Design and Implementation of the Multiprocessor Bandwidth Inheritance Protocol on Linux

Andrea Parri[(a)]
a.parri@sssup.it

Juri Lelli[(a)]
j.lelli@sssup.it

Mauro Marinoni[(a)]
m.marinoni@sssup.it

Giuseppe Lipari[(a),(b)]
giuseppe.lipari@lsv.ens-cachan.fr

[(a)] Scuola Superiore Sant'Anna, Pisa, Italy
[(b)] Ecole Normale Suprieure, Cachan, France

Abstract

The Multiprocessor Bandwidth Inheritance (M-BWI) is a synchronisation protocol for coordinating mutually exclusive access to critical sections in multi-processor real-time systems. It combines resource reservations techniques with the priority inheritance access protocol, so to guarantee temporal isolation between non-interacting tasks and real-time execution. In the original paper, the authors proposed an implementation in LITMUS-RT, a version of Linux specifically designed for research purposes.

In this paper we propose an implementation of the M-BWI protocol on the Linux OS augmented with the SCHED_DEADLINE patch. We describe the architecture of our implementation, highlighting the problems we found and the possible solutions. We also run an extensive set of experiments for evaluating the overhead of the proposed implementation.

1 Introduction and motivation

Demand for supporting real-time execution in GPOS has substantially increased in the past years. Many application require some sort of real-time execution, e.g. real-time audio processing, multimedia processing, telecommunication (routers, etc.) interactive games, etc.

Therefore, many developers have tried to improve the real-time proprties of the Linux OS by providing additional features. For example, the *CONFIG_PREEMPT_RT* patch [7] makes almost every routine in the kernel preemptible, so to reduce the blocking time that higher priority activities suffer in the stock Linux kernel. To further reduce priority inversion, the patch also introduces the priority inheritance protocol [19] for accessing mutually exclusive critical sections in the kernel.

Parallel efforts have been directed to providing alternative real-time scheduling policies in the kernel, and in particular *resource reservation* [17] mechanisms. A resource reservation scheduler allows the user to *reserve* a fraction of the computing resources

to a thread, so that the thread is guaranteed to always receive that amount of computation time. At the same time, the thread cannot use more than reserved. Hence, resource reservation mechanisms provide *temporal isolation*: each thread executes approximately as it were running on a slower dedicated processor of speed equal to the reserved fraction of the processor utilisation. One example of such scheduling mechanism is the Constant Bandwidth Server (CBS) [3], which has been implemented in the SCHED_DEADLINE patch [14] for the Linux kernel. The basic idea is that each thread is assigned a budget Q and a period P, and it is guaranteed to receive up to Q units of computation every period P.

Unfortunately, resource reservation mechanisms are incompatible with classical resource access protocols like priority inheritance. This has been shown in [16]: if the budget is exhausted while the thread is executing inside a critical section, the priority inversion of blocked tasks can grow arbitrarily large. The authors proposed an extension of the priority inheritance protocol, called Bandwidth Inheritance BWI [16], in which the thread that holds the lock inherits not only the priority but also the budget of the blocked tasks. The protocol has recently beed extended to multi-processor systems, thus becoming the Multi-processor Bandwidht Inheritance (M-BWI) protocol [8], and it has been implemented in LITMUS-RT [6], a research-oriented OS designed to make it easy to implement and evaluate new scheduling policies. However, LITMUS-RT is not particularly optimized for use in production systems, and it is not up-to-date with respect to current Linux OS releases.

In this paper we propose an implementation of the M-BWI protocol in Linux patched with SCHED_DEADLINE. We explain the architecture and the practical aspects of our implementation. Also, we present the problems that we encountered, and a few solutions to them. Finally, we present the evaluation of the overhead introduced by the protocol.

2 Related Work

As discussed in the introduction, as lot of real-time enhancements to Linux have been proposed in the past. Among the many, we wish to cite LITMUS-RT [6], a patch to the Linux kernel proposed by researchers at the University of North Caroline at Chapel Hill, that makes it easy to implement new scheduling algorithms. The patch is mostly used in research laboratories for comparing different real-time scheduling policies.

The concept of bandwidth inheritance has been proposed in [16], but also independently by researchers at the Technical University of Dresden in their real-time micro-kernel DROPS [2, 12]. The idea is to let the task inherit not only the priority but also the budget.

Inheritance can be effectively implemented in single processor systems using the concept of *shadow task*, or *proxy execution*. The first has been proposed in the SHARK kernel [9] for seamlessly implementing the priority inheritance policy when combining different scheduling policies. The same implementation can be used as a basis for implementing BWI; however it cannot easily be extended to multi-core systems.

3 Background

3.1 SCHED_DEADLINE

SCHED_DEADLINE is a new scheduling class for the Linux scheduler. It implements the Earliest Deadline First (EDF) real-time scheduling algorithm and uses the Constant Bandwidth Server (CBS) resource reservation scheduling technique to provide *temporal isolation*, among non interacting tasks.

SCHED_DEADLINE can handle:

- periodic tasks (typical in real-time and control applications;
- sporadic tasks (typical in soft real-time and multimedia applications;
- aperiodic tasks.

Scheduling entities of this scheduling class correspond to CBS *servers*. Each server includes original and actual scheduling parameters, basically *runtime* and *deadline*. The former are copied in when the user sets a task to use SCHED_DEADLINE policy, the latter are initialized with original values and then continuosly updated during task execution. Actual *runtime* is decremented with `HZ` frequency and represents remaining runtime for the current job of a task, actual *deadline* is updated following CBS rules and corresponds to a job absolute deadline (used to perform EDF scheduling among active tasks instances).

