An efficient and scalable implementation of global EDF in Linux *

Juri Lelli, Giuseppe Lipari, Dario Faggioli, Tommaso Cucinotta
{name.surname}@sssup.it
Scuola Superiore Sant'Anna - ITALY

August 27, 2011

Abstract

The increasing popularity of multi-core architectures is pushing researchers and developers to consider multi-cores for executing soft and hard real-time applications. Real-Time schedulers for multi processor systems can be roughly categorized into partitioned and global schedulers. The first ones are more adequate for hard real-time embedded systems, in which applications are statically loaded at start-up and rarely change at run-time. Thanks to automatic load balancing, global schedulers may be useful in open systems, where applications can join and leave the system at any time, and for applications with highly varying workloads.

Linux supports global and partitioned scheduling through its real-time scheduling class, which provides `SCHED_FIFO` and `SCHED_RR` fixed priority policies. Recently, the `SCHED_DEADLINE` policy was proposed that provides Earliest Deadline First scheduling with budget control. In this paper we propose a new implementation for global EDF scheduling which uses a heap global data structure to speed-up scheduling decisions. We also compare the execution time of the main scheduling functions in the kernel for four different implementations of global scheduling, showing that our implementation is as scalable and efficient as `SCHED_FIFO`.

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

1 Introduction

Multi-processor and multi-core computing platforms are nowadays largely used in the vast majority of application domains, ranging from embedded systems, to personal computing, to server-side computing including GRIDs and Cloud Computing, and finally high-performance computing.

In embedded systems, small multi-core platforms are considered as a viable and cost-effective solution, especially for their lower power requirements as compared to a traditional single processor system with equivalent computing capabilities. The increased level of parallelism in these systems may be conveniently exploited to run multiple real-time applications, like found in industrial control, aerospace or military systems; or to support soft real-time Quality of Service (QoS) oriented applications, like found in multimedia, gaming or virtual reality systems.

Servers and data centres are shifting towards (massively) parallel architectures with enhanced maintainability, often accompanied by a decrease in the clock frequency driven by the increasing need for "green computing" [13]. Cloud Computing promises to move most of the increasing personal computing needs of users into the "cloud". This is leading to an unprecedented need for supporting a large number of interactive and soft real-time applications, often involving on-the-fly media streaming, processing and transformations with demanding performance and latency requirements. These applications usually exhibit nearly periodic workload patterns which often do not saturate the available computing power of a

single (powerful) CPU. Therefore, there is a strong industrial interest in executing an increasing number of applications of this type onto the same system, node, CPU and even core, whenever possible, in order to minimize the number of needed nodes (and reduce both power consumption and costs). In this context, a key role is played by real-time CPU scheduling algorithms for multi-processor systems. These can be roughly categorised into *global schedulers* and *partitioned schedulers*.

In *partitioning*, tasks are allocated to cores and will rarely (or never) move from one core to another one. Every core has its own private scheduling queue, and its private scheduler. In *closed systems*, the allocation algorithm is executed off-line and tasks are statically allocated to cores. In *open systems* tasks can dynamically join and leave the system: however task join/leave are rare event compared to the time granularity of the run-time events (i.e. the tasks' periods, or the scheduling tick). When a task joins the system, the allocation algorithm is executed, which may cause a re-allocation of existing tasks and hence a migration from one core to another (*load balancing*).

In *global scheduling*, ready tasks are enqueued in a logical global queue, and the M highest priority tasks are selected to run on the M cores. Therefore, a task can be suspended on one core and resume execution on another core. The number of migrations is much higher than in the previous case. *Clustered schedulers* reside in the middle, where the available processors are partitioned into clusters to which tasks are statically assigned, but in each cluster tasks are globally scheduled.

Global scheduling has the advantage of automatically performing load balancing, however it also raises many concerns about its practicality. Indeed, migrations may invalidate the cache, increasing the task execution time. For this reason, partitioned scheduling is mostly used in hard real-time embedded applications, where tasks are known at configuration time, and optimal allocation and load balancing can be performed off-line. On the other end, global scheduling seems more appropriate for dynamic open systems with highly varying workloads.

