Real-Time Scheduling of Parallel Tasks in the Linux Kernel

José Carlos Fonseca, Luís Nogueira, Cláudio Maia, and Luís Miguel Pinho

CISTER Research Centre/INESC-TEC
School of Engineering (ISEP), Polytechnic Institute of Porto (IPP), Portugal
{jcnfo,lmn,crrm,lmp}@isep.ipp.pt

Abstract. This paper proposes a global multiprocessor scheduling algorithm for the Linux kernel that combines the global EDF scheduler with a priority-aware work-stealing load balancing scheme, enabling parallel real-time tasks to be executed on more than one processor at a given time instant. We state that some priority inversion may actually be acceptable, provided it helps reduce contention, communication, synchronisation and coordination between parallel threads, while still guaranteeing the expected system's predictability. Experimental results demonstrate the low scheduling overhead of the proposed approach comparatively to an existing real-time deadline-oriented scheduling class for the Linux kernel.

Keywords: Real-time scheduling, task-level parallelism, work-stealing, Linux

1 Introduction

High-level parallel languages seek to reduce the complexity of multicore programming by giving programmers abstract execution models, such as implicit threading, where programmers annotate their programs to suggest the parallel decomposition. Implicitly-threaded programs, however, do not specify the actual decomposition of computations or the mapping from computations to cores. The annotations act simply as hints that can be ignored and safely replaced with sequential counterparts. The parallel decomposition itself is the responsibility of the language implementation and, more specifically, of the runtime scheduler.

However, scalable performance is only one facet of the problem in embedded multicore real-time platforms. Predictability and computational efficiency are often conflicting goals, as many performance enhancement techniques aim at boosting the average expected execution time, without considering potentially adverse consequences on worst-case execution times. Therefore, the growing importance of parallel programming models to real-time applications introduces a new dimension to multicore real-time scheduling, since a real-time task may split into parallel execution regions at specific points.

Early work in parallel real-time scheduling [1–3] makes simplifying assumptions about task models, such as knowing beforehand the parallelism degree of

jobs and using this information when making scheduling decisions. In practice, this information is not easily discernible, and in some cases can be inherently misleading. Recently, Lakshmanan et al. [4] proposed a scheduling technique for synchronous parallel tasks where every task is an alternate sequence of sequential and parallel regions, respectively. Each parallel region is formed by the same number of threads, all of equal length, that synchronise at the end of the region. In [5], Saifullah et al. considered a slightly more general task model, allowing different regions of the same parallel task to contain different numbers of threads. Nevertheless, it still requires threads of execution that are of equal length within each parallel region. Furthermore, both works consider scheduling parallel tasks by decomposing them into sequential subtasks. In the former, this technique requires a resource augmentation bound of 3.42 under partitioned Deadline Monotonic (DM) scheduling, while the latter proves a resource augmentation bound of 4 for global Earliest Deadline First (EDF) and 5 for partitioned DM scheduling.

In contrast, we consider a more general model of parallel real-time tasks, where the number of spawned threads may be dynamic and each one of them might take arbitrarily different amounts of time to execute. There are many applications for which the previous conditions hold, and it is this kind of irregular parallelism that is of primary interest for us in this paper. Applications with these properties pose significant challenges for high-performance parallel implementations, where equal distribution of work over cores and locality of reference are desired within each core.

One of the simplest, yet best-performing, dynamic load-balancing algorithms for shared-memory architectures is work-stealing [6]. Blumofe and Leiserson have theoretically proven that the work-stealing algorithm is optimal for scheduling fully-strict computations, *i.e* computations in which all join edges from a thread go to its parent thread in the spawn tree. Under this assumption, an application running on P cores achieves P-fold speedup in its parallel part, using at most P times more space than when running on one core. These results are also supported by experiments [7]. The principle of work-stealing is that idle cores, which have no useful work to do, should bear most of the scheduling costs, and busy cores, which have useful work to do, should focus on finishing that work.

Motivated by these observations, this paper breaks new ground in several ways. First, it proposes a novel scheduling algorithm, named Priority-Aware Work-Stealing (PAWS), that combines the global EDF scheduler [8] with a priority-based work-stealing policy, which enables parallel real-time tasks to be executed in more than one processor at a given time. To the best of our knowledge, no research has ever focused on this subject. Second, while several others have previously considered work-stealing as a load balancing mechanism for parallel computations, we are the first to do so considering different tasks' priorities. Third, our work is the first to actually implement support for parallel real-time computations in the Linux kernel.

