Technical Report

Dynamic Global Scheduling of Parallel Real-Time Tasks

Luis Miguel Nogueira

José Fonseca

Cláudio Maia

Luís Miguel Pinho

HURRAY-TR-121005

Version:

Date: 10-17-2012

Dynamic Global Scheduling of Parallel Real-Time Tasks

Luis Miguel Nogueira, José Fonseca, Cláudio Maia, Luís Miguel Pinho

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

High-level parallel languages offer a simple way for application programmers to specify parallelism in a form that easily scales with problem size, leaving the scheduling of the tasks onto processors to be performed at runtime. Therefore, if the underlying system cannot efficiently execute those applications on the available cores, the benefits will be lost.

In this paper, we consider how to schedule highly heterogenous parallel applications that require real-time performance guarantees on multicore processors. The paper proposes a novel scheduling approach that combines the global Earliest Deadline First (EDF) scheduler with a priority-aware work-stealing load balancing scheme, which enables parallel real-time tasks to be executed on more than one processor at a given time instant. Experimental results demonstrate the better scalability and lower scheduling overhead of the proposed approach comparatively to an existing real-time deadline-oriented scheduling class for the Linux kernel.

Dynamic Global Scheduling of Parallel Real-Time Tasks

Luís Nogueira, José Carlos Fonseca, Cláudio Maia, Luís Miguel Pinho
CISTER Research Centre/INESC-TEC
School of Engineering (ISEP), Polytechnic Institute of Porto (IPP), Portugal
{lmn,jcnfo,crrm,lmp}@isep.ipp.pt

***Abstract*—High-level parallel languages offer a simple way for application programmers to specify parallelism in a form that easily scales with problem size, leaving the scheduling of the tasks onto processors to be performed at runtime. Therefore, if the underlying system cannot efficiently execute those applications on the available cores, the benefits will be lost.**

In this paper, we consider how to schedule highly heterogenous parallel applications that require real-time performance guarantees on multicore processors. The paper proposes a novel scheduling approach that combines the global Earliest Deadline First (EDF) scheduler with a priority-aware work-stealing load balancing scheme, which enables parallel real-time tasks to be executed on more than one processor at a given time instant. Experimental results demonstrate the better scalability and lower scheduling overhead of the proposed approach comparatively to an existing real-time deadline-oriented scheduling class for the Linux kernel.

I. Introduction

The advent of multicore technologies has resulted in a renewed interest on parallel programming and dynamic task parallelism is steadily gaining popularity as a programming model for multicore processors. Intra-task parallelism is easily expressed by spawning threads that the implementation is allowed, but not mandated, to execute in parallel, using frameworks such as OpenMP [1], Cilk [2], Intel's Parallel Building Blocks [3], Java Fork-join Framework [4], Microsoft's Task Parallel Library [5], or StackThreads/MP [6].

The idea behind those frameworks is to allow 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. However, scalable performance is only one facet of the problem in multicore embedded real-time platforms. Predictability and computational efficiency are often conflicting goals, as many performance enhancement techniques aim at boosting the average execution time, without considering potentially adverse consequences on worst-case execution times.

Therefore, parallel programming models introduce a new dimension to real-time multicore scheduling, with many open issues to be studied. Recent works on real-time scheduling of parallel tasks define a task as a collection of several regions, both sequential and parallel [7], [8]. A task always starts with a sequential region, which then forks into several parallel independent threads (the parallel region) that finally join in another sequential region. However, these models require that each region of a task contains threads of execution that are of equal length.

In contrast, we consider a more general model of parallel real-time tasks where threads can take arbitrarily different amounts of time to execute. That is, in this paper, different regions of the same parallel task can contain different numbers of threads, regions can contain more threads than the number of cores, and threads can have arbitrarily different execution needs. Therefore, this model is more portable. Indeed, there are many applications for which these conditions hold, and it is this kind of irregular parallelism that is of primary interest for us. The distribution of work and data in such applications cannot be characterised a priori because those quantities are input-dependent and evolve with the computation itself. In practice, such real-time applications span a wide spectrum, including radar tracking, autonomous driving, and video surveillance. Applications with these properties pose significant challenges for high-performance parallel implementations, where equal distribution of work over processors and locality of reference are desired within each processor. Nevertheless, as the problem sizes scale and processor speeds saturate, the only way to meet deadlines in such systems is to parallelise the computation.