One concern of researchers and practitioners is the overhead of scheduling. In particular, the problem is to maintain data structures in the kernel to represent the global queue; such global structures are concurrently accessed by all cores and must therefore be appropriately protected with concurrency control mechanisms (lock-based or lock-free).

1.1 Contributions of this work

In this work, we present an implementation of a global EDF scheduler in Linux using a heap data structure to optimize access to the earliest deadline tasks. The implementation is an improvement over `SCHED_DEADLINE` [8]. After describing the base real-time scheduler of Linux (Section 3), and our implementation (Section 4), we compare its performance against the global POSIX-compliant fixed priority scheduler shipped with stock Linux and with the previous version of `SCHED_DEADLINE` (Section 6). The results show that using appropriate data structures it is indeed possible to build efficient and scalable global real-time schedulers. In Section 7, we also identify space for possible improvements.

All the code that has been developed during the experimental evaluation phase of this study can be downloaded by following the instructions at the very top of this page: `https://www.gitorious.org/sched_deadline/pages/Download`.

2 State of the art

When deciding which kind of scheduler to adopt in a multiple processor system, there are two main options: partitioned scheduling and global scheduling.

In partitioned scheduling, the placement of tasks among the available processors is a critical step. The problem of optimally dividing the workload among the various queues, so that the computing resources be well utilised, is analogous to the bin-packing problem, which is known to be NP-hard in the strong sense [10].This complexity is typically avoided using sub-optimal solutions provided by polynomial and pseudo-polynomial time heuristics, like First Fit, Best Fit, etc. [12, 11].

In **global scheduling**, tasks are ordered into a single logical queue and scheduled onto the available

processors. Using a single logical queue, the load is thus intrinsically balanced, since no processor is idled as long as there is a ready task in the global queue. A class of algorithms, called Pfair schedulers [2], is able to ensure that the full processing capacity can be used, but unfortunately at the cost of a potentially large run-time overhead. Migrative and non-migrative algorithms have been proposed modifying well-known solutions adopted for the single processor case and extending them to deal with the various anomalies [1] that arise on a parallel computing platform.

Complications in using a global scheduler mainly relate to the cost of inter-processor migration, and to the kernel overhead due to the necessary synchronisation among the processors for the purpose of enforcing a global scheduling strategy. As we will see in Section 3, in order to reduce the contention, the logical single queue can actually be implemented as a set of distributed queues plus some helper data structures to maintain consistency.

In addition to the above classes, there are also intermediate solutions, like **clustered**- and restricted-migration schedulers. A clustered scheduler [6] limits the number of processors among which a task can migrate. This method is more flexible than a rigid partitioning algorithm without migration.

Linux supports real-time scheduling with two scheduling policies, `SCHED_RR` and `SCHED_FIFO`. They are based on fixed priority and are compliant with the POSIX standard. The base scheduler is global, i.e. tasks can freely migrate across processors. By specifying task's processor affinity, it is possible to pin a task on one processor (partitioning) or to a set of processors (clustered scheduling). The implementation of such scheduler will be described in Section 3.

Recently, the `SCHED_DEADLINE` scheduling class has been proposed for Linux [8]. It mimics the fixed priority scheduling class, but provides Earliest Deadline First scheduling. Again, using affinity it is possible to implement global, partitioned and clustered scheduling. An overview of `SCHED_DEADLINE` can be found in Section 4.

The line of research closer to the approach of this paper is the one carried out by the Real-Time Systems Group at University of North Carolina at Chapel Hill, conducted by means of their $LITMUS^{RT}$ testbed [5] and investigating how real overheads affect analysis results. There are several works by such group going in this direction: in [5] Calandrino et al. studied the behaviour of some variants of global EDF and Pfair, but did not consider fixed-priority; in [4], Brandenburg et al. explored the scalability of a similar set of algorithms, while in [3] the impact of the implementation details on the performance of global EDF is analysed. In all these works, samples of the various forms of overhead that show up during execution on real hardware are gathered and are then plugged in schedulability analysis, trying to make it realistic, which is an important difference between their works and the present paper (that does not consider schedulability as a metric).

