On the use of Work-Stealing Strategies in Real-Time Systems

Luis Miguel Nogueira

Luis Miguel Pinho

José Fonseca

Cláudio Maia

CISTER-TR-130110

Version:

Date: 01-23-2013

On the use of Work-Stealing Strategies in Real-Time Systems

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

CISTER Research Unit

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: luis@dei.isep.ipp.pt, lmp@isep.ipp.pt, jaf@det.ua.pt, crrm@isep.ipp.pt

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

Abstract

Computers across all domains increasingly rely on multiple processors/cores, with processors starting to appear with dozens of cores, and next generations of multicore processors expected to integrate hundreds to thousands of simple processors into a single chip. The real-time and embedded systems domain is no exception to this trend. Therefore, the problem of scheduling realtime tasks is no longer a problem of scheduling sequential tasks, but increasingly one of scheduling job-level parallelism or intra-task parallelism as it is commonly known, i.e. the possibility of having more than one core executing a single job at any given instant in time. The subject of scheduling parallel execution is well-studied in the high-performance domain, being work-stealing strategies a simple, yet best-performing, dynamic load-balancing scheme. Nevertheless, traditional work-stealing approaches have inherent limitations for real-time systems. In this paper, we review current work by the authors being done to integrate work-stealing with real-time systems scheduling.

On the use of Work-Stealing Strategies in Real-Time Systems

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

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

Abstract. Computers across all domains increasingly rely on multiple processors/cores, with processors starting to appear with dozens of cores, and next generations of multicore processors expected to integrate hundreds to thousands of simple processors into a single chip. The real-time and embedded systems domain is no exception to this trend. Therefore, the problem of scheduling real-time tasks is no longer a problem of scheduling sequential tasks, but increasingly one of scheduling job-level parallelism or intra-task parallelism as it is commonly known, i.e. the possibility of having more than one core executing a single job at any given instant in time. The subject of scheduling parallel execution is well-studied in the high-performance domain, being work-stealing strategies a simple, yet best-performing, dynamic load-balancing scheme. Nevertheless, traditional work-stealing approaches have inherent limitations for real-time systems. In this paper, we review current work by the authors being done to integrate work-stealing with real-time systems scheduling.

1 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. And these execution needs may also vary between instances (jobs) of the same task. 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 parallelize 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 presents a review of the work which is currently being performed to integrate work-stealing strategies with real-time systems scheduling to allow parallel real-time tasks to be dynamically executed in more than one processor at a given time, whilst guaranteeing deterministic behaviour. To the best of our knowledge, those works are the first research to focus on this subject. And while several have previously considered work-stealing as a load balancing mechanism for parallel computations, those works are the first to do so considering different tasks' priorities and deadlines.

2 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 Earliest Deadline First (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 cannot 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.

3 A work-stealing global EDF scheduling approach

In the first approach [17], we consider the scheduling of implicit-deadline periodic independent real-time tasks on *m* identical processors $p_1, p_2, \ldots, p_m$ 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 $\tau_1, \ldots, \tau_n$ 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 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.

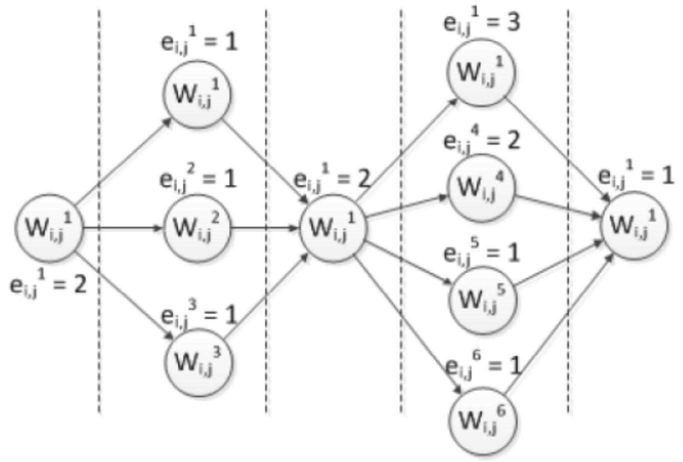

Fig. 1. A multithreaded job with 5 regions

The 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 $a_{i,j} < r_{i,j} < s_{i,j} < f_{i,j}$. 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 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 $w^k_{i,j}$, $1 \leq k \leq n_{i,j}$, where $n_{i,j}$ is the total number of threads belonging to the j^{th} job of task τ_i. We assume $n_{i,j} > 1$ holds for at least one task τ_i in the system. Otherwise, the considered task set does not have intra-task parallelism. As depicted in Figure 1, our work is currently limited to non-nested parallel regions.

