Technical Report

Server-based Scheduling of Parallel Real-Time Tasks

Luis Miguel Nogueira

Luis Miguel Pinho

HURRAY-TR-121001

Version:

Date: 10-07-2012

Server-based Scheduling of Parallel Real-Time Tasks

Luis Miguel Nogueira, Luis 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

Multicore platforms have transformed parallelism into a main concern. Parallel programming models are being put forward to provide a better approach for 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.

It is in this context that this paper proposes a scheduling approach that combines the constant-bandwidth server abstraction with a priority-aware work-stealing load balancing scheme which, while ensuring isolation among tasks, enables parallel tasks to be executed on more than one processor at a given time instant.

Server-based Scheduling of Parallel Real-Time Tasks

Luís Nogueira and Luís Miguel Pinho
CISTER/INESC-TEC
School of Engineering (ISEP), Polytechnic Institute of Porto (IPP)
Porto, Portugal
lmn@isep.ipp.pt, lmp@isep.ipp.pt

ABSTRACT

Multicore platforms have transformed parallelism into a main concern. Parallel programming models are being put forward to provide a better approach for 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.

It is in this context that this paper proposes a scheduling approach that combines the constant-bandwidth server abstraction with a priority-aware work-stealing load balancing scheme which, while ensuring isolation among tasks, enables parallel tasks to be executed on more than one processor at a given time instant.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Design, Algorithms, Theory

Keywords

Real-time systems, Task-level parallelism, Constant-bandwidth servers, Capacity sharing, Work-stealing

1. INTRODUCTION

In contrast to the conventional real-time scheduling theory that focus upon the worst-case analysis of systems that are restricted to execute in strictly controlled environments, there is now the understanding that not all applications need the same degree of real-time support. The constant-bandwidth server abstraction [1] has proved very useful in designing, implementing, and reasoning on systems where tasks can dynamically enter or leave at any time, a paradigm that has been somewhat formalised in the concept of *open real-time* environments [17]. In this approach, each real-time task is assigned a fraction of the computational resources and it is handled by an abstract entity called server to achieve the goals of temporal isolation and real-time execution.

With multicore processors quickly becoming the norm, there have been significant efforts to extend reservation-based real-time scheduling theory to make it applicable to multiprocessor systems as well [5, 39, 18, 24]. Nevertheless, all these works consider task models where real-time tasks use at most a single core at each time instant. The advent of multicore technologies has also resulted in a renewed interest on parallel programming. In fact, dynamic task parallelism is steadily gaining popularity as a programming model for multicore processors. Parallelism is easily expressed by spawning threads that the implementation is allowed, but not mandated, to execute in parallel, using frameworks such as OpenMP [3], Cilk [19], Intel's Parallel Building Blocks [14], Java Fork-join Framework [27], Microsoft's Task Parallel Library [15], or StackThreads/MP [42]. The idea 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, while several models of parallelism have been used in programming languages and Application Programming Interfaces (APIs) few of them have been studied in real-time systems. Recent work on real-time scheduling of parallel tasks define a parallel task as a collection of several regions, both sequential and parallel, with synchronisation points at the end of each region [26, 41]. 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, in this paper we consider a more general model of *fork-join* parallel real-time tasks, where threads within a parallel region can take arbitrarily different amounts of time to execute. Indeed, many real-time applications, such as radar tracking, autonomous driving, and video surveillance, have a lot of potential parallelism which is not regular in nature and which varies with the data being processed. As the problem sizes scale and processor speeds saturate, the only way to meet deadlines in such systems is to parallelise the computation.

EMSOFT'12, October 7–12, 2012, Tampere, Finland.

Irregular parallelism in these applications is often expressed in the form of dynamically generated threads of work that can be executed in parallel, generally represented as a Directed Acyclic Graph (DAG). 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. For task graphs where the number of threads and their actual execution times are not known in advance one must use a dynamic approach to efficiently load-balance the computation. One of the simplest, yet best-performing, dynamic load-balancing algorithms for shared-memory architectures is work-stealing [7]. 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 [7]. 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 [40].