At the same time, implicit threading languages encourage the programmer to divide the program into short-living threads because doing so increases the flexibility to distribute work evenly across processors. The downside of such fine-grained parallelism is that the total scheduling cost can be significant. The best way to reduce the total scheduling cost is to find the sub-costs that matter most and focus on reducing them. One of the simplest, yet best-performing, dynamic load-balancing algorithms for shared-memory architectures is work-stealing [9]. 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. 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 [9]. Under this assumption, an application running on P processors achieves P-fold speedup in its parallel part, using at most P times more space than when running on one CPU. These results are also supported by experiments [10].

Motivated by these observations, this paper breaks new

ground in several ways. First, it proposes the Real-Time Work-Stealing (RTWS) scheduler that combines the global EDF scheduler with a priority-based work-stealing policy which allows 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.

II. Task-level parallelism in real-time systems

Many real-time applications have a lot of potential parallelism which is not regular in nature and which varies with the data being processed. Parallelism in these applications is often expressed in the form of dynamically generated threads of work that can be executed in parallel. The goal is to allow the programmer to express all the available parallelism and let the runtime system execute the program efficiently. The most difficult task for the programmer is partitioning the program across the multiprocessor system so that the computational load is balanced among the cores. Thus, it is important for the underlying architecture to provide help to the programmer in order to ease this burden.

At the same time, 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 processors. The downside of such fine-grained parallelism is that if the total scheduling cost is too large, then parallelism is not worthwhile. Therefore, having many short-lived threads requires a simple and fast scheduling mechanism to keep the overall overhead low.

However, most results in multiprocessor 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.

Early work in real-time scheduling of parallel tasks [12], [13], [14], [15], [16] 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. Since many details of execution, such as the number of iterations in a loop and the number of threads that will be created in a parallel region are often not known in advance, much of the actual work of assigning parallel tasks to cores must be performed dynamically. Unlike static policies, dynamic processor-allocation policies allow the system to respond to load changes, whether they are caused by the arrival of new jobs, the departure of completed jobs, or changes in the parallelism of running jobs - the last case is of particular importance to us in this paper.

Recently, Lakshmanan et al. [7] proposed a scheduling technique for synchronous parallel tasks where every task is an alternate sequence of parallel and sequential regions with each parallel region consisting of multiple threads of equal length that synchronise at the end of the region. In their model, all parallel regions are assumed to have the same number of parallel threads, which must be no greater than the number of processors. In [8], Saifullah et al. considered a more general task model, allowing different regions of the same parallel task to contain different numbers of threads and regions to contain more threads than the number of processor cores. It still requires, however, that each region of a task contains threads of execution that are of equal length. In contrast, this paper considers a more general model of parallel real-time tasks where threads can take arbitrarily different amounts of time to execute.

Furthermore, both works handle scheduling parallel tasks by decomposing them into sequential subtasks. In [7], this technique requires a resource augmentation bound of 3.42 under partitioned Deadline Monotonic (DM) scheduling. For the synchronous model with arbitrary numbers of threads in parallel regions, the work in [8] proves a resource augmentation bound of 4 and 5 for global EDF and partitioned DM scheduling, respectively. Instead, we try to minimise the scheduling overhead by generating parallelism only when required, *i.e.* when a processor becomes idle.

We believe that achieving predictable good performance for fine-grained task-level parallelism in embedded real-time systems is important for several reasons: (i) an efficient implementation of fine-grained parallelism allows more parallelism to be exploited, which is especially important with the expected increase in core counts in future processors; (ii) the programming model is simplified if programmers do not need to avoid spawning small tasks, which is very difficult when task execution times can not be predicted in advance; and (iii) many real-time systems have periodic serialisation points when input is consumed and output is produced. A natural way to program such a system is to parallelise each interval, which then becomes a parallel region.

Therefore, this paper proposes RTWS, a novel scheduler that integrates a priority-based work-stealing policy into global EDF, enabling parallel real-time tasks to be executed on more than one processor at a given time instant. In addition, to the best of our knowledge, no other real-time scheduler that supports task-level parallelism has actually been implemented within a real operating system. In contrast, we have implemented RTWS in Linux and thus have a real working framework. To ease the algorithm's discussion, the system model and the main principles of the proposed approach are discussed in the next sections, while RTWS is presented in Section V.

III. SYSTEM MODEL

We consider the scheduling of implicit-deadline periodic independent real-time tasks on m identical processors $p_1, p_2, \dots, p_m$ using global EDF. With global EDF, each task ready to execute is placed in a system-wide queue, ordered by non-decreasing absolute deadline, from which the first m tasks are extracted to execute on the available processors.

