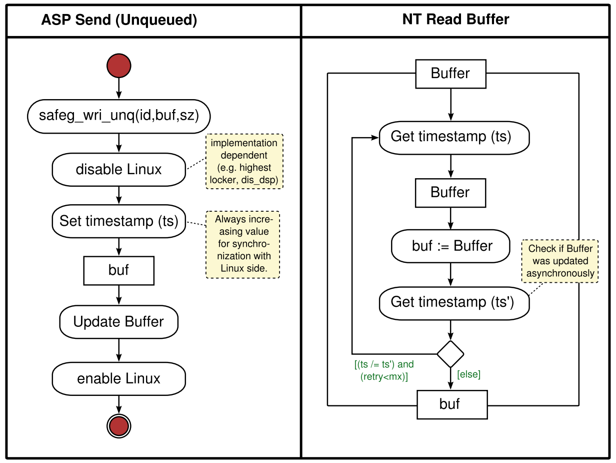

Fig. 10 Unqueued communication flow from ASP to Linux tasks

since ASP tasks are trusted not to cause starvation. The main advantages of this algorithm are its lock-free nature and the fact that it does not require disabling the RTOS interrupts.

5. Related work

There is a notable amount of literature proposing different IOC mechanisms to enable collaboration among the guest OS of a virtualized system. 7) is a good survey of the state-of-the-art in IOC mechanisms with numerous references. Most previous work can be classified in the following groups:

- *Throughput-oriented optimizations*: many works in this group study methods to improve the performance of communications in the XEN hypervisor[18]. As an example, XenSocket[6] was presented to replace the Xen page-flipping mechanism with a static circular memory buffer shared between two domains to construct a high-throughput IOC mechanism. Similar works can be found in 19)–21) for example. Each approach offers varying trade-offs between performance and transparency.
- *Scheduling-aware optimizations*: In most virtualization approaches, the CPU scheduler is unaware of the communication requirements or the internal priority of its guest OS, which has a major influence on the IOC latency. A few works have addressed this problem[22),23] by exploiting the statistics of the guest OS to influence scheduling decisions. Even Xen's credit scheduler[24] provides a mechanism to boost guest OS that are considered I/O bound.
- *Communications in real-time dual-OS monitors*: RTAI[10] provides several interfaces and mechanisms for IOC. This is facilitated by the fact that the RTOS and the Linux kernel are linked together sharing the same memory space. 9) is capable of providing better memory isolation than RTAI and uses RTEMS message queues as the means of communication. However, both share the problem of long IOC latency bounds because they schedule Linux as the RTOS idle task[2]. XtratuM[11] is a hypervisor for safety critical applications which enforces time isolation by scheduling each guest OS following a certain cyclic pattern. Although communication through FIFO queues has been proposed[8], XtratuM currently implements sampling and queuing ports as described in the ARINC653 standard[25]. The main problem in the current implementation is that XtratuM is not preemptable, and therefore a message transmission ocurring at the end of a timeslot may break the temporal isolation.

Compared to previous works, our proposal is based on a globally scheduled dual-OS monitor (i.e., aware of the internal priority of its guest OS), and provides tight integration and low-overhead without compromising the memory and time isolation of the RTOS interrupt handlers and tasks.

6. Conclusions and future work

We proposed a novel approach to IOC focused on real-time dual-OS monitors as opposed to throughput-oriented IOC mechanisms present in enterprise hypervisors. Our approach takes advantage of memory and time isolation asymmetries in the dual-OS model to implement preemptable low-overhead communications and three real-time communication patterns are supported: lock-free unqueued messages, priority-based message queues and shared memory. We leveraged our previous work on a globally scheduled real-time dual-OS monitor that enables specifying the priority and bandwidth of each communication channel or port. The architecture was compared to previous approaches to demonstrate its advantages for hard real-time virtualized systems. As future work, we plan to evaluate the current implementation of the system and study its application to a multi-core version of SafeG.