2 System model

We consider the scheduling of periodic independent real-time tasks on m identical cores $p_1, p_2, \ldots, p_m$ using global EDF. Global EDF places each task ready to execute in a system-wide queue, ordered by nondecreasing absolute deadline, from which the first m tasks are extracted to execute on the available cores.

We primarily consider a synchronous task model, where each task $\tau_1, \ldots, \tau_n$ can generate a virtually infinite number of multithreaded jobs. A multithreaded job is a sequence of several regions, and each region may contain an arbitrary number of parallel threads which synchronise at the end of the region (see Figure 1). To generate such parallel structures, one may enforce not only parallel *for* loops but also any thread-based construct available in established parallel programming paradigms such as OpenMP [9], as long as no regions contain nested parallelism (it is beyond the scope of this work). For any region with more than one thread, the threads on that region can be executed in parallel on different cores. All parallel regions in a task share the same number of cores and threads have implicit deadlines. For now, our work is focused on systems where all parallel threads are fully independent, *i.e.*, except for the m-cores there are no other shared resources, no critical sections, nor precedence constraints.

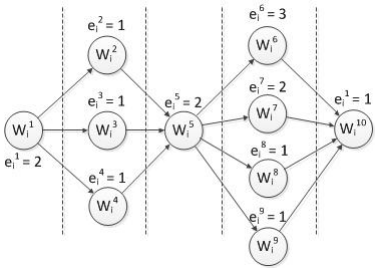

Fig. 1. A multithreaded job with five regions.

The j^{th} job of task τ_i arrives at time $a_{i,j}$, is released to the global EDF queue at time $r_{i,j}$, starts to be executed at time $s_{i,j}$ with deadline $d_{i,j} = r_{i,j} + t_i$, with t_i being the period of τ_i, and finishes its execution at time $f_{i,j}$. These times are characterised by the relations $a_{i,j} \leq r_{i,j} \leq s_{i,j} \leq f_{i,j}$. Successive jobs of the same task are required to execute in sequence.

During the course of its execution, the j^{th} job of task τ_i can dynamically generate an arbitrary number of parallel threads which synchronise at the end of that parallel region. A thread is denoted $w_i^j, 1 \leq j \leq n_i$, with n_i being the total number of threads generated by the j^{th} job of task τ_i in that parallel region. We assume $n_i \geq 2$ holds for at least one task τ_i in the system. Otherwise, the considered task set does not have intra-task parallelism.

The execution requirement of a thread w_i^j is denoted by e_i^j. Threads of the same region may have different execution requirements. Therefore, the worst

case execution time (WCET) C_i of task τ_i on a multicore platform is the sum of the execution requirements of all of its threads, if all threads are executed sequentially in the same core.

The fraction of the capacity of one core that is assigned to a task τ_i is defined as its utilisation $u_i = \frac{C_i}{T_i}$. We further define $U_\Pi = \sum_i^n u_i$ as the system utilisation on the identical multicore platform Π comprised of m unit-capacity cores and $u_\Pi = max_{1 \leq i \leq n} u_i$ as the maximum task utilisation.

A task set Γ is said to be schedulable by algorithm $\mathcal{A}$, if $\mathcal{A}$ can schedule Γ such that every $\tau_i \in \Gamma$ can meet its deadline d_i. With global EDF, a task τ_i executed on the identical multicore platform Π comprised of m unit-capacity cores never misses its scheduling deadline under the following conditions [10]:

$$U_\Pi \leq m - u_\Pi(m-1)$$
$$u_\Pi \leq 1$$

Contrary to regular jobs of a task, dynamically generated parallel threads are not pushed to the global EDF queue but instead maintained in a local priority-based work-stealing double-ended queue (deque) of the core where the job is currently being executed, thus reducing contention on the global queue. For any busy core, parallel threads are pushed and popped from the bottom of the deque and these operations are synchronisation free.

3 Real-time task-level parallelism

Most results in real-time scheduling concentrate on sequential tasks running on multiple processors or cores [11]. While these works allow several tasks to execute on the same multicore host and meet their deadlines, they do not allow individual tasks to take advantage of a multicore machine. It is essential to develop new approaches for intra-task parallelism, where real-time tasks themselves are parallel tasks which can run on multiple cores at the same time instant.