The execution requirements of a thread $w^k_{i,j}$ of task τ_i is denoted by $e^k_{i,j}$. 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.

A work-stealing scheduler employs a fixed number of worker threads, 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 [18], [19], [20] explain how a non-blocking deque can be implemented to limit overheads.

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 [21], [22], [23]. 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 [24]. Furthermore, the semantics of priority queues naturally suggest an ordered insertion method, which is against the work-stealing deque philosophy.

The RTWS algorithm [17] replaces 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 [25]. 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. In [26] we provide an initial work which considered a random based approach.

4 A work-stealing server-based scheduling approach

The second approach [27] targets more open environments, where there is little previous knowledge about real execution requirements, and processing capacities are then typically allocated based on average-case execution times, with the result that the expected (mean) tardiness of a task is bounded [28]. Nevertheless, as increasing real-time applications mix different timing criticality in the same system, it is still necessary to isolate and protect the temporal behaviour of one application from the others.

For this, two-level scheduling schemes, commonly known as bandwidth servers, are often used. In [29], Mercer et al. propose a scheme based on capacity reserves to remove the need of knowing the WCET of each task under the Rate Monotonic scheduling policy. A reserve is a couple (C_i, T_i) indicating that a task τ_i can execute for at most C_i units of time in each period T_i. If a task job needs to execute for more than C_i, the remaining portion of the job is scheduled in background.

Based on a similar idea of capacity reserves, Abeni and Buttazo [30] proposed the Constant Bandwidth Server (CBS) scheduler to handle soft real-time requests with variable or unknown execution behaviour under EDF scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task gets a fraction of the CPU and it is scheduled in such a way that it will never demand more than its reserved bandwidth, independently of its actual requests. This is achieved by assigning each soft task a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded. As a consequence, overruns occurring on a served task will only delay that task, without compromising the bandwidth assigned to other tasks.

Resource reservation approaches have been considerably extended, but of which M-CBS [31], M-CASH [32], and EDF-HSB [33] specifically target multicore systems. However, while these resource reclaiming schemes allow tasks to efficiently execute on the same multicore system, they do not allow an individual task to take advantage simultaneously of several cores, preventing task-level parallelism.

The p-CSWS algorithm [27] extends M-CBS and combines a residual capacity reclaiming scheme with a priority-aware work-stealing policy which, while ensuring isolation among tasks, enables parallel tasks to be executed on more than one processor at a given time instant. This way, it is possible to have parallel and non-parallel tasks with different levels of temporal criticality coexisting in the same system, while achieving the goals of temporal isolation and real-time execution. Contrarily to the RTWS algorithm in the previous section, this approach considers the scheduling of multithreaded real-time jobs without any previous knowledge about their real execution requirements, number of parallel regions, and when and how many threads will be generated at each parallel region.

This approach considers instead the scheduling of sporadic independent *servers* on the *m* identical processors $p_1, p_2, ..., p_m$ still using global EDF. A multithreaded job is modelled as a dynamic Directed Acyclic Graph (DAG), defined as $G = (V, E)$, where V is a set of nodes and E is a set of directed edges, both created at runtime. A node represents a thread, a set of instructions which must be executed sequentially. Jobs may dynamically create an arbitrary number of threads, which may have different execution requirements. Therefore, the worst case execution time (WCET) for the j^{th} job of task τ_i is the sum of the execution requirements of all of its threads, if all threads are executed sequentially in the same core.