Several events can happen during a task execution, the following is a non exhaustive list that is also useful to introduce terminology for the rest of the paper:

- *runtime overrun*, happens if the current job exceeds its allowed runtime → the active server is *throttled* (see below) and the amount of overrun is accounted for the next job instance;

- *deadline miss*, similar to the previous case, if deadlines are set equal to periods no throttling is required;

- server *throttling*, can happen as a result of the previous cases, the server is stopped (*throttled*) and a timer is set to fire at the *replenishment* instant (next server period);

- runtime *replenishment*, actual runtime is refilled and actual deadline is set as current clock plus relative deadline (original).

3.2 Model of a critical section

Tasks can access shared memory using mutually exclusive semaphores, often referred to as *mutex*es. The portion of code in a task between a *lock* and an *unlock* operation on a mutex is called *critical section*. If a task needs to enter a critical section, it may be blocked by the fact that another task has already locked the corresponding mutex: this latter task is called *lock owner*, or *lock holder*. In real-time systems it is important to compute for how long, in the worst case, a task may remain blocked on a lock.

A *priority inversion* happens when a task is blocked by a low priority task. Without a proper protocol to control access to critical sections, the duration of the priority inversion may become too long, or even unbounded; for this reason, the *priority inheritance protocol* [18] (PIP) has been proposed as a simple and effective way to reduce priority inversion. According to the PIP, when a task is blocked on a lock, the lock owner *inherits* the highest between its priority and the priorities of the blocked tasks. In this way, it cannot be preempted by medium priority tasks, thus reducing the blocking time.

Other protocols have been proposed as improvements over the PIP, notably the *priority ceiling protocol* [10], and the *stack resource policy* [4]. However, as we will briefly discuss in the following, such protocols are not adequate for use in *open systems*.

3.3 Open Systems

An *open real-time system* is a real-time system where tasks can be dynamically created and destroyed at any time. In contrast, in a *closed system*, the developer knows the number of tasks in the system and their parameters at design time.

In a closed system, a real-time analysis is performed off-line to guarantee that all tasks will meet all deadlines under all conditions. Naturally, closed systems can take advantage from the fact that everything is known at design time, including the code of the tasks. Therefore the system scheduling algorithm and the access protocol can be optimised for the specific system.

In an open system, instead, we do not know anything about the tasks at design time. Linux can be considered as an example of open system: non-real-time tasks (threads or processes) can be dynamically created at any instant. If we want to apply a similar approach to real-time systems on Linux, we must require the user to specify at least some parameters to the scheduling algorithm, so to check and guarantee real-time execution.

Therefore an *admission control* test must be performed on-line to make sure that all admitted tasks will indeed meet their deadlines. According to the *resource reservation framework*, and to the SCHED_DEADLINE scheduling algorithm, the new task requires a percentage of the computational bandwidth in the form of a *budget* Q and a period P. If task passes the admission control, the system guarantees that it will receive at least Q units of time every period P. An admission control test is nothing else than a schedulability analysis test that checks that the new incoming task maintains the system schedulable.

One example of admission control test for the EDF+CBS algorithm on single processor systems consists in checking that the total required bandwidth does not exceed the available bandwidth: in formula, $\sum_i \frac{Q_i}{P_i} \leq 1$.

When considering also mutex semaphores and critical section, the analysis becomes more complex, as it is now necessary to compute the maximum blocking time for each task. To compute the maximum blocking time, it is necessary to know the duration of the critical sections of all the tasks in the system. Once the blocking time has been computed, an example of admission control test for EDF is the following

$$\forall i, \sum_{D_j \leq D_i} \frac{Q_j}{P_j} + \frac{B_i}{P_i} \leq 1.$$

In an open system, however, it is not possible to ask the user to specify too many detailed information on *every task*, otherwise the system becomes too difficult to use and manage. In fact, typically

in a open system tasks with different levels of criticality may coexist, and asking detailed and precise information on non-critical task may make the job of the developer/user too difficult. Also, notice that an error in the specification of the blocking times may compromise the schedulability of the whole system (i.e. any task can miss its deadline).

3.4 Combining resource reservations and critical sections

When trying to combine resource reservations with a resource access protocol, we need to solve two problems. The first problem is concerned with how to take into account blocking time in the admission control formula, as described in the previous section.

The second problem is concerned with handling the situation that occurs when a task is in a critical section and its budget is exhausted. In that case, the scheduler algorithm *throttles* the task (i.e. suspends it) until the next period, when the budget is recharged to its maximum value. However, another task blocked on the semaphore must also wait for the budget to recharge. Therefore, the worst case blocking time may become very large.

An example of such situation is depicted in Figure 1, where task τ_1 suffers a long blocking time from τ_3 whose budget is exhausted at time $t = 4$ while in a critical section on mutex M.

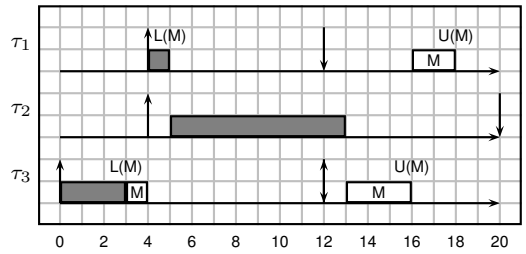


FIGURE 1: *A task exhaust its budget while in a critical section, thus increasing the blocking time of another task.*

We want to avoid this problem. At the same time, we would like to avoid the necessity to compute the blocking time of non critical tasks, and maintain the useful property of temporal isolation. So, the goals for our resource access protocol are the following

1. We shall not require the user to specify any parameter to run the task, other that the desired budget and period (Q, P);
2. *Temporal protection*: the performance of a task (i.e. its ability to meet its deadline) shall depend only on its parameters (Q, P), on its worst-case execution time and period, and on the duration of the critical sections of the tasks with which it interacts;
3. If we do know the worst-case execution times (and duration of critical sections) of the task and of all *interacting tasks*, it must be possible to compute (Q, P) such that the task will meet its deadline.
4. We want to do this on multi-core systems as well.

3.5 Interacting tasks

What do we mean with *interacting tasks*? In the simplest case, we say that two tasks interact when they access a critical section with the same mutex. However, since critical sections can be nested within each other, the general case is a little more complex.