Many real-time applications have a lot of potential parallelism which is not regular in nature and which varies with the data being processed. Implicit threading encourages the programmer to divide the program into threads that are as small as possible, increasing the scheduler's flexibility when distributing work evenly across cores. Frameworks such as OpenMP [9], Cilk [12], Java Fork-join Framework [13], or StackThreads/MP [14] are able to take advantage of that distributed computational power, once they allow parallelism to be easily expressed by spawning threads, which can execute in parallel.

The downside of such fine-grained parallelism is that if the total scheduling cost is too large, then parallelism is not worthwhile. Having many short-lived threads requires a simple and fast scheduling mechanism to keep the overall overhead low. The underlying architecture must then provide an efficient runtime that can efficiently map ready threads to cores, dynamically balancing the workload, while easing this burden from the programmer.

Dynamic scheduling of parallel computations by work-stealing [6] has gained popularity in academia and industry for its good performance, ease of implementation and theoretical bounds on space and time. Work stealing has proven

to be effective in reducing the complexity of parallel programming, especially for irregular and dynamic computations, and its benefits have been confirmed by several studies [15, 16].

A work-stealing scheduler employs a fixed number of workers, usually one per core. Each of those workers has a local deque to store threads. Workers treat their own deques as a stack, pushing and popping threads from the bottom, but treat the deque of another busy worker as a queue, stealing threads only from the top, whenever they have no local threads to execute. This reduces contention, by having stealing workers operating on the opposite end of the queue than the worker they are stealing from, and also helps to increase locality, since stealing a thread also migrates its future workload [12]. All queue manipulations run in constant-time (O(1)), independently of the number of threads in the queues. Furthermore, several papers [17, 18] explain how a non-blocking deque can be implemented to limit overheads.

Following [19], we denote T_∞ as the execution time of an algorithm on an infinite number of cores and T_1 as the sequential time of this algorithm. It is proved that the time T_p required for execution, on an ideal machine with no scheduling overhead, on p cores verifies Equation 1.

$$T_p \leq \frac{T_1}{p} + T_\infty \tag{1}$$

This time appears asymptotically optimal in the case of highly parallel applications where $T_\infty \leq T_1$. However, the need to support tasks' priorities fundamentally distinguishes the problem at hand in this paper from other work-stealing choices previously proposed in the literature [20–22]. With classical work-stealing, threads waiting for execution in a deque may be repressed by new threads, which are enqueued at the bottom of the worker's deque. As such, a thread at the top of a deque might never be executed if all workers are busy. Consequently, there is no upper bound on the response time of a multithreaded real-time job. Therefore, considering threads' priorities and using a single deque per core would require, during stealing, that a worker iterate through the threads in all deques until the highest priority thread to be stolen was found. This cannot be considered a valid solution since it greatly increases the theft time and, subsequently, the contention on a deque.

Our proposal is to replace the single per-core deque of classical work-stealing with a per-core priority queue, each element of which is a deque. A deque holds one or more threads of the same priority. At any time, a core picks the bottom thread from the highest-priority non-empty deque. If it finds its queue empty, it randomly selects another busy core and steals a thread from the top of the highest-priority non-empty deque of the chosen core's queue.

Blumofe and Leiserson [6] demonstrate that a random choice of the stolen core is fair and presents the advantage that the choice of the target does not require more information than the total number of cores in the execution platform. Since the schedulability of the task set is guaranteed by global EDF, we state that priority-aware work-stealing is robust to small deviations from a strict

priority schedule. In fact, as the achieved results demonstrate, controlled priority inversion concerning parallel threads may be actually acceptable, provided it helps reduce contention, communication, synchronisation and coordination between them. Note that task-level scheduling is not affected by this approach, and work-stealing takes place exclusively when a core would, otherwise, be idle.

4 The PAWS scheduler

One approach to scheduling parallel applications using work-stealing is to include the calls to a user-space runtime library that manages threads themselves explicitly in the application. This technique places a lot of onus on the programmer, requiring that he is fully aware of the runtime library and the details of scheduler, which in turn affects the productivity. Hence, work-stealing schedulers generally resort to an alternate approach where the parallelism is expressed at a higher-level of abstraction using some parallel constructs in a language. This code is then transformed into an equivalent version with appropriate calls to the work-stealing runtime library using a compiler. However, the compiler needs to do a good job mapping threads appropriately in order to match the performance of a good hand-written application with direct calls to runtime.