Furthermore, in this work we propose an efficient implementation of a global EDF scheduler that, althought shares basic principles with [3], perfectly fits into the stock Linux scheduler, providing experimental evidence of its usability on a large multi-core machine.

3 `SCHED_FIFO` scheduler

In the Linux kernel, schedulers are implemented inside *scheduling classes*. Stock Linux comes with two classes, one for fair scheduling of non-real-time activities (`SCHED_OTHER` policy) and one implementing fixed priority real-time scheduling (`SCHED_FIFO` or `SCHED_RR` policies), following the POSIX 1001.3b [9] specification. In this paper we will focus on the real-time scheduling policies.

The fixed priority scheduling class already supports global scheduling. However, to reduce memory contention and improve locality, the logical global queue is implemented using a set of distributed run-queues, one for each CPU. Tasks are migrated across CPUs using *push* and *pull* operations.

3.1 Run-queues, masks and locks

To keep track of active tasks, the kernel uses a data structure called *runqueue*. There is one runqueue for

each CPU and they are managed separately in a distributed manner. Every runqueue is protected by a spin-lock to guarantee correctness on concurrent updates. Runqueues are modular, in the sense that there is a separate sub-runqueue for each scheduling class. Key components of the fixed priority sub-runqueue are:

- a priority array on which tasks are actually queued;
- fields used for load balancing;
- fields to speed up decisions on a multiprocessor environment.

Tasks are *enqueued* on some runqueue when they wake up and are *dequeued* when they are suspended.

An additional data structure, called `cpupri`, is used to reduce the amount of work needed for a push operation. This structure tracks the priority of the highest priority task in each runqueue. The system maintains the state of each CPU with a 2 dimensional bitmap: the first dimension is for priority class and the second for CPUs in that class. Therefore a push operation can find a suitable CPU where to send a task in $O(1)$ time, since it has to perform a two bits search only (if we don't consider affinity restriction). Concurrent access to bitmap fields is protected by means of spinlocks in a fine-grained way. In particular, there is a different spinlock for each CPU of each class; concurrent tasks have to spin waiting only if they need to update data regarding the same CPU.

3.2 Push and pull operations

When a task is activated on CPU k, first the scheduler checks the local runqueue to see if the task has higher priority than the executing one. In this case, a preemption happens, and the preempted task is inserted at the head of the queue; otherwise the waken-up task is inserted in the proper runqueue, depending on the state of the system. In case the head of the queue is modified, a *push* operation is executed to see if some task can be moved to another queue. When a task suspends itself (due to blocking or sleeping) or lowers its priority on CPU k, the scheduler performs

```
struct dl_rq {
    struct rb_root rb_root;
    struct rb_node *rb_leftmost;
    unsigned long dl_nr_running;

#ifdef CONFIG_SMP
    struct {
        /* two earliest tasks in queue */
        u64 curr;
        u64 next; /* next earliest */
    } earliest_dl;
    int overloaded;
    unsigned long dl_nr_migratory;
    unsigned long dl_nr_total;
    struct rb_root pushable_tasks_root;
    struct rb_node *pushable_tasks_leftmost;
#endif /* CONFIG_SMP */
};
```

Figure 2: `struct dl_rq` extended

a *pull* operation: it looks at the other run-queues to see if some other higher priority tasks need to be migrated to the current CPU. Pushing or pulling a task entails modifying the state of the source and destination runqueues: the scheduler has to dequeue the task from the source and then enqueue it on the destination runqueues.

4 SCHED_DEADLINE

Recently, a new scheduling class has been made available for the Linux kernel, called `SCHED_DEADLINE` [8]. It implements partitioned, clustered and global EDF scheduling with hard and soft reservations [1]. The approach used for the implementation is the same used in the Linux kernel for the fixed-priority scheduler. This is usually called *distributed run-queue*, meaning that each CPU maintains a private data structure implementing its own ready queue and, if global scheduling is to be achieved, tasks are migrated among processors when needed.