We primarily consider a synchronous task model, where each task $\tau_1, \dots, \tau_n$ 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 Fig. 1). 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 processors and threads inherit the parent's deadline. 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.

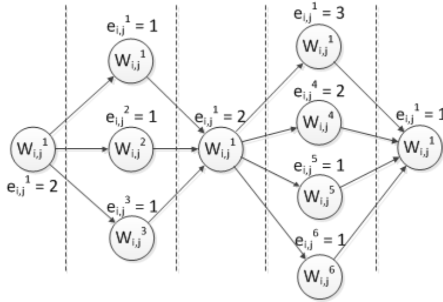

Fig. 1. A multithreaded job with 5 regions

The jth 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 jth job of task τ_i can enter in a parallel region and dynamically generate an arbitrary number of parallel threads which synchronise at the end of that region. A thread is denoted $w_{i,j}^k, 1 \leq k \leq n_i$, where n_i is the total number of threads belonging to the jth job of task τ_i. 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 requirements of a thread $w_{i,j}^k$ of task τ_i is denoted by $e_{i,j}^k$. 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.

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.

The fraction of the capacity of one processor 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 multiprocessor platform Π comprised of m unit-capacity processors 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 multiprocessor platform Π comprised of m unit-capacity processors never misses its scheduling deadline under the following conditions [17]:

$$u_\Pi \leq 1;$$

$$U_\Pi \leq m - u_\Pi(m-1) \tag{1}$$

Naturally, if only soft real-time tasks are considered, jobs may miss their deadlines by bounded amounts, eliminating such restrictive utilisation limits. It has been shown that, when using global EDF to schedule sporadic soft real-time tasks on m processors, deadline tardiness is bounded, provided total utilisation is at most m [18].

IV. TOWARDS REAL-TIME WORK-STEALING

Dynamic scheduling of parallel computations by work-stealing [9] 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 [19], [20].

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 deque than the worker they are stealing from, and also helps to increase locality, since stealing a thread also migrates its future workload [2]. All deque manipulations run in constant-time O(1), independently of the number of threads in the deque. Furthermore, several papers [21], [22], [23] explain how a non-blocking deque can be implemented to limit overheads.

Following [24], we denote T_∞ as the execution time of an algorithm on an infinite number of processors 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 processors verifies Equation 2.

$$T_p \leq \frac{T_1}{p} + T_\infty \quad (2)$$

This time appears asymptotically optimal in the case of very 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 [25], [26], [27]. 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.

Using a single global concurrent priority-based deque is also not viable. While priority queues are often used in single core schedulers, when moving to a parallel context, concurrent priority queues are hard to make both scalable and fast [28]. Furthermore, the semantics of priority queues naturally suggest an ordered insertion method, which is against the work-stealing deque philosophy.

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 steals a thread from the top of the highest-priority non-empty deque of the chosen core's queue.

Two approaches are possible for selecting the victim processor: (i) a probabilistic approach, where the victim is chosen randomly [9]; or a (ii) deterministic approach, where the core is chosen by the priorities of the threads in the deques waiting to be executed. Blumofe and Leiserson [9] 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. However, random selection, while fast and easy to implement, may not always select the best victim to steal from. As core counts increase, the number of potential victims also increases, and the probability of selecting the best victim decreases. This is particularly true under severe cases of work imbalance, where a small number of cores may have more work than others [29]. Moreover, when a thief cannot obtain tasks quickly, the unsuccessful steals it performs waste computing resources, which could otherwise be used to execute waiting threads. In fact, if unsuccessful steals are not well controlled, applications can easily be slowed down by 15%–350% [9]. Therefore, we follow a deterministic approach, following a strict priority schedule.

Definition 1: The set of processors P_s eligible for work-stealing among the set of m identical processors $P = \{p_1, p_2, \ldots, p_m\}$ is given by $P_s = \{p_i | p_s \in P, n_{p_i} \geq 1\}$, where n_{p_i} is the number of threads in the local priority queue of processor p_i.

Having P_s, an idle processor steals the earliest deadline thread w_{edf} among the ones in the top of the highest-priority non-empty deques (first entry in each of the processor's local priority queue) from the set of eligible processors P_s.