Motivated by these observations, this paper breaks new ground in several ways. It proposes p-CSWS (Parallel Capacity Sharing by Work-Stealing), a novel scheduling approach for parallel real-time runtimes that will coexist with a wide range of other complex independently developed applications, without any previous knowledge about their real execution requirements, number of parallel regions, and when and how many those parallel threads will be generated. Schedulers in these type of systems are therefore required to maintain a certain (quantifiable) level of service for each application, with the exact guarantee depending upon the CPU reservation's parameters. p-CSWS combines a multiprocessor residual capacity reclaiming scheme with a priority-based work-stealing policy which, while ensuring isolation among tasks, allows a task 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.

The remainder of this paper is structured as follows. The next section discusses the current challenges in supporting task-level parallelism in open real-time systems. Section 3 presents the system model. Sections 4 and 5 present the main principles of the proposed approach, while the p-CSWS scheduler is formally presented in Section 6 and proved correct in Section 7. Section 8 validates the effectiveness of p-CSWS through extensive simulations. Finally, Section 9 concludes the paper and discusses future work.

2. TASK-LEVEL PARALLELISM IN OPEN REAL-TIME SYSTEMS

Most results in multiprocessor real-time scheduling concentrate on sequential tasks running on multiple processors or cores [16]. 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 real-time intra-task parallelism, where real-time tasks themselves are parallel tasks which can run on multiple cores at the same time instant.

Different scheduling algorithms and assumptions in parallel real-time scheduling can be found in [32, 25, 28, 13, 23]. Most early work in parallel real-time scheduling makes simplifying assumptions about task models, assuming that the parallelism degree of jobs is known beforehand 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. [26] 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 [41], 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. Nevertheless, it still requires that each region of a task contains threads of execution that are of equal length.

In contrast, in this paper we consider a more general model of parallel real-time tasks where parallel jobs are represented as a DAG and threads (nodes) can take arbitrarily different amounts of time to execute. Also, to the best of our knowledge, we are the first to consider the scheduling of multithreaded real-time jobs in open environments, 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.

The design and implementation of open real-time environments is an active research area in the discipline of real-time computing. As an increasing number of users runs both real-time and traditional desktop applications in the same system, it is necessary to isolate and protect the temporal behaviour of one application from the others.

Conventional real-time scheduling theory has tended to focus upon the worst-case execution time (WCET) analysis of systems that are restricted to execute in strictly controlled environments. This traditional perspective of real-time scheduling theory has served the safety-critical embedded systems community well. However, even on single-core systems, WCET analysis is highly problematic and pessimistic WCET estimates are used. This leads to an under-utilisation of computing resources in practice and severely limits the computational workload that can be supported by the system. The problem is exacerbated on a multicore processor, where the worst-case scenario may be even less likely but even more costly. Such a waste of resources can only be justified for critical systems in which a single missed deadline may cause catastrophic consequences.

Therefore, over the last years, there is a new perspective towards being able to provide significant real-time support within the context of general-purpose operating systems with the understanding that not all applications need the same degree of real-time support. For soft real-time tasks, 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 [35].

Most open real-time environments that have been implemented are based upon two-level scheduling schemes, commonly known as bandwidth servers. In [34], 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 [31] 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 instance needs to execute for more than C_i, the remaining portion of the instance is scheduled in background.

Based on a similar idea of capacity reserves, Abeni and Buttazo [1] proposed the Constant Bandwidth Server (CBS) scheduler to handle soft real-time requests with a variable or unknown execution behaviour under the Earliest Deadline First (EDF) [31] 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.

However, the performance of CBS is highly dependent on the correct allocation of resource shares [9]. If a server completes a task in less than its budgeted execution time no other server is able to efficiently reuse the amount of computational resources left unused. In order to make effective use of the available computational bandwidth, a scheduling methodology that makes use of this residual processing capability is desirable. Therefore, CBS has been extended with several resource reclaiming schemes [30, 10, 33, 11, 29, 38] proposed to support an efficient sharing of computational resources left unused by early completing tasks. Such techniques have been proved to be successful in improving the response times of soft real-time tasks while preserving all hard real-time constraints.

Unfortunately, due to well-known multiprocessor scheduling anomalies [2], adopting the same rules as the uniprocessor case would lead to deadline violations in spite of the fact that the considered task set is schedulable by using a global EDF scheduler. As such, the extension of these reclaiming schemes for the multiprocessor case is not trivial and only a few works have address this subject.