In more details:

- the tasks of each CPU are kept into a CPU-specific run-queue, implemented as a *red-black*

[1] Full source code available at: `http://gitorious.com/sched_deadline`.

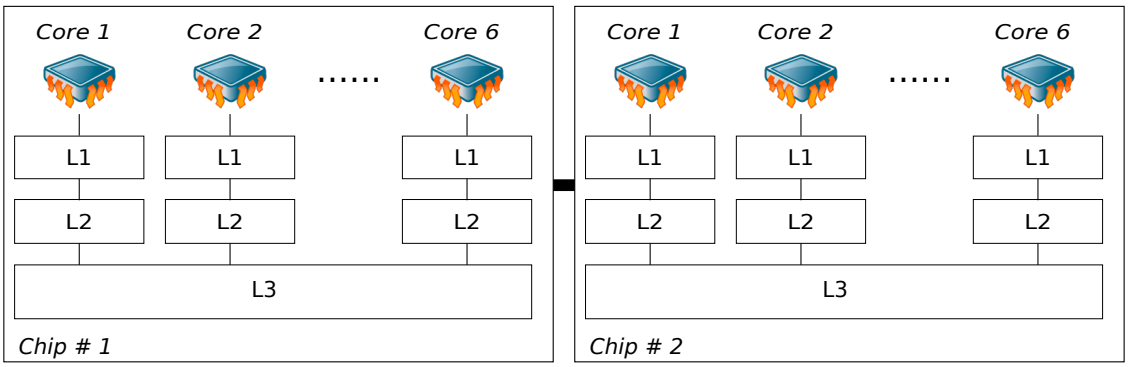

Figure 1: Architecture of a single processor (Multi Chip Module) of the Dell PowerEdge R815.

tree ordered by absolute deadlines;

- tasks are migrated among run-queues of different CPUs for the purpose of fulfilling the following constraints:
  - on m CPUs, the m earliest deadline ready tasks run;
  - the CPU affinity settings of all the tasks is respected.

Migration points are the same as in the fixed priority scheduling class. Decisions related to push and pull logic are taken considering deadlines (instead of priorities) and according to tasks affinity and system topology. The data structure used to represent the EDF ready queue of each processor has been modified, as shown in Figure 2 (new fields are the one inside the `#ifdef CONFIG_SMP` block).

- `earliest_dl` is a per-runqueue data structure used for "caching" the deadlines of the first two ready tasks, so to facilitate migration-related decisions;
- `dl_nr_migratory` and `dl_nr_total` represent the number of queued tasks that can migrate and the total number of queued tasks, respectively;
- `overloaded` serves as a flag, and it is set when the queue contains more than one task;
- `pushable_tasks_root` is the root of the red-black tree of tasks that can be migrated, since they are queued but not running, and it is ordered by increasing deadline;
- `pushable_tasks_leftmost` is a pointer to the node of `pushable_tasks_root` containing the task with the earliest deadline.

A *push* operation tries to move the first ready and not running task of an overloaded queue to a CPU where it can execute. The best CPU where to push a task is the one which is running the task with the latest deadline among the m executing tasks, considering also the constraints due to the CPU affinity settings. A *pull* operation tries to move the most urgent ready and not running tasks among all tasks on all overloaded queues in the current CPU.