Definition 2: The earliest deadline thread w_{edf} from the set of eligible processors P_s is defined as $\exists^1 w_{edf} \in P_s : min_{d_k^r}(P_s), P_s \neq \emptyset$.

Note that the $\exists^1$ relation is guaranteed by the min function which, whenever there is more than one thread with the same earliest deadline, always returns the first thread on the list.

V. THE RTWS SCHEDULER

One approach to schedule parallel applications using work-stealing is to include the calls to a user-space runtime library that manages the threads themselves explicitly in the application. This technique places a lot of onus on the programmer, requiring that the programmer 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 of mapping the 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 [30].

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} [31] and SCHED_DEADLINE (originally named SCHED_EDF) [32]. 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 [32] 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. The proposed RTWS scheduler 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 processor 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 non-decreasing 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 processor p 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 processor's local priority queue to reduce contention on the global queue.
- **Rule C:** each entry in the processor's local priority queue is a deque, holding one or more threads of the same priority. At any time, a processor 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 processor searches for jobs in the global EDF queue.
- **Rule E:** still, if there is no eligible job in the global EDF queue, the processor will steal the earliest deadline eligible thread from the top of the deterministically selected busy processor's deque.
- **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 processor's local priority queue.

Each released job is enqueued in a system-wide global EDF queue ordered by non-decreasing absolute 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.

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 in 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 condition is true, the job/thread 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 it means that there is a deadline to be met, but also to take advantage of data locality. If the local deques are empty, the global queue is searched. This step assumes that no matter how many threads the other cores in the system still have 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 EDF queue with respect to parallel threads generated on other cores. Recall that we try to minimise the scheduling overhead by generating parallelism only when required, *i.e.* when a processor would be otherwise idle.

Finally, if no work has yet been found, a stealing operation takes place which assures that the top-right parallel thread (*i.e.* the highest priority thread), in the deterministically chosen core, is stolen. This reduces contention, by having stealing cores operating on the opposite end of the deque than the core they are stealing from, and also imputes the load balancing operation costs to the idle core.

A. Scheduling multithreaded jobs with RTWS

Consider the following task set, described by WCET and period, $\tau_1 = (5, 10)$, $\tau_2 = (10, 20)$, and $\tau_3 = (4, 19)$. Task τ_1 executes sequentially for three time units and then spawns two threads which have an execution requirement of one time unit each. Task τ_2 has a sequential execution requirement of two time units and then spawns four threads, with the first and third threads having an execution requirement of one time unit, whereas the second and fourth threads have an execution requirement of three time units. Finally, task τ_3 only executes sequentially. Note that the task set is schedulable under global EDF, $u_\Pi = 0.5$ and $U_\Pi = 1.21$.

Fig. 2 depicts a possible schedule generated by RTWS for those three tasks in two identical processors. For the sake of simplicity of presentation, the first thread of each fork-join task does not execute any code in a parallel region.

All tasks are released at $t = 0$. The ones with a lower deadline, τ_1 and τ_3, are selected for execution in the two cores. In the interval $t = [0, 5]$ none of the cores is idle. Therefore, task τ_1 executes sequentially, although it spawns parallel threads. At $t = 5$, task τ_2 is scheduled for execution in core 1. Its sequential part executes until $t = 7$ and then it spawns four threads. As core 2 is idle at time $t = 7$ and there is pending work in the priority queue of core 1, it is able to work-steal. Therefore, at $t = 7$, core 2 steals $w^2_{2,1}$ from the highest-priority non-empty deque of core 1.

At $t = 10$, a job from task τ_1 is released and preempts $w^3_{2,1}$, which has a lower priority. According to the RTWS policy,

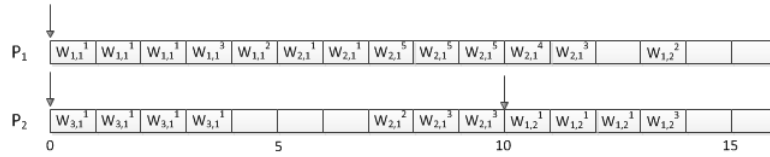

Fig. 2. Scheduling multithreaded jobs with RTWS

$w_{2,1}^3$ is enqueued in the global queue until one of the cores is able to finish its execution. In the depicted example, $w_{2,1}^3$ is executed at $t = 11$ in core 1.

As core 2 is idle after $t = 12$, threads generated by the second job of task τ_1 can be executed in parallel by both cores, by work-stealing at time $t = 13$.