A blocking chain from a task τ_i to a task τ_j is a sequence of alternating tasks and semaphores:

$$H_{i,j} = \{\tau_i \rightarrow R_{i,1} \rightarrow \tau_{i,1} \rightarrow R_{i,2} \rightarrow \ldots \rightarrow R_{i,\nu} \rightarrow \tau_j\}$$

such that τ_j is the lock owner on semaphore $R_{i,\nu}$ and τ_i is blocked on $R_{i,1}$. As an example, consider the blocking chain $H_{1,3} = \{\tau_1 \rightarrow R_1 \rightarrow \tau_2 \rightarrow R_2 \rightarrow \tau_3\}$, in which τ_3 accesses R_2, τ_2 accesses R_2 with a cs nested inside cs on R_1, τ_1 accesses R_1.

We say that a task τ_j **interferes** with task τ_i only if a blocking chain from τ_i to τ_j exists. We say that task τ_i is *independent* of, or *temporally isolated* from τ_j when there exist no blocking chain from τ_i to τ_j. We would like to maintain the temporal isolation property:

> The ability of a task to meet its deadlines depends only on its worst-case computation time and arrival pattern, its assigned budget and period, and the duration of the critical sections in the blocking chains starting from τ_i.

3.6 The M-BWI protocol

We informally describe here the rules of the algorithm. A more complete and detailed description can be found in [8].

- When a task is blocked on a mutex we have several cases:

- if the lock-owner is itself blocked, the blocking chain is followed until a non-blocked lock-owner is found;
- if the lock-owner is executing on another processor, the blocking task actively waits for the lock owner to release the mutex;
- if the lock-owner is not executing, then it *inherits* the budget and scheduling deadline of the blocked task

- Therefore, when holding the lock on a mutex, a task can have a list of pairs (budget,deadline) that it can use; it will always execute consuming the budget of the earliest deadline pair.
- When a task releases the mutex, it will discard the pairs of (budget,deadline) of the blocked tasks from its list.

Rather than going through a formal analysis of the protocol, we will present here one example that demonstrate how the protocol works. In Figure 2 we show the schedule produced by three tasks scheduled on 2 processors with migration. All of them access the same mutex M_1. At time $t = 5$ tasks τ_A and τ_B are executing on the two processors, and task τ_C is the lock-owner but it is not executing. At time $t = 6$ τ_B attempts to lock the mutex, so τ_C is woken up and inherits the budget and the deadline of τ_B, while this is blocked. At time $t = 9$ also τ_A attempts to lock the mutex, and since the lock owner is already executing on another processor, it starts a spin loop actively waiting for the mutex to be unlocked. At time $t = 14$ τ_C releases the lock, and the protocol uses a FIFO policy to wake up blocked tasks, so it wakes up τ_B. Finally, when at time $t = 18$ τ_B also releases the lock, τ_A stops its active waiting and starts to execute its critical section.

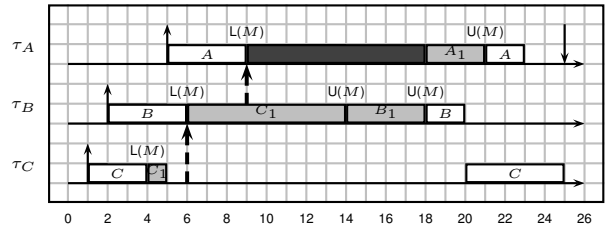


FIGURE 2: *Example of M-BWI: τ_A, τ_B, τ_C, executed on 2 processors, that access only mutex M_1.*

Given a task τ_i, the total amount of time that other tasks execute consuming the budget Q_i and that τ_i must actively wait for a lock release, is called interference time I_i. It is possible to compute an upper bound to the interference I_i by analysing all blocking chains starting from τ_i. In the general case of nested critical sections, the algorithm is rather complex: we remand the interested reader to [8] for more information.

The algorithm has two important properties:

- It guarantees **temporal isolation**: a task cannot receive interference from independent tasks;
- It is possible to compute an upper bound on interference, therefore it is possible to assign the budget Q_i to a task such that it will meets all its deadlines.

4 Implementation

Our implementation consists of about 300 lines of codes applied on top of Linux 3.10-rc1, with SCHED_DEADLINE Version 8.

Besides SCHED_DEADLINE, the work presented here is strongly based on Linux's current implementation of the Priority Inheritance (PI) algorithm. For this reason, we first proceed with a (incomplete) review of Linux's PI design and implementation (see also [1]), and then with the detailed description of our contribution. The reader which already knows about Linux's PI infrastructure may want to skip the next subsection.

The terms *task* and *process* are used interchangeably in this document.

4.1 Priority Inheritance in Linux

4.1.1 Basic structures

In the following discussion, we will be adopting the following terminology:

Mutex. A mutex semaphore which is shared by the processes and by which processes may interact and synchronise. The mutex structure contains a pointer to the owner of the mutex. If the mutex is not owned, this pointer is set to NULL.[1]

PI chain. It is the same as the *blocking chain* defined in Section 3.5, i.e. an alternating series of mutexes and processes such that each process in the chain is blocked on the next mutex in the chain (if any), and it is the owner the previous

[1]Linux actually implements a more involved mechanism that considers "Pending Ownership" and "Lock Stealing".

one in the chain (if any). The PI chain causes processes to inherit priorities from a previous process that is blocked on some of its mutexes.

Lock. A *spin-lock* that is used to protect parts of the PI algorithm. These locks disable preemption on UP and prevents multiple CPUs from entering critical sections simultaneously on SMP.

Waiter. A structure, stored on the stack of a blocked task, that holds a pointer to the process, as well as the mutex that the task is blocked on. It also contains the node structures to place the task in the right place within the PI chains. More on this below.

Top waiter The highest priority process waiting on a mutex.