A directed edge $(a,b) \in E$ represents the constraint that b's computation depends on results computed by a. Therefore, a living thread may either be ready or stalled due to an unresolved dependency. Because multithreaded jobs with arbitrary dependencies can be impossible to schedule efficiently, we limit our study to fully-strict computations. Any multithreaded computation that can be executed in a depth-first manner on a single processor can be made fully-strict by altering the dependency structure, possibly affecting the achievable parallelism, but not affecting the semantics of the computation [9].

All multithreaded jobs generated by a task τ_i are dedicated to a p-CSWS server S_i, an extension for the parallel case of the M-CBS algorithm. Each p-CSWS server S_i is characterised by a pair (Q_i, T_i), where Q_i is the server's maximum reserved capacity and T_i its period. The ratio $U_i = Q_i / T_i$ is known as the server's bandwidth and denotes the fraction of the capacity of one processor that is assigned to the server.

Note that if the needed execution time (WCET) and the minimum inter-arrival time of jobs are known beforehand, it is possible to guarantee the deadline of hard tasks by assigning its server a proper pair (Q_i, T_i). For soft real-time tasks, the timing constraints can be more relaxed. In this case, we do not require an a priori upper bound on the value of $e_{i,j}$ and for soft real-time tasks the pair (Q_i, T_i) can be set based on the served tasks' expected average values. Recall that there is still the goal of providing isolation among the servers and to guarantee a certain degree of service to each individual server. If a job does not receive an allocation of $e_{i,j}$ time units before its implicit deadline $d_{i,j}$, then it is tardy. If a job executes for $e_{i,j} < Q_i$ time units, the resulting unused capacity is referred to as dynamic *residual capacity* and is released to the global queue to be reused in the system.

With global EDF, each server ready to execute is placed in a system-wide queue, ordered by non-decreasing absolute deadline, from which the first m servers are extracted to execute on the available processors. Dynamically generated ready threads are maintained in a local work-stealing double-ended queue (deque) of the server where the job is currently being executed, thus reducing contention on the global queue. For any busy server, parallel threads are pushed and popped from the bottom of the deque and these operations are synchronisation free. Each p-CSWS server successively dequeues a thread from the head of its deque, executes it, and continues with the next thread unless the deque is empty.

At runtime, the performance of the system is enhanced through a novel redistribution of residual capacities that not only lessens tardiness for soft real-time tasks and quickly adapts to load changes, but also enables parallel tasks to be executed on more than one processor at a given time instant. For that, the p-CSWS scheduler considers a second type of servers named residual capacity work-stealing servers. A residual capacity server is a p-CSWS server that applies a priority-based work-stealing policy whenever its local deque is empty.

These servers are created whenever residual capacities greater than a lower bound Q_{min} are released to the global queue. Whenever a residual capacity server S^{r}_{j} is enqueued in the global queue it competes for processor time as if it were a regular active server with pending work and deadline at time d^{r}_{j}. If a residual capacity server is selected for execution, then it may execute only prior to time d^{r}_{j} and the processor time it receives can be consumed by any eligible thread with a current deadline at least d^{r}_{j}, through work-stealing.

Due to work-stealing overheads, not every amount of residual capacity can be efficiently released as a new residual capacity server. Thus, residual capacities smaller than Q_{min} are assigned to the processor on which it was generated and will be consumed by the next server with a later deadline that executes on that processor, in a similar fashion of the residual capacity reclaiming scheme of M-CASH. This allows small capacities to accumulate into usable chunks, avoiding excessive overheads.

If a processor ever idles and there is any residual capacity server in the global queue, then it dequeues the earliest deadline residual capacity server and executes it without donating the resulting execution to any job/thread. The processor continues to execute the residual capacity server as long as it would otherwise be idle or the capacity is neither exhausted nor expired.

In the server-based approach, the inherent limitations of the traditional stealing approach of work-stealing schedulers with only one deque per-core and random victim selection are bridged with the concept of a *virtual deque*. A *virtual deque* of a p-CSWS server S_i is composed by its local deque and by all the deques of active residual capacity servers that have stolen some thread from S_i at some time instant. Thus, all parallel threads of job $j_{i,k}$ continue to be dedicated to the same server S_i, ensuring isolation among tasks.