VI. Experimental evaluation

Based on the design principles presented in the previous section, we have implemented RTWS in the standard Linux kernel 2.6.36 as a new scheduling class called SCHED_RTWS. The experiments reported in this paper were conducted in a machine equipped with 16 GB of main memory and an eight-core processor, where each of the cores is running at 2.0 GHz. The Linux kernel was configured as follows: disabled CPU frequency scaling, hyper-threading, 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 on it to set equal grounds.

A set of three major experiments was conducted, where in each of the experiments twenty random task sets were used, running in 2, 4 and 8 cores. In order to dynamically generate the task sets, we have defined the minimum task utilisation (u_{min}) equal to 0.1, the maximum task utilisation (u_{max}) equal to 0.5, a minimum period (T_{min}) of 700 ms, and a maximum period (T_{max}) of 800 ms. The period T_i of each task was computed as $T_i = T_{min} + x * (T_{max} - T_{min})$, where x denotes a random value between 0 and 1.

In order to analyse the scalability of the proposed approach with respect to the number of tasks/threads in the system, until the maximum system utilisation calculated by Equation 1 is reached, three utilisation windows ($[U_{\Pi min}, U_{\Pi max}]$) were chosen: $[0.38, 0.40]$, $[0.58, 0.60]$ and $[0.73, 0.75]$. The tightness of the chosen intervals is justified by the need to ensure similarities between task sets within the same experiment. 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 $\sum_{k=1}^{n} u_k \geq U_{\Pi min}$ and $\sum_{k=1}^{n} u_k \leq U_{\Pi max}$. Finally, C_i is given by $C_i = T_i * u_i$.

The number of parallel threads per task was dynamically derived as $n_i = x * (m * 2)$, whereas the number of tasks (n) was totally dynamic, based on the system utilisation window condition being satisfied (please refer to Table I). Note that as we keep increasing $U_{\Pi max}$, and u_{max} remains constant, n scales. We strongly believe that these parameters can deeply assess our scheduler features.

m	Total tasks			Total threads		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	50	82	98	128	218	216
4	102	158	193	457	703	866
8	217	320	401	1736	2720	3491

TABLE I
Composition of each experiment

Each task was a simple fork-join application whose actual work was limited to a series of NOP instructions to avoid memory and cache interferences. Each of the task's jobs (i) executes sequentially; (ii) splits into multiple parallel threads; and (iii) synchronises at the end of the parallel region, resuming the execution of the master thread. Sequential, parallel, and total execution times were derived randomly, with the actual total execution time upper bounded by C_i.

Data was collected and averaged concerning the number of context switches and migrations, parameters which represent the main sources of scheduling overhead. Fig. 3 depicts the average number of migrations that occurred for each scheduling policy when all cores were online. In the case of SCHED_RTWS, the number of migrations refers to the number of steals performed by the idle cores, while the values collected for SCHED_DEADLINE refer to pure migrations that occurred between the cores.

The overall results show that SCHED_RTWS outperforms SCHED_DEADLINE in every experiments. These results can be explained by our decision to favour data locality, generating parallelism only when strictly required, *i.e.* when a core becomes idle. In fact, the results are far better for medium/high workloads since load balancing calls are more frequently required on SCHED_DEADLINE with the greater number of tasks. Remarkably, the number of migrations barely increases on SCHED_RTWS under such heavy circumstances. For lower workloads, the difference becomes slighter mainly because on our scheduling policy the system lacks parallel threads to keep all cores busy.

Regarding the average number of context switches, depicted in Fig. 4, no matter the considered workload rate, SCHED_RTWS also outperforms SCHED_DEADLINE on eight cores. SCHED_DEADLINE blindly assigns new jobs of a task to the core where the last job of that task was executed, which rather frequently leads to a preemption of the running job. Contrariwise, in SCHED_RTWS, preemptions are minimised because a released job is assigned to a idle core (if available) or inserted into the global queue when its priority 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

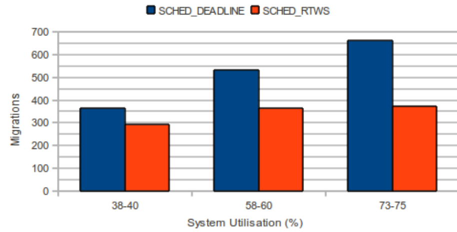

Fig. 3. Average number of migrations on 8 cores

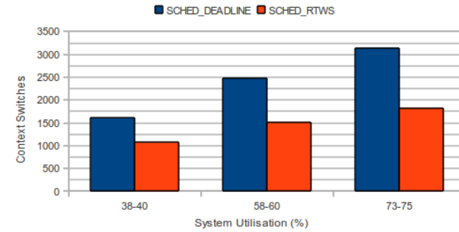

Fig. 4. Average number of context switches on 8 cores

of context switches increases with higher system utilisations, values indicate a less than linear scalability for both policies, which can be seen as a good behaviour.