Top PI waiter The highest priority process waiting on one of the mutexes that a specific process owns.

Since a process may own more than one mutex, but can never be blocked on more than one, PI chains can merge at processes. Also, since a mutex may have more than one process blocked on it, but never be owned by more than one, we can have multiple chains merge at mutexes.[2]

In order to store the PI chains, Linux adopts two priority-sorted linked lists:

Waiter list. Every mutex keeps track of all the waiters that are blocked on it by storing them in its *waiter list*. This list is protected by a lock called *wait lock*. Since no modification of the waiter list is done in an interrupt context, the *wait lock* can be taken without disabling interrupts.

PI list Each process stores the top waiters of the mutexes that are owned by the process in its *PI list*. Note that, in general, this list does not hold all the waiters that are blocked on these mutexes. The list is protected by a lock called *PI lock*. This lock may also be taken in interrupt context, so when locking the PI lock, interrupts must be disabled.

The top of the task's PI list is always the highest priority task that is waiting on a mutex that is owned by the task. So if the task has inherited a priority, it will always be the priority of the task that is at the top of this list.

4.1.2 Priority adjustments

In the implementation of the priority inheritance protocol, there are several different locations in the code where a process must adjust its priority. With the help of the *PI list* it is rather easy to know what need to be adjusted; we now describe in more detail the main functions involved in this process.

The function that is responsible for adjusting the priority of a given task is `__rt_mutex_adjust_prio`.

The function first obtains the priority that the task should have, that is either the task's own normal priority, or the priority of a higher priority process that is waiting on a mutex owned by the task. By the above discussion, this is simply a matter of comparing the priority of the top PI waiter's with the task's normal priority. The function then examines the result, and if this does not match the task's current priority, the task's scheduling class methods are called to implement the actual change in priority.

Note that `__rt_mutex_adjust_prio` can either increase or decrease the priority of the task. In the case that a higher priority process has just blocked on a mutex owned by the task, the function would increase (or *boost*) the task's priority. But if a higher priority task were for some reason to leave the mutex (e.g., timeout or signal), this same function would decrease (or *unboost*) the priority of the task. This is because the PI list always contains the highest priority task that is waiting on a mutex owned by the task.

When the function `__rt_mutex_adjust_prio` is performed on a task, the nodes of the task's waiter are not updated with the new priority, therefore this task may not be in the proper locations in the waiter list of the mutex the task is blocked on and in the PI list of the corresponding owner. The function `rt_mutex_adjust_prio_chain` solves all this: it walks along the PI chain originating from the task, and updates nodes and priorities of each process it finds.

The PI chain walk can be a time-consuming; for this reason, `rt_mutex_adjust_prio_chain` not only defines a maximum lock depth, but it also only holds at most two different locks at a time, as it walks the PI chain (this means that the state of the PI chain can change while in `rt_mutex_adjust_prio_chain`). When the function is called no locks is held. Then, roughly, a loop is entered where:

1. the PI lock of the task is taken to prevent more changes to its PI list;

[2]The maximum depth of the PI chain is not dynamic and can actually be defined by an static analysis of the code.

2. if the task is not blocked on a mutex, the loop is exited (we are at the top of the PI chain); otherwise the wait lock of the mutex is taken to update the task's node location in the wait list;
3. the PI lock of the task is released, and the PI lock of the mutex's owner is taken to update the task's node location in the PI list of the new process;
4. the PI lock of the previous owner and the wait lock of the mutex are released; a new iteration of the loop is started where the previous owner will be the next task to be processed.

4.2 The implementation of BWI

The majority of our modifications (and of the difficulties we found, see below) consists in the integration of the SCHED_DEADLINE patch described in section 3.1, with Linux's PI infrastructure described in section 4.1. Indeed, at this time SCHED_DEADLINE only implements an approximated version of deadline-inheritance, in which the relative deadlines of the tasks are inherited but without control on the corresponding bandwidths; on the other hand, the implementation of PI was designed and optimised for fixed-priority tasks, hypothesis which can not be assumed for tasks scheduled with SCHED_DEADLINE policy. In the course of our work, we have tried to keep at the minimum the modifications to original data structures in Linux and in SCHED_DEADLINE, and to maintain the original functions' semantics.

4.2.1 Structures

As mentioned in section 4.1, Linux implements the waiter and the PI lists using priority-sorted linked lists. These are linked lists suitable to sort processes with fixed priorities; unlike ordinary lists, the head of this list is a different element than the nodes of the list. On the other hand, the version of deadline-inheritance which comes with SCHED_DEADLINE replaces these priority-sorted linked lists with red-black trees ordered by (absolute) deadline.

Our implementation adopts these structures to store chains of tasks and mutexes (that we will continue to call PI chains) so that it can keep the list of servers that a task can inherit sorted accordingly. By caching the left-most node of the tree, it takes $O(1)$ to retrieve the highest priority (earliest deadline) task in the list, as for Linux's priority-sorted linked-lists.

A pointer of type `sched_dl_entity` (the scheduling entity for SCHED_DEADLINE task) has been added to each `sched_dl_entity` structure. This pointer, named `bwi`, is required to store the effective scheduling parameters (deadline and capacity) of a SCHED_DEADLINE task, and it will point either to the task's default scheduling entity or, if a waiter with earlier deadline exists in the task's PI list, to the scheduling entity of the task's top PI waiter. In either cases, we will called the scheduling entity pointed by `bwi` the *inherited* scheduling entity or the *inherited server*.

We note that this terminology is not completely consistent with the one in [13], even if the net result is the same. In the original presentation, each process "inherits" all the servers in its PI list, and it can executes in any of them; the CBS algorithm will then select the server (and the only runnable task within the server) with the earliest deadline. For the scope of this document, *the* inherited server, or the inherited scheduling parameters, will be this last. The difference in the terminology is mainly due to the fact that nor Linux neither SCHED_DEADLINE provide us with a server structure, and to the fact that SCHED_DEADLINE stores the server parameters in the scheduling entity of the task.