4.1 Idle processor improvement

The push mechanism core is realized in a small function that finds a suitable CPU for a to-be-pushed task. The operation can be easily accomplished on a small multi-core machine (for example a quad-core) just by looking at all queues in sequence. The original `SCHED_DEADLINE` implementation realizes a complete loop through all cores for every push decision (pseudo-code on Figure 3). The execution time of such function increases linearly with the number of cores, therefore it does not scale well to systems with large number of cores.

```
cpu_mask push_find_cpu(task) {
    for_each_cpu(cpu, avail_cores) {
        mask = 0;
        if (can_execute_on(task, cpu) &&
            dline_before(task, get_curr(cpu)))
            mask |= cpu;
    }
    return mask;
}
```

Figure 3: Find CPU eligible for push.

```
cpu_mask push_find_cpu(task) {
    if (dlf_mask & affinity)
       return (dlf_mask & affinity);
    mask = 0;
    for_each_cpu(cpu, avail_cores) {
        if (can_execute_on(task, cpu) &&
            dline_before(task, get_curr(cpu)))
            mask |= cpu;
    }
    return mask;
}
```

Figure 4: Using idle CPU mask.

A simple observation is that on systems with large number of processors and relatively light load, many CPUs are idle most of the time. Therefore, when a task wakes up, there is a high probability of finding an idle CPU. To improve the execution time of the push function, we can use a bitmask that stores the idle CPUs with a bit equal to 1. On a 64-bit architecture, we can represent the status of up to 64 processors by using a single word. Therefore, the code of Figure 3 can be rewritten as in Figure 4, where `dlf_mask` is the mask that represents idle CPUs, and the loop is skipped (returning all suitable CPUs to the caller) if is it possible to push the task on a free CPU.

This simple data structure introduces little or no overhead for the scheduler and significantly improves performance figures in large multi-core systems (more on this later). Updates on `dlf_mask` are performed in a thread-safe way: we use a low level `set_bit()` provided in Linux which performs an atomic update of a single bit of the mask.

5 Heap Data structure

When the system load is relatively high, idle CPUs tend to be scarce. Therefore, we introduce a new data structure to speed-up the search for a destination CPU inside a push operation. The requirements for the data structure are: $O(1)$ complexity for searching the best CPU; and less-than-linear complexity for updating the structure. The classical heap data structure fulfils such requirements as it presents $O(1)$ complexity for accessing to the first element, and $O(\log n)$ complexity for updating (if contention is not considered). Also, it can be implemented using a simple array. We developed a *max heap* to keep track of deadlines of the earliest deadline tasks currently executing on each runqueue. Deadlines are used as keys and the *heap-property* is: if B is a child node of A, then $deadline(A) \geq deadline(B)$. Therefore, the node in the root directly represent the CPU where the task need to be pushed.

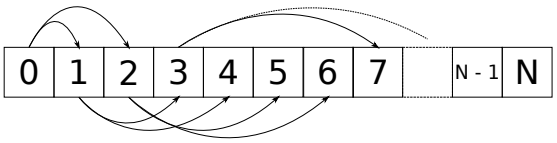

Figure 5: Heap implementation with a simple array.

A node of the heap is a simple structure that contains two fields: a deadline as key and an `int` field representing the associated CPU (we will call it `item`). The whole heap is then self-contained in another structure as described in Figure 6:

- `elements` contains the heap; `elements[0]` con-

```
struct dl_heap {
    spinlock lock;
    int size;
    int cpu_to_idx[NR_CPUS];
    item elements[NR_CPUS];
    bitmask free_cpus;
};
```

Figure 6: Heap structure.

tains the root and the node in `elements[i]` has its left child in `elements[2*i]`, its right child in `elements[2*i+1]` and its parent in `elements[i/2]` (see Figure 5);

- `size` is the current heap size (number of non idle CPUs);
- `cpu_to_idx` is used to efficiently update the heap when runqueues state changes, since with this array we keep track of where a CPU resides in the heap;
- `free_cpus` accounts for idle CPUs in the system.

Special attention must be given to the `lock` field. Consistency of the heap must be ensured on concurrent updates: every time an update operation is performed, we force the updating task to spin, waiting for other tasks to complete their work on the heap. This kind of coarse-grained lock mechanism simplifies the implementation but it increases contention and overhead. In the future, we will look for alternative lock-free implementation strategies.