Using the obtained results of both scheduling policies in two cores as the base case, we have measured the scalability of SCHED_RTWS and SCHED_DEADLINE when scheduling the parallel task sets described in Table I. The obtained results are depicted in Tables II and III. Note that, in both tables, a scale up ratio of two means that the considered metric has doubled.

m	SCHED_RTWS			SCHED_DEADLINE		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	1	1	1	2.75	3.27	3.08
4	5.88	5.55	5.92	11.38	12	13
8	36.38	33	31.08	45.38	48.36	55.08

TABLE II
SCALE UP RATIOS ON NUMBER OF MIGRATIONS

According to the values reported in Table II and considering the properties of our experiments, one can conclude that the number of migrations is largely influenced by the number of dynamically generated parallel threads. Provided that we create more tasks when m is increased, the number of threads exponentially grows as can be easily seen in Table I. Nonetheless, this growth factor is not directly proportional to the scale up ratio. Note the reaction triggered by C_i being constant in every experiment: the more we parallelise, the less executing time will be assigned to each thread, faster threads will finish, migrations will scale. Thereby, we have to multiply the ratio of the system's total number of threads by the ratio of each task's maximum number of threads to be able to find the linear scalability value. For example, for $m = 4$ and a utilisation interval $[0.38, 0.40]$, the scale up ratio is expected to be $\frac{457}{128} * \frac{8}{4} = 7.14$. After analogously calculating for the remaining cases, it is clear that SCHED_RTWS efficiently scales as respects to the number of migrations.

m	SCHED_RTWS			SCHED_DEADLINE		
	38-40%	58-60%	73-75%	38-40%	58-60%	73-75%
2	1	1	1	1.35	1.40	1.33
4	2.91	2.80	3	4.43	4.45	4.74
8	10.60	9.97	10.75	15.93	16.36	18.51

TABLE III
SCALE UP RATIOS ON NUMBER OF CONTEXT SWITCHES

Under global EDF, context switches occur either when a job is released or when it completes. However, not every job release will swap the currently executing job. Thus, the number of context switches over a time interval of length L is upper bounded by twice the number of jobs' releases during that interval. As every experiment has lasted exactly the same time and its periodicity parameters were constant, the scale up ratio on the number of jobs is given by the scale up ratio on the number of tasks. Intuitively, for $m = 4$, SCHED_RTWS scales in a very efficient manner, as Table III reflects, since there are approximately twice more tasks (*e.g.* $\frac{158}{82} = 1.93$) but the scale up ratios on the number context switches are lower than the upper bounded value of 4. Following the same logic, for $m = 8$ our scheduling algorithm appears to scale poorly because the amount of tasks is almost four times higher ($\frac{217}{50} \approx \frac{320}{82} \approx \frac{401}{98} \approx 4$). Nevertheless, recall that in RTWS stolen parallel threads may also preempt any schedulable entity, plus we still have to account each thread's completion as a context switch, seriously inflating the upper bounded scale up ratio from global EDF. In this case, it is particularly noticeable by having to dispatch an incredibly high number of threads, which in turn also potentiates work-stealing (please refer to Table I and Fig. 4 again).

VII. Conclusions and Future Work

It is expected that parallel workloads to become rather common as multicore platforms become ubiquitous. In contrast to prior work on real-time scheduling of parallel workloads, this paper considered a more general model of parallel real-time tasks where dynamically generated threads can take arbitrarily different amounts of time to execute. It proposed RTWS, a novel scheduling policy 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 significantly reduces the scheduling overhead through an efficient and scalable control of migrations and context switches, while still achieves good dynamic load balancing even with low communication costs. Nonetheless, we will conduct further experiments to evaluate more metrics, such as worst-case response time and task latency.

As the complexity of multicore systems grows, it would be interesting to evaluate RTWS in large multicore systems that are likely to have hierarchical cache layouts. One possible

extension to RTWS for such systems could be a scheduling approach that mixes aspects of partitioning and global scheduling. In particular, while task migrations within a cluster of cores that share some lower level cache might be acceptable, migrations among processors that are "far apart" in the cache hierarchy may be too expensive.