4.2.2 Functions

Linux resorts to the PI's logic (priority adjustments and chain walks) each time an event occurs that can possibly result in a PI chain modification. Common examples are the blocking of a task on a mutex, the release of a mutex, or the explicit call to a system call that can modify the priority and the scheduling class of a task (`sched_setscheduler`, `sched_setparam`). This remains true in the case of BWI for SCHED_DEADLINE tasks: priority adjustments and chain walks are necessary when the CBS algorithm modifies the deadline of a server, i.e. at each capacity replenishment and each deadline update. For this reason, our implementation modifies the method `enqueue_task` of the SCHED_DEADLINE scheduling class, where updates and replenishments can happen, in order to fire those adjustments.

Even if the logic underlying the chain walks and the priority adjustments remains similar to the one in Linux's PI, these modifications presented a few challenges.

(IRQ safety.) As mentioned in section 4.1, Linux does not disable interrupts before taking wait locks, because it never modifies the waiter lists in interrupt

context. This is not true for SCHED_DEADLINE tasks under BWI, since replenishments do happen in timers interrupt context.

To solve this issue, our implementation disable interrupts before entering the corresponding critical section and re-enable them after after leaving that. Since not all the critical section are within the scope of a functions, it was necessary to add a field in the mutex structure, in order to store the status of the interrupts.

(Wake-ups in chain walks.) Consider the chain $R_i \rightarrow \tau_i$ and the arrival of a new task τ_j that blocks on R_i:

$$\tau_j \rightarrow R_i \rightarrow \tau_i.$$

Linux begins updating this chain starting from τ_j. As described in section 4.1, τ_j's PI lock is taken to prevent additional modification to its PI list; then R_i's wait lock is taken to insert τ_j's waiter structure in R_i's wait list. Finally, τ_j's PI lock is released, but keeping R_i's wait lock, to continue with the next task (and its PI lock) in the chain, τ_i. Due to this fine-grained locking mechanism, nothing prevents τ_i from releasing the mutex at this time: in this case (i.e. if it is found that R_i has no owners at this time), the chain walk needs to be interrupted and τ_j, R_i's top waiter, woken up (i.e., enqueue back to its runqueue). Since this is happening before τ_j's waiter structure had been freed, the enqueue of τ_j will generate a deadlock on R_j's wait lock (when firing the chain walk starting from τ_j (R_j's wait lock being already locked before the wakeup).

Our implementation detects this situation from the status of the task to be enqueued: if the task is waking up, the enqueue will only pursue the updating on the task's deadline and capacity (if needed), without firing a useless chain walk.

(Concurrent chain walks.) The triggering of chain walks and of the corresponding priority adjustments during the enqueueing of a task, may generate other deadlock situations, in case of concurrent chain walks. This is due to the fact that the method `enqueue_task` of any Linux's scheduling class must hold the PI lock of the task that it is to be enqueued, and the runqueue's lock (the runqueue in which we are going to enqueue the task, nested inside the task's PI lock). Our implementation does not break these rules, because releasing any of these locks would have inevitably changed the semantics of the method.

To see the problem, consider a situation similar to the one above, in which a task τ_i is executing in the server inherited from τ_j (that is, τ_j has an earlier deadline than τ_i and τ_j is blocked on a mutex R_i owned by τ_i). In this situation, τ_i consumes τ_j's capacity and a replenishment is required when τ_i depletes it: in the `enqueue_task` (of task τ_i) the chain from τ_j is walked (τ_i had its scheduling parameters modified). In this case, the arrival of a new task τ_k blocking on R_i will fire a second chain walk,

$$\begin{array}{ccccc} \tau_j & \rightarrow & R_i & \rightarrow & \tau_i \\ & \nearrow & & & \\ \tau_k & & & & \end{array}$$

that can deadlock with the first:

- (in the chain from τ_j) τ_i (`enqueue_task`) and τ_j's PI locks held, take R_i's wait lock;
- (in the chain from τ_k) R_i's wait lock held, take τ_i's PI lock.

There are several solutions to this problem. The simplest, even if not the most rigorous one, is probably to detect the contention on one of these locks, and to just give up after a certain number of retries. This is the solution that we adopted in our implementation, where we allow the chain walk in the enqueue to fail, eventually. Clearly, any correct solution will need to modify either the locking order (e.g. by releasing τ_i's PI lock, and so the runqueue's lock!) or the locking granularity in the chain walk (e.g., by using a single lock per chain).

As already mentioned, our implementation modifies all the methods of the SCHED_DEADLINE class to use the effective scheduling parameters of the task. In particular, the method `update_curr_task` , where the execution of the current server is accounted to its capacity, now acts on the inherited server (if any). Also, the functions `push_dl_task`, `pull_dl_task`, and the methods of the `cpudl` heap structure have been modified to act on the inherited server (if any), so that global scheduling for SCHED_DEADLINE is available in SMPs ([15], see section 4.3).

4.3 Issues with clustered scheduling

In order to improve schedulability in multi-core systems, the Linux kernel provides a mechanism to set an affinity mask defining on which CPUs each task could execute. This approach improves performances by using application-specific information like the pattern of cache accesses or the use of specific devices. Recently, Gujarati et al.[11] demonstrated that job-level scheduling algorithms based on arbitrary processor affinity (APA) outperforms global, clustered, and partitioned approaches. However, integrate affinities inside the M-BWI protocol is not

straightforward because is not yet clear what happens when a lock owner inherits the affinities of a blocked task.

Inheriting only the bandwidth and not the affinity from a blocked task could jeopardise the temporal isolation, as shown in Figure 3. In this case, tasks τ_1 and τ_3 can execute only on $CPU0$ while τ_2 is assigned exclusively to $CPU1$. At time $t = 2$, task τ_2 blocks on the mutex owned by τ_1, which inherits the bandwidth and the deadline continuing its execution on $CPU0$. When task τ_3 arrives at time $t = 3$ it cannot preempt τ_1 because of job priorities ($d_2 < d_3 < d_1$), but τ_2 has not been considered in the schedulability analysis of $CPU0$ thus leading to a deadline miss in $t = 11$.