Therefore, implementing a work-stealing scheduler at the kernel level, by exploiting the operating system's capabilities, allows one to finally switch from the current support of user-space runtime libraries or compilers to native support from the operating system. Furthermore, existing user-level work-stealing schedulers are not effective in the increasingly common setting where multiple applications time-share a single multicore, suffering from both system throughput and fairness problems [23].

On the other hand, kernel-focused work has been invaluable in demonstrating the capabilities and limitations of new multicore resource allocation techniques on actual hardware. Among research projects, the works more related to our proposal of extending the Linux kernel with the concept of actual timing constraints, *e.g.* deadlines, are LITMUS^{RT} [24] and SCHED_DEADLINE (originally named SCHED_EDF) [25]. The LITMUS^{RT} patch is a soft real-time extension of the Linux kernel with a focus on multicore real-time scheduling and synchronisation. The Linux kernel is modified to support the sporadic task model and modular scheduler plugins. Work in [25] targets global/clustered EDF scheduling specifically through dynamic task migrations. This means that tasks can migrate among (a subset of) cores when needed, by means of pushes and pulls. Nevertheless, none of those patches directly supports parallel real-time tasks.

PAWS extends the Linux kernel with a global EDF scheduling scheme combined with a priority-based work-stealing load balancing policy, used to allow parallel tasks to execute on more than one core at a time. The major rules of the proposed scheduler are described next.

- **Rule A:** a single global ready queue exists in the system, ordered by nondecreasing absolute deadlines. At each instant, the higher priority (with shorter absolute deadline) jobs are scheduled for execution.

- **Rule B:** whenever a job of a task τ_i being executed at a core k enters a parallel region and dynamically generates a set of parallel threads, those threads are not pushed to the global EDF queue but instead maintained in the core's local priority queue to reduce contention on the global queue.
- **Rule C:** each entry in the local priority queue is a deque, holding one or more threads of equal priority. At any time, a core first looks into its local queue, picking the bottom thread from the highest priority non-empty deque.
- **Rule D:** if the local queue is empty and there is no thread to pick, then a core searches for jobs in the global EDF queue.
- **Rule E:** still, if there is no eligible job in the global EDF queue, the core will randomly select a busy core and steal the thread at the top of the highest priority non-empty deque from the core's priority queue.
- **Rule F:** opposed to a locally generated thread, a stolen thread preempted by a new arriving job with a shorter deadline is enqueued in the global queue and not back in the respective deque of the core's local priority queue.

Each released job is enqueued in a system wide global EDF queue ordered by non-decreasing deadlines, with ties broken by FIFO. At $t = 0$, all the m cores are idle and the m higher priority jobs are selected for execution. By following a global approach, cores are responsible for dequeueing the highest priority jobs from the global queue and therefore the bin-packing problem of partitioned approaches is avoided. Furthermore, our implementation includes a dispatching routine, which directly enqueues a task on the most suitable core (if any). Therefore, contention on the global queue is minimised and only occurs in the case when two or more cores simultaneously finish their executions, their local queues are empty, and there is pending work on the global queue.

When entering a parallel region, a job generates an arbitrary number of threads, possibly with different execution requirements. To avoid uncontrolled priority inversion when stealing, each core has a deadline-ordered queue, each element of which is a deque. Therefore, each dynamically generated thread is enqueued at the bottom of the respective deque, so that data locality is achieved and communication and synchronisation among cores are minimised.

Whenever a new job is released and enqueued in the global EDF queue and all the cores are busy, the scheduler verifies if the core executing the lowest priority job/thread among all the executing jobs/threads has a higher deadline than the newly arrived job. If this is the case, this job is preempted. One of three possible situations occurs, depending on the properties of the preempted entity: (i) the job is enqueued back in the global queue; (ii) the locally generated thread is enqueued back in the respective deque in the core's local priority queue; and (iii) a previously stolen thread is enqueued in the global queue in order to prevent starvation and, therefore, a possible deadline miss.