M-CASH [39] is a resource reclaiming mechanism for identical multiprocessor platforms built on top of the M-CBS algorithm [5], an extension for the multicore case of the original CBS algorithm. It holds a global queue of residual capacities ordered by non decreasing absolute deadline. Each time a server becomes idle with an execution capacity greater than zero, a new residual capacity is inserted into the global queue with the current server's deadline and remaining capacity. Each time there is a residual capacity at the head of the queue and one or more servers with deadline greater than or equal to the capacity's deadline are scheduled for execution, those servers consume the capacity instead of their own reserved capacity. Thus, residual capacities are equally distributed across all processors, including idle ones.

EDF-HSB [8] uses a similar residual capacity redistribution method. Jobs that finish early donate their unused capacity to a global capacity queue. These capacities are treated as schedulable entities by the top-level scheduler, *i.e.* they compete for processor time as regular jobs with a deadline d. Whenever a capacity is selected to execute, its processor time is donated to soft real-time tasks that are likely to be tardy and best-effort jobs.

However, while these resource reclaiming schemes allow tasks to efficiently execute on the same multicore host, they do not allow an individual task to take advantage of the several cores. In multicore platforms, an application can rely on increasing its concurrency level to maximise its performance, which often requires the application to divide its work into several short-living work units, which can be mapped to threads or other appropriate scheduling representation, increasing the scheduler's flexibility when distributing work evenly across processors. The downside of such fine-grain 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. Since many details of execution, such as the number of iterations in a loop and the number of threads that will created in a parallel region are often not known in advance, much of the actual work of assigning threads of 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. One technique commonly employed to attempt to accomplish this dynamic load balancing is work-stealing.

As such, although previous works have previously considered residual capacity reclamation schemes in the context of multiprocessors, we are the first to do so within a scheme where reclamation is combined with a work-stealing policy to support parallel multithreaded tasks. Different scheduling algorithms and assumptions in parallel real-time scheduling can be found in [32, 25, 28, 13, 23, 26, 41]. Most work in parallel real-time scheduling assumes that the parallelism degree of jobs is known beforehand and uses this information when making its decisions. In practice, this information is not easily discernible, and in some cases can be inherently misleading. In contrast, p-CSWS allows the system to dynamically respond to load changes, whether they are caused by the arrival of new jobs, the departure of completed jobs, or changes in the parallelism degree of running jobs.

p-CSWS extends M-CBS with a novel residual capacity reclaiming scheme and 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. To ease the algorithm's discussion, the system model and the main principles of the proposed approach are discussed in the next sections while the p-CSWS scheduler is formally presented in Section 6.

3. SYSTEM MODEL

We consider the scheduling of sporadic independent servers on m identical processors $p_1, p_2, \ldots, p_m$ using global EDF. With global EDF, each server ready to execute is placed in a system-wide queue, ordered by nondecreasing absolute deadline, from which the first m servers are extracted to execute on the available processors.

We primarily consider a parallel implicit-deadline task model where each task τ_i in the system can generate a virtually infinite number of multithreaded jobs. A multithreaded job is modelled as a dynamic DAG, defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes and $\mathcal{E}$ is a set of directed edges, both created on the fly 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 $(a, b) \in \mathcal{E}$ 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 [7].

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 [5]. 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 $U_i = \frac{Q_i}{T_i}$ is known as the server's bandwidth and denotes the fraction of the capacity of one processor that is assigned to the server. 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 server bandwidth.

If the needed execution time 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). As such, t_i refers to the minimum inter-arrival time between successive jobs of τ_i so that $a_{i,j+1} \geq a_{i,j} + t_i$ and its execution requirements $e_{i,j}$ are characterised by the task's WCET. Thus, for a hard real-time task τ_i, its dedicated server S_i has a reserved capacity Q_i equal to the task's WCET and a period T_i equal to the task's period.

For soft real-time tasks, the timing constraints are more relaxed. In particular, for a soft task τ_i, t_i represents the expected inter-arrival period between successive jobs. As such, the arrival time $a_{i,j}$ of a particular job is only revealed at runtime and the exact execution requirements $e_{i,j}$ can only be determined by actually executing the job to completion until time $f_{i,j}$. Thus, as with M-CBS, we do not require an *a priori* upper bound on the value of $e_{i,j}$ and for soft real-time tasks, Q_i and T_i are set based on the served tasks' expected average values. Recall that our goal with respect to designing the global scheduler is to be able to provide complete 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 $Q_i - e_{i,j}$ is referred to as dynamic residual capacity.