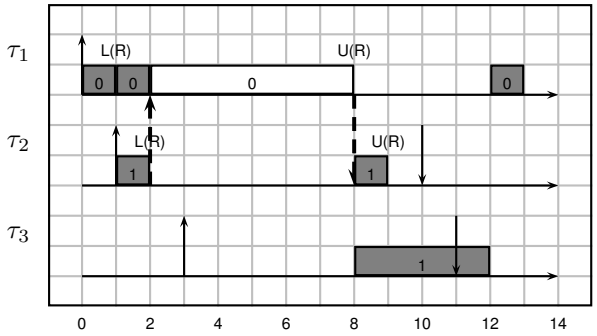
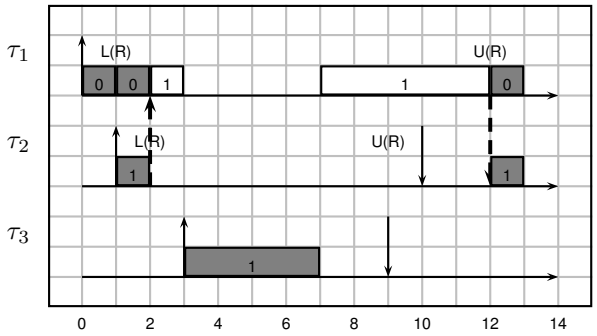


FIGURE 3: *Deadline miss caused by a task inheriting bandwidth and deadline but not affinity from a blocked task.*

Inheriting the affinity mask together with deadline and buffer is not enough to solve the problem, as shown by the example described in Figure **??** where tasks τ_1 can execute only on $CPU0$ while τ_2 and τ_3 are assigned exclusively to $CPU1$. At time $t = 2$, task τ_2 tries to acquire the lock owned by τ_1 which consequently migrates from $CPU0$ to $CPU1$. When the task τ_3 is activates at time $t = 3$ it preempts τ_1 which cannot execute till $t = 7$ even if its originally assigned CPU is idle, generating a deadmiss for task τ_2.



The above examples show that our implementation of BWI (and similarly, Linux's implementation of PI) does not extend to the case of clustered scheduling.

Brandenburg ([5]), proposed the Migratory Priority Inheritance protocol for clustered scheduling, and proved results of its optimality in terms of maximum PI-blocking for Job-Level-Fixed-Priority scheduling (JLFP). Under migratory priority inheritance, whenever a job J is not scheduled (but ready) and there exists a job J_i waiting for J to release a resource such that J_i is eligible to be scheduled in its assigned cluster, J migrates to J_i's cluster (if necessary) and assumes J_i's priority. The idea is that jobs should inherit (both the priority and the affinity mask) only when they "have to". However it is not yet clear how to extend this idea to the not-JLFP context, and how to implement it within the Linux kernel.

5 Evaluation

5.1 Experimental setup

We ran experiments on an Intel®Core2™ quad-core machine (Q6600) with 4GB of RAM and running at 2.4GHz.

Runtime validation consisted of executing two synthetic benchmarks. The first implements a simple situation in which two tasks share a resource protected by a mutex, and a third one, independent from the other two, is periodically activated to check if the inheritance mechanism works. The second executes a similar configuration on an SMP system.

Runtime overheads were instead measured running another benchmark, called (`rt-app`[3]). Using this application we simulated a real-time periodic load consisting of multiple SCHED_DEADLINE threads sharing resources protected by mutexes.

5.2 Runtime validation

We performed simple tests to validate the implementation. In the first test two threads are run that operate on the same mutex (denoted as `A`). A third thread has nothing to do with the first two, its only intent is to demonstrate that BWI mechanism (when enabled) works properly. All threads are restricted to execute on the same CPU.

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

[4]Execution diagrams in this section are created through the KernelShark (https://lwn.net/Articles/425583/) utility from execution traces extracted from the kernel via ftrace (Documentation/trace/ftrace.txt).

Figure 4 shows a visual representation[4] of a run when BWI mechanism is disabled. Threads τ_1 and τ_3 share a resource for which mutual exclusion is achieved through the use of a mutex. Both threads are periodic with periods of, respectively, $24ms$ and $72ms$ (deadline are set equal to periods). τ_1 executes, entirely inside the critical section, for $8ms$ every period, τ_3 has an execution time of $24ms$, of which $20ms$ are spent inside the critical section. τ_2 has no critical section and executes for $8ms$ every $24ms$. Thread τ_3 is the first to be activated, after a while it acquires the mutex (`L(A)` in the figure). Then τ_1 is woken up, tries to lock the same mutex and blocks on `A` queue, waiting for τ_3 to release it. Since it has a shorter deadline than τ_3, when τ_2 is activated it preempts τ_3 causing unexpected delay inside the critical section. τ_3 can only resume execution once τ_2's job has finished. When τ_3 releases the mutex (`U(A)`) τ_1 executes inside the critical section and then both threads' jobs complete.

Same configuration is run with BWI mechanism enabled and is depicted in Figure 5. In this case, when τ_1 is activated and blocks on mutex `A`, τ_3 can start executing in τ_1's server (highest priority server among τ_3 waiters) and this is highlighted with a darker blue shade in the figure. Since τ_3 has inherited also τ_1's deadline, when τ_2 arrives it doesn't immediately preempt τ_3. Mutex owner is actually preempted only when τ_1's server budget is exhausted and its deadline postponed (τ_2's deadline becoming the earliest), event `Rep(S1)` in the figure. After that the execution proceeds like in the previous situation.