As with any p-CSWS server, a residual capacity server dequeues a thread from the head of its deque, executes it, and continues with the next thread unless the deque is empty. Similarly, all dynamically generated ready threads are pushed to the bottom of the residual server's deque. Therefore, a residual capacity server follow the same rules of operation as a regular p-CSWS server, except when (i) it finds its local deque empty, since it tries to work-steal; and (ii) when its capacity is exhausted or expired, since it is not replenished. Thus, in order to efficiently manage the virtual deque of a p-CSWS server, whenever a steal occurs, a pointer to the bottom of the residual capacity stealing server's deque is added to a thief list of the stolen server. This pointer remains in the list until all work dedicated to the stolen server, currently in the residual capacity server's deque, has been executed (a residual capacity server only remains active if there is some pending work, even if its capacity is exhausted or expired. Otherwise, the residual capacity server no longer exists.

Whenever a server *Si* finds its local deque empty, it verifies its thief list. If not empty, S_i follows the first pointer in the thief list, iteratively removing and executing the parallel threads from the top of the pointed residual capacity server's deque. Whenever a pointed deque has no more parallel threads dedicated to S_i, the pointer is removed from the server's thief list, and the next pointer is followed, until no more pointers exist.

5 Conclusions

The real-time and embedded systems domain is increasingly considering the problem of scheduling job-level parallelism / intra-task parallelism, i.e. the possibility of having more than one core executing a single job at any given instant in time. This paper presented two approaches which have been recently proposed to integrate work-stealing with real-time systems scheduling.

The first approach considers global EDF based scheduling where jobs can be executed in parallel with work-stealing, whilst the second applies a similar concept to server-based scheduling approaches.

This latter approach already supports nested parallel regions, a problem which is still being analysed for the former, where dequeues are per core, thus introducing extra complexity for handling priorities correctly.

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. 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 multi-threaded 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 realtime 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. L. Nogueira, J. C. Fonseca, C. Maia, L. M. Pinho “Dynamic Global Scheduling of Parallel Real-Time Tasks”, 10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, Cyprus, December 2012, to appear
18. 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.
19. 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.
20. 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.
21. 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.
22. 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.
23. V. Vrba, H. Espeland, P. Halvorsen, and C. Griwodz, “Limits of workstealing scheduling,” in Proceedings of the 14th International Workshop on Job Scheduling Strategies for Parallel Processing, May 2009, pp. 280–299.
24. 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.
25. 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.
26. J. C. Fonseca, L. Nogueira, C. Maia, L. M. Pinho “Real-Time Scheduling of Parallel Tasks in the Linux Kernel”, CISTER Technical Report, TR-120714, July 2012

27. L. Nogueira, L. M. Pinho, “Server-based Scheduling of Parallel Real-Time Tasks“, 12th International Conference on Embedded Software (EMSOFT 2012) , Finland, October 2012
28. A. Mills and J. Anderson, “A stochastic framework for multiprocessor soft real-time scheduling”, In Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 311 -320, Stockholm, Sweden, April 2010.
29. C. W. Mercer, S. Savage, and H. Tokuda, “Processor capacity reserves: Operating system support for multimedia applications. In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, pages 90-99, May 1994.
30. L. Abeni and G. Buttazzo, “Integrating multimedia applications in hard real-time systems”, In Proceedings of the 19th IEEE Real-Time Systems Symposium, page 4, Madrid, Spain, December 1998.
31. S. Baruah, J. Goossens, and G. Lipari, “Implementing constant-bandwidth servers upon multiprocessor platforms”, In Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium, pages 154-163, September 2002.
32. R. Pellizzoni and M. Caccamo, “M-cash: A real-time resource reclaiming algorithm for multiprocessor platforms”, Real-Time Systems, 40:117-147, 2008.
33. B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In Proceedings of the 19th Euromicro Conference on Real-Time Systems, pages 61 -70, Pisa, Italy, July 2007.