At each instant t, the following values are associated with a p-CSWS server S_i: (i) its currently assigned deadline d_k^i; and (ii) its remaining execution capacity $0 \leq c_k^i \leq Q_i$. Each time a new job of τ_i arrives, it is enqueued in a FCFS job queue held by S_i. The server is said to be active if its job queue is not empty, otherwise it is idle. Whenever the server is active, the job at the top of the queue is released with deadline equal to d_k^i. Upon reaching 0, the execution capacity of a server S_i is recharged to Q_i and its deadline is incremented by T_i.

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.

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 its empty.

4. SHARING RESIDUAL CAPACITIES

Although the server abstraction is an essential method in open real-time systems for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent on a correct resource allocation. To overcome this limitation, an efficient reclaiming of unused computation times generated by earlier completions is fundamental in order to relax the bandwidth constraints enforced by isolation and efficiently manage overruns.

By the very dynamic nature of open real-time systems, the availability of residual capacities is unknown beforehand and can only be scheduled dynamically when it is detected. Therefore, the proposed algorithm considers two different types of servers: (i) a p-CSWS server for managing the execution of each task in the system; and a (ii) residual capacity work-stealing server for managing each residual capacity that is dynamically generated by the earlier completion of jobs.

A p-CSWS server extends the M-CBS server with a work-stealing deque for supporting the parallel execution of multithreaded jobs. Dynamically generated threads are maintained in a local work-stealing deque of the server where the job is currently being executed. 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. If at time t, S_i finishes the execution of its currently served job without exhausting its reserved execution capacity Q_i and it has no pending work, a residual capacity of $min(c_k^i, d_k^i - t)$ and deadline d_k^i is dynamically generated. By pending work we refer to the case when there exists at least a served job such that its release time is $s_{i,j} \leq t < f_{i,j}$.

Residual capacities greater than a lower bound Q_{min} are released to the global queue as a new residual capacity work-stealing server. A residual capacity server is a p-CSWS server that applies a priority-based work-stealing policy whenever its local deque its empty. Whenever a residual capacity server S_j^r is enqueued in the global queue it competes for pro-

cessor time as if it were a regular active server with pending work and deadline at time d_j^r. If a residual capacity server is selected for execution, then it may execute only prior to time d_j^r and the processor time it receives can be consumed by any eligible thread with a current deadline at least d_j^r, through work-stealing. This subject will be detailed in the next section.

Whenever a residual capacity server is executing a thread, the execution capacity of the threads's dedicated server remains unchanged. If the thread completes while consuming the residual capacity and if that residual capacity is neither expired nor exhausted, the leftover capacity $c_j^r > 0$ may be used to execute another thread.

If the available execution capacity of a residual capacity server is either expired or exhausted it is not replenished. If there is pending work, the residual capacity server remains active until all work has been reclaimed back by the respective dedicated server. Otherwise, it becomes idle and is erased from the system.

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.

5. PRIORITY-BASED WORK-STEALING

Dynamic scheduling of parallel computations by work-stealing [7] 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 [37, 36]. Therefore, it has been widely adopted in both commercial and open-source software and libraries, including Cilk++, Intel TBB, Microsoft Task Parallel Library (TPL) in the .NET framework, and the Java Fork/Join Framework.

A work-stealing scheduler employs a fixed number of workers, usually one per core. Each of those workers has a local double-ended queue (deque) to store ready threads. Workers treat their own deques as a stack, pushing and popping threads from the bottom, but treat the deque of another randomly chosen 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 operate 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 [19]. All queue manipulations run in constant-time (O(1)), independently of the number of threads in the queues. Furthermore, several works (e.g. [4, 12, 22]) have addressed 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 extensions previously proposed in the literature [45, 21, 44]. 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 tail 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 of 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 the concept of a per-server *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. The concept is detailed in the next paragraphs.

Whenever a residual capacity server with execution capacity $c_k^r > 0$ finds its local deque empty, c_k^r can be used to execute any eligible thread with a current deadline at least d_k^r through work-stealing.