For each core, the local deques are the first place to look for work, not only due to the fact that if they have work to execute it means that there is a deadline to be met, but also to take advantage of data locality. If local deques are empty, the global queue is searched. This step assumes that if the other cores in the system have work to execute they are able to finish their work within the deadline

(the schedulability of the task set is assured by global EDF). Clearly, this step favours jobs in the global queue with respect to parallel threads generated on other cores. Finally, the last step considers stealing which implies that a core is idle. Therefore, the cost of stealing operations is the responsibility of the idle core. Nevertheless, simultaneously, the system is taking advantage of potential parallelism which contributes to reduce the worst-case response time of the jobs.

The stealing operation assures that the top-right parallel thread, *i.e.* the highest priority thread in that core, is stolen. This reduces contention, by having stealing cores operate on the opposite end of the deque than the core they are stealing from. Moreover, as this thread is the oldest in that deque, it increases the probability of not having its data in the cache and also of being an element with a heavier workload, thus making it the best candidate for stealing.

5 Experimental evaluation

Based on the design principles presented in the previous section, we have implemented PAWS in the standard Linux kernel 2.6.36 as a new scheduling class called SCHED_PAWS. The experiments reported in this paper were conducted in a machine equipped with a dual-core processor, where each of the cores is running at 1.6 GHz, and 4 GB of main memory. Concerning the running environment, the Linux kernel was configured as follows: disabled CPU frequency scaling, hyperthreading and tickless system; HZ macro set to 1000; preemptible kernel selected as preemption model. Since our evaluation is also based in a comparison to SCHED_DEADLINE (version 3), we have disabled bandwidth management to set equal grounds.

A set of four experiments was conducted, where in each of the experiments 20 random task sets were utilised. In order to dinamically generate the task sets, we have defined the number of cores in the system (m) to 2; the minimum task utilisation (u_{min}) equal to 0.1; the maximum task utilisation (u_{max}) equal to 0.4; a minimum period (T_{min}) of 100 ms; and finally, a maximum period (T_{max}) of 150 ms. The period T_i is computed as follows: $T_i = T_{min} + x * (T_{max} - T_{min})$, where x denotes a random value between 0 and 1.

Four utilisation windows were chosen, which constitute the basis for the experiments, where each interval was defined based on U_{min}, the minimum task set utilisation and U_{max}, the maximum task set utilisation. More precisely, the intervals are given by ($[U_{min}, U_{max}]$): $[0.28, 0.30]$, $[0.58, 0.60]$, $[0.78, 0.80]$ and $[0.83, 0.85]$. With these parameters, we compute each task utilisation as follows: u_i is given by $u_i = u_{min} + x * (u_{max} - u_{min})$, where $\sum_{k=1}^{n} u_k \geq U_{min}$ and $\sum_{k=1}^{n} u_k \leq U_{max}$. Finally, C_i is given by $C_i = T_i * U_i$.

The number of parallel threads per task was not defined a priori but derived as follows: $n_i = x * (m * 3)$, whereas the number of tasks (n) was totally dynamic, based on the system utilisation window condition being satisfied. Note that as we keep increasing U_{max}, and u_{max} remains constant, n scales. We strongly believe that this set of task sets can deeply assess our scheduler features.

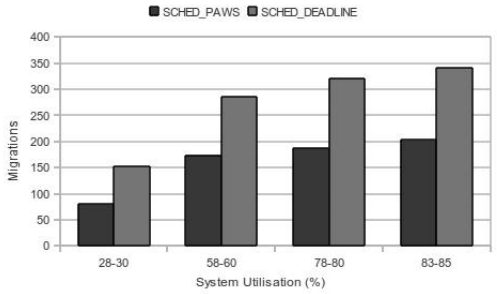

Fig. 2. Average number of migrations between cores

Regarding workloads, each task is an academic application whose periodicity is simulated by a periodically activated "infinite" loop that: (i) executes sequentially; (ii) splits into multiple parallel threads; (iii) syncronizes and terminates them; and (iv) resumes master thread execution. All and each one of the work performed is limited to a bunch of NOP instructions to avoid memory and cache interferences. Sequential, parallel and total execution times are not assigned by a rigid calculation but instead derived randomly, being the actual total execution time upper bounded by C_i.

Data was collected and averaged concerning the number of context switches and migrations. The main goal of these two metrics, which represent the main sources of scheduling overhead, is to highlight the overhead caused by such preemptive time-constrained real-time schedulers. Moreover, Table 1 shows the number of deadline misses verified in the experiments, so that we can understand if an increase in overheads is justified by a gain in schedulability.