References

1) Heiser, G.: The Role of Virtualization in Embedded Systems, *1st Workshop on Isolation and Integration in Embedded Systems*, Glasgow, UK, ACM SIGOPS, pp. 11–16 (2008).
2) Takada, H., Iiyama, S., Kindaichi, T. and Hachiya, S.: Linux on ITRON: A Hybrid Operating System Architecture for Embedded Systems, *SAINT-W '02: Proceedings of the 2002 Symposium on Applications and the Internet (SAINT) Workshops*, Washington, DC, USA, IEEE Computer Society, pp.4–7 (2002).
3) Cereia, M. Bertolotti, I.: Asymmetric virtualisation for real-time systems, *ISIE 2008*, Cambridge, pp.1680 – 1685 (2008).
4) Yoo, S., Liu, Y., Hong, C.-H., Yoo, C. and Zhang, Y.: MobiVMM: a virtual machine monitor for mobile phones, *MobiVirt '08: Proceedings of the First Workshop on Virtualization in Mobile Computing*, New York, NY, USA, ACM, pp.1–5 (2008).
5) Heiser, G.: Hypervisors for consumer electronics, *CCNC'09: Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, Piscataway, NJ, USA, IEEE Press, pp.614–618 (2009).
6) Zhang, X., McIntosh, S., Rohatgi, P. and Griffin, J. L.: XenSocket: a high-throughput interdomain transport for virtual machines, *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, New York, NY, USA, Springer-Verlag New York, Inc., pp.184–203 (2007).
7) Wang, J.: Survey of State-of-the-art in Inter-VM Communication Mechanisms (2009).
8) Shuwei, B., Yiqiao, P., Kairui, S., Qingguo, Z., Nicholas, M. and Lian, L.: XM-FIFO: Interdomain Communication for XtratuM, *9th Real-Time Linux Workshop* (2007).
9) Beltrame, G., Fossati, L., Zulianello, M., Braga, P. and Henriques, L.: xLuna: a Real-Time, Dependable Kernel for Embedded Systems, *IP-SOC: IP based electronics system conference and exhibition* (2010).
10) RTAI: Official website. `https://www.rtai.org/`.
11) Masmano, M., Ripoll, I., Crespo, A. and Metge, J.: XtratuM: a Hypervisor for Safety Critical Embedded Systems, *11th Real-Time Linux Workshop*, Dresden, Germany (2009).
12) Nakajima, K., Honda, S., Teshima, S. and Takada, H.: Enhancing Reliability in Hybrid OS System with Security Hardware, *The IEICE Transactions on Information Systems*, Vol.93, No.2, pp.75–85 (2010-02-01).
13) ARM Ltd.: *ARM Security Technology. Building a Secure System using TrustZone Technology, PRD29-GENC-009492C* (2009).
14) ARM Ltd.: *ARM1176JZF-S. Technical Reference Manual, DDI 0301G* (2008).
15) Sangorrin, D., Honda, S. and Takada, H.: Real-Time Global Scheduling for a High-Reliability Dual-OS Monitor, Technical report, University of Nagoya (2011).
16) Armand, F. and Gien, M.: A practical look at micro-kernels and virtual machine monitors, *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, CCNC'09, Piscataway, NJ, USA, IEEE Press, pp.395–401 (2009).
17) Takada, H. and Sakamura, K.: "μITRON for small-scale embedded systems", *IEEE Micro*, vol. 15, pp. 46-54, Dec. 1995.
18) Chisnall, D.: *The definitive guide to the xen hypervisor*, Prentice Hall Press, Upper Saddle River, NJ, USA, first edition (2007).
19) Li, D., Jin, H., Shao, Y., Liao, X., Han, Z. and Chen, K.: A High-Performance Inter-Domain Data Transferring System for Virtual Machines, *JSW*, Vol.5, No.2, pp.206–213 (2010).
20) Chen, H., Shi, L. and Sun, J.: VMRPC: A high efficiency and light weight RPC system for virtual machines, *Quality of Service (IWQoS), 2010 18th International Workshop on*, pp.1 –9 (2010).
21) Diakhaté, F., Perache, M., Namyst, R. and Jourdren, H.: Euro-Par 2008 Workshops - Parallel Processing, Springer-Verlag, Berlin, Heidelberg, chapter Efficient Shared Memory Message Passing for Inter-VM Communications, pp.53–62 (2009).
22) Kim, H., Lim, H., Jeong, J., Jo, H. and Lee, J.: Task-aware virtual machine scheduling for I/O performance., *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, New York, NY, USA, ACM, pp.101–110 (2009).
23) Govindan, S., Nath, A.R., Das, A., Urgaonkar, B. and Sivasubramaniam, A.: Xen and co.: communication-aware CPU scheduling for consolidated xen-based hosting platforms, *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, New York, NY, USA, ACM, pp.126–136 (2007).
24) Cherkasova, L., Gupta, D. and Vahdat, A.: Comparison of the three CPU schedulers in Xen, *SIGMETRICS Perform. Eval. Rev.*, Vol.35, pp.42–51 (2007).
25) ARINC-653: *Airlines Electronic Engineering Committee, 2551 Riva Road, Annapolis, Maryland 21401-7435. Avionics Application Software Standard Interface* (1996).