DEFINITION 1. *The set of active servers A_r eligible for work-stealing is given by $A_r = \{A_r | A_r \in A, d_l^j \geq d_k^r, c_k^r > 0\}$, where A is the set of all active p-CSWS servers with parallel threads in their local deques, d_l^j is the current deadline of parallel threads on the top of a deque, and d_k^r is the currently assigned deadline of server S_r.*

Having A_r, a residual capacity server S_r with available capacity and an empty local deque steals the thread from the top of the deque of the earliest deadline active server S_{edf} from the set of eligible servers P_r, following a deterministic approach as opposed to the random selection of classical work-stealing.

DEFINITION 2. *The earliest deadline active server S_{edf} from the set of eligible servers P_r is defined as $\exists^1 S_r \in P_r : min_{d_k^r}(P_r), P_r \neq \emptyset$.*

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

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. Recall that 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 S_i 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.

6. THE P-CSWS SCHEDULER

The p-CSWS scheduler extends the Multiprocessor Constant Bandwidth Server (M-CBS), first introduced by Baruah et al. [5], with a powerful residual capacity reclaiming scheme combined with a work-stealing load balancing policy used to allow parallel tasks to execute on more than one processor at the same time instant. Recall that our goal with respect to designing the global scheduler is to be able to provide complete isolation among the servers, and to guarantee a certain degree of service to each individual server.

A single ready queue exists in the system, ordered by non-decreasing absolute deadlines. At each instant, the higher priority (with shorter absolute deadline) servers are scheduled for execution. Execution capacities and deadlines are managed using the following rules:

- **Rule A:** whenever a server S_i changes its state from idle to active at some time t, a test is executed. If $c_k^i < (d_k^i - t)U_i$, no update of deadline and budget is necessary. Otherwise, c_k^i is recharged to Q_i and the new value $d_k^i = t + T_i$ is assigned to its deadline.

- **Rule B:** whenever a server S_i is selected for execution, it picks the thread at the bottom of its deque, dynamically generated by its k^{th} job. While executing it, its budget c_k^i is decreased by the same amount. If the server's capacity is either expired or exhausted, it is recharged to Q_i and its deadline d_k^i is incremented by T_i.

- **Rule C:** whenever a server S_i finds its local deque empty, it verifies its thief list. If non-empty, S_i follows the first pointer in the list, iteratively removing and executing those parallel threads. Whenever a pointed deque has no more parallel threads dedicated to server S_i, the pointer is removed from the thief list, and the next pointer (if present) is followed, until no more pointers exist.

- **Rule D:** whenever a server S_i completes its k^{th} job at time $t < d_k^i$, after having consumed $e_k^i < Q_i$ time units, and it has no pending work, a new residual capacity with capacity $min(c_k^i, d_k^i - t)$ and deadline d_k^i is generated. S_i becomes idle and its remaining reserved capacity c_k^i is set to zero.

- **Rule E:** a new residual capacity less than a lower bound Q_{min} is assigned to the processor in which it was generated. The next active server S_j with a later deadline that executes on that processor consumes the earliest deadline residual capacity prior to consuming its own dedicated capacity. When consuming a residual capacity, server S_j runs with the deadline of the residual capacity. If the processor idles beforehand, or if the capacity expires or is exhausted, it is disposed of.

- **Rule F:** a new residual capacity consisting of at least Q_{min} is released to the global ready queue as a new residual capacity server with an execution capacity of $min(c_k^i, d_k^i - t)$ and deadline d_k^i. Whenever a residual capacity server is enqueued, it immediately competes for processing time as if it were a regular server with deadline d_k^r.

- **Rule G:** if a residual capacity server is selected for execution, it may only execute until time d_k^r and the processor time c_k^r it receives is used to steal and execute the earliest deadline eligible thread with a current deadline at least d_k^r. Whenever a steal occurs, a pointer to the bottom of the deque of the residual capacity server is added to the thief list of the stolen server.

- **Rule H:** whenever a thread is executed by a residual capacity server, it is scheduled using the residual server's capacity c_k^r and deadline d_k^r. As such, the execution capacity c_k^i of its dedicated server S_i remains unchanged. If the execution capacity of the residual capacity server is either expired or exhausted, it is not recharged. If there is pending work, the residual capacity server remains active. Otherwise, it is removed from the system.