Potential points of update for the heap are enqueue and dequeue functions. If something changes at the top of a runqueue, a new task starts executing becoming the so-called *curr*, or the CPU becomes idle, the heap must be updated accordingly. We argued, and then experimented, an increase in overhead for the aforementioned operations, but we will show in Section 6 data that suggest this price is worth paying in comparison with push mechanism performance improvements.

```
cpu_mask push_find_cpu(task) {
    if (dl_heap->free_cpus & affinity)
        return (dl_heap->free_cpus & affinity);
    if (maximum(dl_heap) & affinity)
        return maximum(dl_heap);
    mask = 0;
    for_each_cpu(cpu, avail_cores) {
        if (can_execute_on(task, cpu) &&
            dline_before(task, get_curr(cpu)))
            mask |= cpu;
    }
    return mask;
}
```

Figure 7: Find eligile CPU using a heap.

With the introduction of the heap, code in Figure 4 can be changed as in Figure 7, where `maximum(...)` returns the heap root. As we can see from the pseudocode we first try to push a task to idle CPUs, then we try to push it on the *latest deadline* CPU; if both operations fail, the task is not pushed away.

This kind of functioning is compliant with classical global scheduling, as it performs continuous load balancing across cores: rather than compacting all tasks on few cores we prefer every core share an (as much as possible) equal amount of real-time activities.

6 Evaluation

6.1 Experimental setup

The aim of the evaluation is to measure the performance of the new data structures compared with the reference Linux implementation (`SCHED_FIFO`) and the original `SCHED_DEADLINE` implementation. Since all mechanisms described so far share the same structure (i.e. distributed runqueues, and push and pull operations for migrating tasks), we measures the average number of cycles of the main operations of the scheduler: to enqueue and dequeue a task from one of the runqueues; the push and pull operations.

We conducted our experiments on a Dell PowerEdge R815 server equipped with 64GB of RAM, and 4 AMD^R Opteron^{TM} 6168 12-core processors (running at 1.9 GHz), for a total of 48 cores. From a NUMA viewpoint, each processor of the machine contains two 6-core NUMA nodes and is attached to two memory controllers. The memory is globally shared among all the cores, and the cache hierarchy is on 3 levels (see Figure 1), private per-core 64 KB L1D and 512 KB L2 caches, and a global 10240 KB L3 cache. Cache lines are 64B long. The R815 server was configured with a Debian Sid distribution running a patched 2.6.36 Linux kernel.

In the following we will refer the three patches we developed as:

- `original`, the original `SCHED_DEADLINE` implementation;

- `fmask`, `SCHED_DEADLINE` plus changes described in Section 4.1;
- `heap`, `SCHED_DEADLINE` plus the heap described in Section 5.

The reference Linux scheduler is denoted with `SCHED_FIFO`.

6.2 Task set generation

The algorithm for generating task sets used in the experiments works as follows. We generate a number of tasks $N = x \cdot m$, where m is the number of processors (see below), and x is set equal to 3. Further investigation with a much higher number of tasks is left as future work.

The overall utilisation U of the task set is set equal to $U = R \cdot m$ where R is 0.6, 0.7 and 0.8. To generate the individual utilisation of each task, the `randfixedsum` algorithm [7] has been used, by means of the implementation publicly made available by Paul Emberson[2]. The algorithm generates N randomly distributed numbers in $(0, 1)$, whose sum is equal to the chosen U. Then, the periods are randomly generated according to a log-uniform distribution in $[10ms, 100ms]$. The (worst-case) execution times are set equal to the task utilisation multiplied by the task period.

We generated 20 random task sets considering 2, 4, 8, 16, 24, 32, 40 and 48 processors. Than we ran each task set for 10 seconds using a synthetic benchmark (that lets each task execute for its WCET every period). We varied the number of active CPUs using Linux CPU hotplug feature. We collected scheduler statistics through `sched_debug` as to maintain measuring overhead at a minimum value.