	28-30%	58-60%	78-80%	83-85%
SCHED_PAWS	0	0	0	0
SCHED_DEADLINE	0	0	4	3

Table 1. Number of task sets which missed deadlines

Figure 2 depicts the average number of migrations that occurred for each scheduling policy. The overall results show that, no matter the considered workload rate, SCHED_PAWS outperforms SCHED_DEADLINE. This is easily explained by our decision to favour data locality, generating parallelism only when strictly required, *i.e.* when a core becomes idle. As the reader may notice, the number of migrations increases with higher system utilisations. This happens because load balancing calls are more frequently required, as a greater number of tasks must enter the system to fullfill the increasing $[U_{min}, U_{max}]$ (as previously explained), and each one of them create more threads in the parallel regions. Nonetheless, values indicate a linear scalability for both policies, which can be seen as a good behaviour.

Regarding the average number of context switches, depicted in Figure 3, SCHED_PAWS also outperforms SCHED_DEADLINE in every experiments.

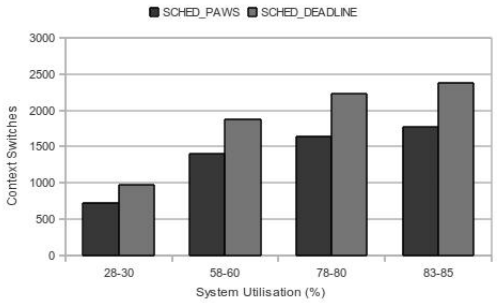

Fig. 3. Average number of context switches

Note that the latter policy blindly assigns new jobs to the core where the task was last executed, which rather frequently leads to a preemption, the only variable factor when accounting such metric. Contrariwise, in our implementation, preemption is minimised because a released job is assigned to a idle core (if available) by the dispacthing routine or inserted into the global queue when its piority is lower than the ones currently executing. Moreover, we do not allow parallel threads to preempt other threads or jobs, unless they have been stolen. Even though the number of context switches follows the previous metric trend, when system workload gets heavier, it appears to scale less than linearly.

Although the underlying operating system's unpredictability, the achieved results seem promising and match to the expectations that guided our scheduling algorithm design. Still, we are now conducting new experiments in a machine with more cores, as well as taking some more metrics to further evaluate the behaviour of PAWS scheduling algorithm.

6 Conclusions and Future Work

Multicore platforms have transformed parallelism into a main concern and dynamic task-level parallelism is steadily gaining popularity as a programming model. The idea is to encourage application programmers to expose the opportunities for parallelism by pointing out potentially parallel regions within tasks, leaving the actual and dynamic scheduling of these regions onto processors to be performed at runtime, exploiting the maximum amount of parallelism.

In contrast to prior work on real-time scheduling of parallel tasks, this paper considers a more general model of parallel real-time tasks, where the number of spawned threads may be dynamic and each one of them might take arbitrarily different amounts of time to execute. It proposes PAWS, a novel scheduling algorithm, implemented in the Linux kernel, that combines the global EDF scheduler with a priority-based work-stealing policy, allowing parallel real-time tasks to be executed in more than one processor at a given time. To the best of our knowledge, we are the first to: (i) deal with real-time priorities in a work-stealing

scheduler; and (ii) to actually implement support for parallel real-time computations in the Linux kernel.

Experimental results show that the proposed scheduler, in comparison to other practical work, significantly reduces the scheduling overhead through an efficient and scalable (regarding tasks/threads and workloads) control of migrations and context switches. While focused on keeping overheads low and achieving good data locality, we have never neglected the system's schedulability. In fact, our scheduling algorithm proved to be very robust as we did not get any deadline miss on the performed experiments. Therefore, we can conclude that some priority inversion, on the randomized work-stealing algorithm, does not compromise the schedulability goals, and it even helps to reduce contention as well as to keep system tracking information to a minimum.

In future work, we will consider an even more general parallel task model that allows nested parallelism and conduct further experiments to evaluate more metrics, such as worst-case response time and task latency, as well as to measure the scalability of PAWS on machines with a greater number of cores.

Acknowledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within REGAIN and VipCore projects, refs. FCOMP-01-0124-FEDER-020447 and FCOMP-01-0124-FEDER-015006.