- **Rule I:** 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 thread. The processor continues to execute the residua capacity server as long as it would otherwise be idle or the capacity of the residual capacity server is neither exhausted nor expired.

Note that from the point of view of the global scheduler a p-CSWS server performs the same three actions as a M-CBS server: (i) it inserts an execution request in the ready queue each time the server transitions from idle to active; (ii) it removes the execution request when it transitions from active to idle; and (iii) it postpones the deadline of its execution request once its capacity is depleted. Postponing a deadline is effectively equal to removing the current execution request and inserting a new one. Hence, as in [5], we can also abstract the execution requirements of a p-CSWS server as a series of server jobs. However, as opposed to M-CBS, with p-CSWS the computation time of a served multithreaded job can actually be greater than the reserved capacity of its dedicated server and it can be executed in more than one processor at a time, since parallel threads can be stolen and executed by residual capacity servers as well as by its dedicated server.

7. CORRECTNESS

In [5], it is proven that a M-CBS server with parameters (Q_i, T_i) cannot occupy a bandwidth greater than $\frac{Q_i}{T_i}$.

That is, if $D_{S_i}(t_1, t_2)$ is the server's bandwidth demand in the interval $[t_1, t_2]$, it is shown that $\forall t_1, t_2 \in N : t_2 > t_1, D_{S_i}(t_1, t_2) \leq \frac{Q_i}{T_i}(t_2 - t_1)$. This isolation property allow us to use a bandwidth reservation strategy to allocate a fraction of the processor to a task whose demand is not known a priori. The most important consequence of this property is that soft real-time tasks, characterised by average values, can be scheduled together with hard tasks, even in the presence of overloads.

Here, we show that the residual capacity reclaiming scheme and work-stealing policy of p-CSWS do not compromise the real-time correctness of the system.

We start by proving that each p-CSWS server S_i never miss its deadlines, or equivalently, whenever a server deadline d_k^i is reached, its reserved execution capacity is zero. Hence, S_i is able to guarantee an execution time Q_i every T_i time units to its served task.

THEOREM 1. *A server S_i executed on the identical multiprocessor platform Π comprised of m unit-capacity processors never misses its scheduling deadline under the following conditions:*

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

Proof Sketch

The scheduling condition is equivalent to the schedulability bound expressed in [20] for periodic task sets scheduled by global EDF. It has been shown, *e.g.* in [1, 39], that the resulting schedule of a resource reservation-based system is the same as the one of a set of periodic real-time tasks, one per server, each with an utilisation equal to the server's reserved capacity Q_i and period equal to the server's period T_i.

Since the computation time of task τ_i is equal to Q_i, its job $j_{i,k}$ will complete before its deadline leaving zero execution capacity. Then, by following the same reasoning, job $j_{i,k+1}$ will be released to the ready queue at $a_{i,k+1} = d_k^i$, and will therefore meet its deadline as well. Hence, the theorem follows by induction.

□

We now ensure that all generated capacities under p-CSWS are either consumed or exhausted before their respective deadlines.

LEMMA 1. *Given a set of p-CSWS servers, each execution capacity generated during scheduling is either consumed or discharged until its deadline.*

Proof sketch

Let $a_{i,k}$ denote the instant at which a new job $J_{i,k}$ arrives and its associated p-CSWS server $S_i \in I$ is inactive. At $a_{i,k}$, a new capacity $c_k^i = Q_i$ is generated and S_i is released to the ready queue.

Let $\forall_{i,k}\ d_k^i = max\{a_{i,k}, d_{k-1}^i\} + T_i$ be the deadline associated with the generated execution capacity c_k^i.

Let $[t, t+\Delta_t[$ denote a time interval during which server S_i is executing, consuming its own capacity c_k^i. Consequently, S_i has used an amount equal to $c_k^{i'} = c_k^i - \Delta_t \geq 0$ of its own capacity during Δ_t. As such, the server's reserved capacity c_k^i must be decreased to $c_k^{i'}$, until it is exhausted.

Let $f_{i,k}$ denote the time instant at which server S_i completes its job $J_{i,k}$. Assume that there are no pending jobs for server S_i at time $f_{i,k}$ and $c_k^i > 0$. According to rule D, a new residual capacity with capacity $min(c_k^i, d_k^i - t)$ and deadline d_k^i is generated.