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 project, ref. FCOMP-01-0124-FEDER-020447 and by the EU ARTEMIS JU funding, within RECOMP project, ref. ARTEMIS/0202/2009, JU Grant nr. 100202

REFERENCES

[1] O. ARB, "Openmp," Available at http://www.openmp.org/.
[2] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 212–223, 1998.
[3] I. Corporation, "Parallel building blocks," Available at http://software.intel.com/en-us/articles/intel-parallel-building-blocks/.
[4] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, 2000, pp. 36–43.
[5] M. Corporation, "Task parallel library," Available at http://msdn.microsoft.com/en-us/library/dd460717.aspx.
[6] K. Taura, K. Tabata, and A. Yonezawa, "Stackthreads/mp: integrating futures into calling standards," *ACM SIGPLAN Notices*, vol. 34, no. 8, pp. 60–71, 1999.
[7] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proceedings of the 31st IEEE Real-Time Systems Symposium*, December 2010, pp. 259 –268.
[8] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, December 2011, pp. 217 –226.
[9] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, September 1999.
[10] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling scalability and performance in a large scale cmp environment," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 73–86, June 2007.
[11] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, pp. 35:1–35:44, October 2011.
[12] G. Manimaran, C. S. R. Murthy, and K. Ramamritham, "A new approach for scheduling of parallelizable tasks inreal-time multiprocessor systems," *Real-Time Systems Journal*, vol. 15, pp. 39–60, July 1998.
[13] O.-H. Kwon and K.-Y. Chwa, "Scheduling parallel tasks with individual deadlines," in *Algorithms and Computations*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1995, vol. 1004, pp. 198–207.
[14] W. Y. Lee and H. Lee, "Optimal scheduling for real-time parallel tasks," *Transactions on Information and Systems*, vol. E89-D, pp. 1962–1966, June 2006.
[15] S. Collette, L. Cucu, and J. Goossens, "Integrating job parallelism in real-time scheduling theory," *Information Processing Letters*, vol. 106, pp. 180–187, May 2008.
[16] S. Kato and Y. Ishikawa, "Gang edf scheduling of parallel task systems," in *Proceedings of the 30th IEEE Real-Time Systems Symposium*, December 2009, pp. 459 –468.
[17] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Systems Journal*, vol. 25, pp. 187–205, September 2003.
[18] P. Valente and G. Lipari, "An upper bound to the lateness of soft real-time tasks scheduled by edf on multiprocessors," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, December 2005, pp. 311–320.
[19] D. Neill and A. Wierman, "On the benefits of work stealing in shared-memory multiprocessors," Department of Computer Science, Carnegie Mellon University, Tech. Rep., 2009.
[20] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval, "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, pp. 517–518.
[21] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the 10th annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1998, pp. 119–129.
[22] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, 2005, pp. 21–28.
[23] D. Hendler, Y. Lev, M. Moir, and N. Shavit, "A dynamic-sized nonblocking work stealing deque," *Distributed Computing*, vol. 18, pp. 189–207, February 2006.
[24] R. D. Blumofe and C. E. Leiserson, "Space-efficient scheduling of multithreaded computations," in *Proceedings of the 25th ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1993, pp. 362–371.
[25] Z. Vrba, P. Halvorsen, and C. Griwodz, "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, pp. 925–930.
[26] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "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, pp. 1–12.
[27] v. Vrba, H. Espeland, P. Halvorsen, and C. Griwodz, "Limits of work-stealing scheduling," in *Proceedings of the 14th International Workshop on Job Scheduling Strategies for Parallel Processing*, May 2009, pp. 280–299.
[28] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," The University of Texas at Austin, Department of Computer Sciences, Tech. Rep. TR-11-39, November 2011.
[29] A. Bhattacharjee, G. Contreras, and M. Martonosi, "Parallelization libraries: Characterizing and reducing overheads," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 1, pp. 5:1–5:29, February 2011.
[30] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang, "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, pp. 365–378.
[31] 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*, 2006, pp. 111–126.
[32] D. Faggioli, M. Trimarchi, and F. Checconi, "An implementation of the earliest deadline first algorithm in linux," in *Proceedings of the 2009 ACM symposium on Applied Computing*, March 2009, pp. 1984–1989.