The second test is performed on a 2 CPUs system. Two tasks (τ_1 and τ_3) share a resource protected by a M-BWI enabled mutex `A` (we omit the standard case for brevity) and are free to execute on every CPU. Other two tasks (τ_2 and τ_4) are pinned each one on a different CPU and are independent from the others and between themselves (they are thought to create interference). Figure 6 zooms in a particular execution window. A job of task τ_3 arrives on CPU1 and gets scheduled, τ_3 acquires mutex `A` and enters the critical section. A few instants after a job of τ_1 arrives, τ_1 tries to acquire mutex `A` and blocks, donating its server to τ_3. The interesting part comes when τ_4 is activated. Having an earliest deadline than τ_3's original one, τ_4 should have preempted it, but its execution is delayed until τ_3 releases mutex `A` and is consequently deboosted. After this instant of time execution continues with original parameters. Without the M-BWI mechanism working τ_3 would have been preempted inside the critical section by τ_4, delaying τ_1 execution.

5.3 Overheads measurements

We measured runtime overheads comparing execution of the same benchmark with the BWI mechanism activated, with simple deadline inheritance, and against the stock fixed priority Linux scheduler. Similarly to Brandenburg [5]'s evaluation, on each core, we launched four tasks with periods $1ms$, $25ms$, $100ms$, $1000ms$ and execution time of $0.1ms$, $2ms$, $15ms$, $600ms$. The one-millisecond tasks did not access any shared resources. All other tasks shared the same lock (in groups of three, i.e., one lock for each core) with an associated maximum critical section length of $1ms$, and each of their jobs acquired the lock once.

We ran the task set once using the stock Linux scheduler (SCHED_FIFO with priority inheritance enabled, called `pi` in what follows), once using the original SCHED_DEADLINE implementation (deadline inheritance, `dl`) and once with the M-BWI mechanism enabled (`bwi`), for 60 seconds each. Although the same task sets can be run with priority inheritance mechanisms turned off, we don't report figures coming from that configurations here as they are hardy comparable to cases when priority inheritance (or BWI) is enabled. In fact, execution paths inside the kernel are completely different, and unrelated functions get called, thus making the comparison of little interest for the present discussion.

Figure 7 reports measurements of kernel functions, obtained using `ftrace`, that could be ill-affected by the mechanism implementation:

a) `schedule()`, scheduler core, it decides which task to run next and performs the context switch;

b) `do_futex()`, `sys_futex()` system call entry point;

c) `enqueue_task_dl()`/`enqueue_task_rt()`, enqueue a task, respectively, on the `dl` or the `rt` runqueues;

d) `rt_mutex_slowlock()`, work required to acquire a mutex;

e) `rt_mutex_slowunlock()`, work required to release a mutex.

Results show that overheads of `dl` (yellow, oblique lines, boxes) and `bwi` (red boxes) are comparable. Differences between `bwi` and `pi` (blue, small circles, boxes) measurements remain in the same order of magnitude (even if `bwi` doubles `pi` in some case). These differences can be ascribed to the

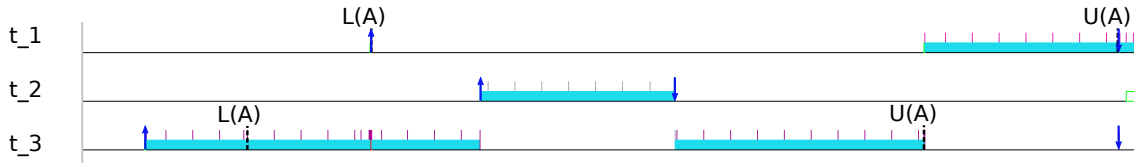


FIGURE 4: *Two task (τ_1 and τ_3) sharing one resource (protected by a normal mutex). A third independent task (τ_2) arrives and preempts τ_3 even if τ_1's server has higher priority than τ_2's.*

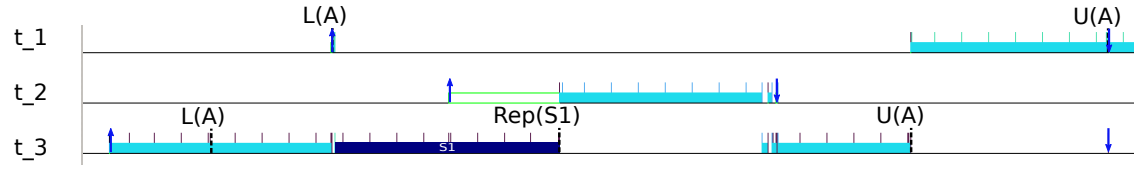


FIGURE 5: *Two task (τ_1 and τ_3) sharing one resource (protected by a BWI-enabled mutex). A third independent task (τ_2) has to wait τ_1's server replenishment event to start executing.*

slightly higher complexity of `bwi` implementation, but also to the fact that tasks interactions can be modified by scheduling the same taskset using different scheduling policies (in this can have an impact on runtime overheads).

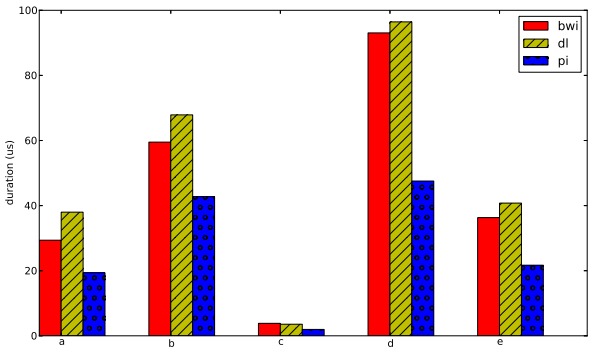


FIGURE 7: *Kernel functions durations (in μs) from a run on a real machine.*

We have then modified the previous example in order to create longer PI chains: a new task with period $2000ms$ and execution time $700ms$, and two more mutexes were added to the above taskset. Like in the previous example, there is a task that does not use any resource; no task accesses more than two mutexes, but the resulting PI chain can reach a depth of 4:

$$\tau_1 \rightarrow R_1 \rightarrow \tau_2 \rightarrow R_2 \rightarrow \tau_3 \rightarrow R_3 \rightarrow \tau_4$$

We replicated this taskset 3 times for a total of 15 tasks, due to bandwidth contraint. The results displayed in Figure 8 show that the effect of the chain's depth contributes in an equivalent amount for the three implementations.