References

1. Lee, W.Y., Lee, H.: Optimal scheduling for real-time parallel tasks. Transactions on Information and Systems **E89-D** (June 2006) 1962–1966
2. Collette, S., Cucu, L., Goossens, J.: Integrating job parallelism in real-time scheduling theory. Information Processing Letters **106** (May 2008) 180–187
3. Kato, S., Ishikawa, Y.: Gang edf scheduling of parallel task systems. In: Proceedings of the 30th IEEE Real-Time Systems Symposium. (December 2009) 459 –468
4. Lakshmanan, K., Kato, S., Rajkumar, R.: Scheduling parallel real-time tasks on multi-core processors. In: Proceedings of the 31st IEEE Real-Time Systems Symposium. (December 2010) 259 –268
5. Saifullah, A., Agrawal, K., Lu, C., Gill, C.: Multi-core real-time scheduling for generalized parallel task models. In: Proceedings of the 32nd IEEE Real-Time Systems Symposium, Vienna, Austria (December 2011) 217 –226
6. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM **46**(5) (September 1999) 720–748
7. Saha, B., Adl-Tabatabai, A.R., Ghuloum, A., Rajagopalan, M., Hudson, R.L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., Fang, J.: Enabling scalability and performance in a large scale cmp environment. ACM SIGOPS Operating Systems Review **41**(3) (June 2007) 73–86
8. Liu, C.L., Layland, J.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **1**(20) (1973) 40–61

9. ARB, O.: Openmp. Available at http://www.openmp.org/
10. Goossens, J., Funk, S., Baruah, S.: Priority-driven scheduling of periodic task systems on multiprocessors. Real-Time Systems Journal **25** (September 2003) 187–205
11. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Computing Surveys **43**(4) (October 2011) 35:1–35:44
12. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. ACM SIGPLAN Notices **33**(5) (1998) 212–223
13. Lea, D.: A java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande. (2000) 36–43
14. Taura, K., Tabata, K., Yonezawa, A.: Stackthreads/mp: integrating futures into calling standards. ACM SIGPLAN Notices **34**(8) (1999) 60–71
15. Neill, D., Wierman, A.: On the benefits of work stealing in shared-memory multiprocessors. Technical report, Department of Computer Science, Carnegie Mellon University (2009)
16. Navarro, A., Asenjo, R., Tabik, S., Caşcaval, C.: Load balancing using work-stealing for pipeline parallelism in emerging applications. In: Proceedings of the 23rd International Conference on Supercomputing, New York, NY, USA, ACM (2009) 517–518
17. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: Proceedings of the 10th annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, ACM (1998) 119–129
18. Hendler, D., Lev, Y., Moir, M., Shavit, N.: A dynamic-sized nonblocking work stealing deque. Distributed Computing **18** (February 2006) 189–207
19. Blumofe, R.D., Leiserson, C.E.: Space-efficient scheduling of multithreaded computations. In: Proceedings of the 25th ACM symposium on Theory of computing, New York, NY, USA, ACM (1993) 362–371
20. Vrba, Z., Halvorsen, P., Griwodz, C.: A simple improvement of the work-stealing scheduling algorithm. In: Proceedings of the 4th International Conference on Complex, Intelligent and Software Intensive Systems. (February 2010) 925–930
21. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In: Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing. (April 2010) 1–12
22. Vrba, v., Espeland, H., Halvorsen, P., Griwodz, C.: Limits of work-stealing scheduling. In: Proceedings of the 14th International Workshop on Job Scheduling Strategies for Parallel Processing. (May 2009) 280–299
23. Ding, X., Wang, K., Gibbons, P.B., Zhang, X.: Bws: balanced work stealing for time-sharing multicores. In: Proceedings of the 7th ACM European Conference on Computer Systems, New York, NY, USA, ACM (2012) 365–378
24. Calandrino, J.M., Leontyev, H., Block, A., Devi, U.C., Anderson, J.H.: Litmus^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: Proceedings of the 27th IEEE International Real-Time Systems Symposium. (2006) 111–126
25. Faggioli, D., Trimarchi, M., Checconi, F.: An implementation of the earliest deadline first algorithm in linux. In: Proceedings of the 2009 ACM symposium on Applied Computing. (March 2009) 1984–1989
26. Department, V.Y., Yodaiken, V.: The rtlinux manifesto. In: In Proc. of The 5th Linux Expo. (1999)