6.3 Results

In Figures 8 and 9 we show the number of clock cycles required by a push operation in average, depending on the number of active cores. In Figure 8, we considered an average load per processor equal to $U = 0.6$, while in Figure 9 the load was increased to $U = 0.8$. We measured the 95% confidence interval of each average point, and it is always very low (in the order of a few tens of cycles), so we did not report it in the graphs for clarity.

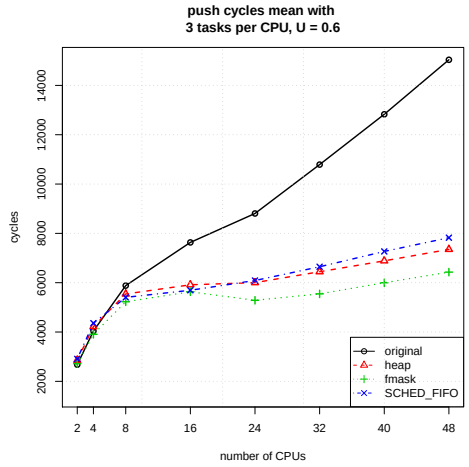

Figure 8: Number of push cycles for average loads of 0.6.

From the graphs it is clear that the overhead of the original implementation of `SCHED_DEADLINE` increases linearly with the number of processors, as expected, both for light load and for heavier load.

In `fmask`, we added the check for idle processors. Surprisingly, this simple modification substantially decreases the overhead for both types of loads, and it becomes almost constant in the number of processors. For light load, `fmask` is actually the one with lowest average number of cycles; this confirms our observation that for light loads the probability of finding an idle processor is high. For heavier loads, the probability of finding an idle processor decreases, so the `SCHED_FIFO` and the heap implementations are now the ones with lowest average overhead. Notice also that the latter two show very similar performance. This means that the overhead of implementing global EDF is comparable (and sometimes even lower) than implementing global Fixed Priority.

To gain a better understanding of these perfor-

[2]More information is available at: `http://retis.sssup.it/waters2010/tools.php`.

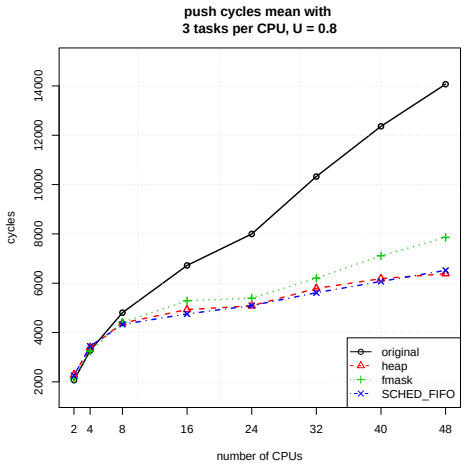

Figure 9: Number of push cycles for average loads of 0.8.

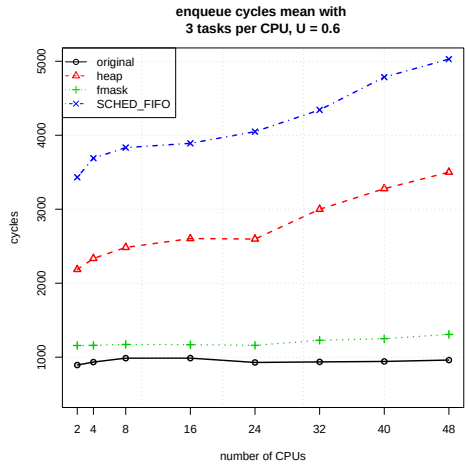

Figure 10: Number of enqueue cycles for average loads of 0.6.

mance figures, it is also useful to analyse the overhead of two basic operations, enqueue and dequeue. Please remind that push and pull operations must perform at least one dequeue and one enqueue to migrate a task.