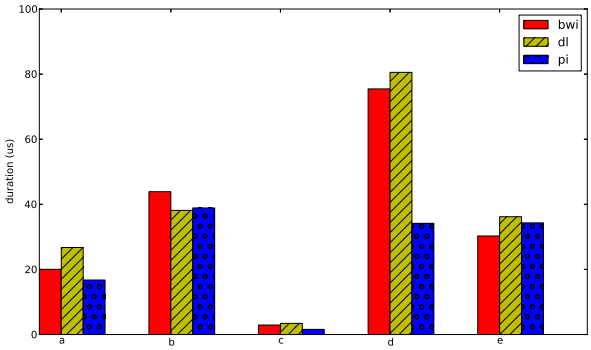


FIGURE 8: *Kernel functions durations (in μs) with nested critical sections, from a run on a real machine.*

6 Conclusions

In this paper we presented an implementation of the M-BWI protocol in the Linux kernel with the SCHED_DEADLINE patch. We tried to be as adherent as possible to the original implementation of the priority inheritance protocol in Linux and to the SCHED_DEADLINE patch by minimising the number of modifications. The overhead of our implemen-

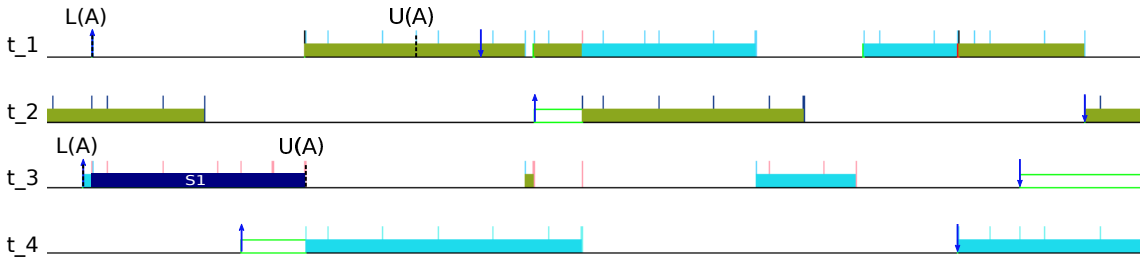


FIGURE 6: *System with two CPUs. Two task (τ_1 and τ_3) sharing one resource (protected by a BWI-enabled mutex). Other two independent tasks (τ_2 and τ_4) are pinned each one on a different CPU.*

tation is only slightly larger than the typical overhead of the PI with SCHED_FIFO. We also believe that such overhead can be reduced by a careful optimisation of the code.

As future work, we are investigating the problems that we encountered when trying to inherit the affinity mask of the lock-owner task. We believe that to overcome such difficulties it is necessary to rethink the current implementation of the SCHED_DEADLINE patch, by introducing the concept of *server* as a separate scheduling entity in the implementation.

References

[1] "Documentations/rt-mutex.txt, Documentation/rt-mutex-design.txt," http://www.kernel.org.

[2] "DROPS The Dresden Real Time Operating System Project."

[3] L. Abeni and G. Buttazzo, "Integrating Multimedia Applications in Hard Real-Time Systems," in *Proc. 19th IEEE Real Time Systems Symposium*, 1998.

[4] T. P. Baker, "Stack-based scheduling for real-time processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, Apr. 1991.

[5] B. B. Brandenburg, "A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications," in *Proceedings of 25th Euromicro Conference on Real-Time Systems (ECRTS 2013)*, July 2013, pp. 292–302.

[6] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "Litmus-rt: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, ser. RTSS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 111–126. [Online]. Available: http://dx.doi.org/10.1109/RTSS.2006.27

[7] "Real-time linux wiki," https://rt.wiki.kernel.org/index.php/Main_Page, Oct. 2013, latest accessed on 10 Oct. 2013.

[8] D. Faggioli, G. Lipari, and T. Cucinotta, "Analysis and implementation of the multiprocessor bandwidth inheritance protocol," *Real-Time Systems*, vol. 48, pp. 789–825, 2012, 10.1007/s11241-012-9162-0.

[9] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, "A new kernel approach for modular real-time systems development," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 199–206.

[10] J. B. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks," Carnegie-Mellon University, Tech. Rep. CMU/SEI-88-SR-4, March 1988.

[11] A. Gujarati, F. Cerquerira, and B. B. Brandenburg, "Schedulability analysis of the linux push and pull scheduler with arbitrary processor affinities," in *Proceedings of the 25th Euromicro conference on Real-time systems*, ser. ECRTS13. IEEE Computer Society, 2013, pp. 69–79. [Online]. Available: http://dx.doi.org/10.1109/ECRTS.2013.18

[12] H. Härtig, R. Baumgartl, M. Borriss, C. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter, "DROPS: OS support for distributed multimedia applications," in *Proc. 8th ACM SIGOPS European Workshop*, Sep. 1998.

[13] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronization in open systems," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, dec. 2001, pp. 151 – 160.

[14] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta, "An efficient and scalable implementation of global edf in linux," in *Proceedings of the 7^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2011)*, Porto, Portugal, 7 2011.

[15] ——, "An efficient and scalable implementation of global edf in linux," in *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2011 2011.

[16] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems." *IEEE Trans. Computers*, vol. 53, no. 12, pp. 1591–1601, 2004.

[17] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems," in *Proc. Conf. on Multimedia Computing and Networking*, January 1998.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," in *IEEE Transactions on Computers*, vol. 39, 1990, pp. 1175–1185.

[19] ——, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, September 1990.