The number of cycles for enqueue operations for the four implementations is shown in Figure 10 for light load, and in Figure 11 for higher loads. The implementation with the lower enqueue overhead is `original`, because it is the one that requires the least locking and contention on shared data structures: it only requires to lock the runqueue of the CPU where the task is being moved to. `fmask` has a slightly higher overhead, as it also require to update the idle CPU mask with an atomic operation. Heap and `SCHED_FIFO` require the higher overhead as they must lock and update also global data structures (heap in the first case and priority mask in the second case). Updating the heap takes less time probably because it is a small data structure guarded by one single coarse-grain lock, whereas the prioriy mask is a complex and larger data structure with fine-grained locks. Dequeue operations have very similar performance figures and are not shown here for lack of space.

Pull operations do not take advantage of any dedicated data structure. In all four schedulers, the pull operation always looks at all runqueues in sequence to find the tasks that are eligible for migration. Therefore, the execution cycles are very similar to each other. As a future work, we plan to optimize pull operations using dedicated data structures.

7 Conclusions and future work

In this paper we presented an efficient implementation of global EDF seamless integrated with the Linux scheduler. We also compared our implementation with the `SCHED_FIFO` Linux scheduler and with our previous implementation of `SCHED_DEADLINE`. Our implementation is scalable, and its performance is very close to the one of `SCHED_FIFO`.

Is there space for improvements? We believe that it is possible to work along two different directions. First, our work was directed toward improving push operations only. However, migrations may happen also through pull operations. By using appropriate data structures for pulling out tasks from queues, we could speed up also this phase.

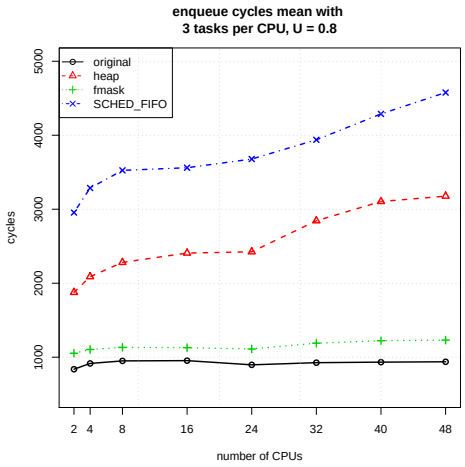

Figure 11: Number of enqueue cycles for average loads of 0.8.

Second, locking is a costly operation in modern multi-core processors, as it is evident from Figures 10 and 11. We will acquire data on time wasted in spinlocks and then investigate the use of lock-free techniques for protecting access to the heap, hoping for reducing the number of cycles required by an enqueue/dequeue.

References

[1] B. Andersson and J. Jonsson. Preemptive multiprocessor scheduling anomalies. In *Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, pages 12–, Washington, DC, USA, 2002. IEEE Computer Society.

[2] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[3] B. B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proc. of the* 30^{th} *IEEE Real-Time Systems Symposium (RTSS 2009)*, Washington D.C., USA, December 2009.

[4] B. B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proc. of the* 29^{th} *IEEE Real-Time Systems Symposium (RTSS 2008)*, Barcelona, Spain, December 2008.

[5] J. Calandrino, H. Leontyev, A. Block, U. Devi, , and J. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the* 27^{th} *IEEE Real-Time Systems Symposium (RTSS 2006)*, Rio de Janeiro, Brazil, December 2006.

[6] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–258, Washington, DC, USA, 2007. IEEE Computer Society.

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

[8] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. An EDF scheduling class for the Linux kernel. In *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, September 2009.

[9] IEEE. *Information Technology - Portable Operating System Interface - Part 1: System Application Program Interface Amendment: Additional Realtime Extensions.* 2004.

[10] Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. Mooney III. A configurable hardware scheduler for real-time systems. In *in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101. CSREA Press, 2003.

[11] Sylvain Lauzac, Rami Melhem, and Daniel Mossé. An improved rate-monotonic admission control and its applications. *IEEE Trans. Comput.*, 52:337–350, March 2003.

[12] Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44:1429–1442, December 1995.

[13] Ernst von Weizsaecker, Karlson Hargroves, Michael Smith, Cheryl Desha, and Peter Stasinopoulos. *Factor Five – Transforming the Global Economy through 80% Improvements in Resource Productivity.* Earthscan, November 2009.