According to Rule E, small capacities are assigned to the processor in which they were generated and are consumed by the next server with a later deadline that executes on that processor prior to its own reserved capacity. Assume that, at instant $f_{i,k}$, another active server S_j is scheduled for execution on the same processor. According to Rule E, if the inequality $d_k^i \leq d_l^j$ holds, let $[t, t+\Delta_t[$ denote the time interval during which server S_j is executing, consuming the residual capacity. Consequently, c_k^i must be decreased to $c_k^{i'} = c_k^i - \Delta_t \geq 0$, until the residual capacity is exhausted or the currently assigned deadline d_k^i is reached. In this later case, the residual capacity is depleted.

As for larger capacities that are dealt with via Rule F, the residual capacity dynamically generated on a early job completion can immediately be released as a new residual capacity server S_r. Assume now that, at instant $f_{i,k}$, S_r is scheduled for execution. According to Rule G, if the inequality $d_l^j \geq d_k^r$ holds, let $[t, t+\Delta_t[$ denote the time interval during which server S_r is executing the stolen thread with deadline d_l^j, consuming its own capacity c_k^r. Consequently, c_k^r must be decreased to $c_k^{r'} = c_k^r - \Delta_t \geq 0$, until the capacity of server S_r is exhausted or the currently assigned deadline d_k^r is reached. At deadline d_k^r, any remaining residual capacity c_k^r of server S_r not used is discharged.

□

THEOREM 2. *The dynamic residual capacity reclaiming scheme of p-CSWS does not invalidate the timing guarantees made in Theorem 1.*

Proof Sketch

The theorem follows immediately from Lemma 1. In fact, Lemma 1 ensures that each generated capacity is always exhausted before or discharged at its deadline.

Small residual capacities that are dealt with Rule E may merely cause the consuming job to consume less of its dedicated execution capacity, which cannot increase tardiness.

As for larger residual capacities that are dealt with via Rule F, then the reasoning is also straightforward. The execution capacity of the residual capacity server is scheduled essentially as it would have been included as part of the execution of the donating job.

Since the worst case response time of a task is independent of whether the reserved capacity of some server is being used by that server to execute its dedicated task or it is being consumed to execute any other task in the system, the system's schedulability is independent of whether the proposed dynamic residual capacity reclaiming mechanism of p-CSWS is in operation or not. In the worst case, the longest time a residual capacity can be used to execute tasks dedicated to other servers is bounded by the original server's capacity and deadline.

□

8. EXPERIMENTAL EVALUATION

Both M-CBS, M-CASH, and p-CSWS have been implemented in Linux, at user-space level, to measure the performance of the proposed approach to support parallel real-time tasks in open real-time systems through extensive simulations. In particular, we have considered a system of 4 processors with 15 periodic tasks where $\tau_1, \ldots, \tau_5$ were hard real-time tasks, each served by a server with maximum budget equal to the task's worst-case execution time, and $\tau_6, \ldots, \tau_{15}$ were parallel soft real-time tasks that could experience overload conditions.

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 (a random number between $[2, 10]$); and (iii) synchronises at the end of the parallel region, resuming the execution of the master thread. This sequence could occur a random number of times between $[1, 3]$.

The actual execution time of each hard task varied between $[0.3 * Q_i, Q_i]$ of its dedicated server's reserved capacity Q_i, according to a Gaussian distribution. Hence, this variation provides a measure of the amount of bandwidth left free by early completing hard real-time jobs. For parallel multithreaded soft tasks, the total execution time varied between $[0.7 * Q_i, 1.8 * Q_i]$ of their respective dedicated server's reserved capacity Q_i, with sequential and parallel execution times being randomly distributed as a function of the chosen total execution time. All periods were chosen to be uniformly distributed in the interval $[600, 6000]$ ms.

In each simulation run, the performance of each algorithm was evaluated by computing the average tardiness for all soft real-time tasks. The global tardiness was computed by averaging over all soft real-time jobs executed in the simulation run.

Since both M-CBS and M-CASH do not support the parallel execution of multithreaded jobs, all dynamically generated threads were sequentially executed by their dedicated server when scheduled by these two algorithms. On the other hand, with p-CSWS, a multithreaded job could be simultaneously executed by both its dedicated server and one or more residual capacity work-stealing servers at the same time instant. Figure 1 shows the obtained results.

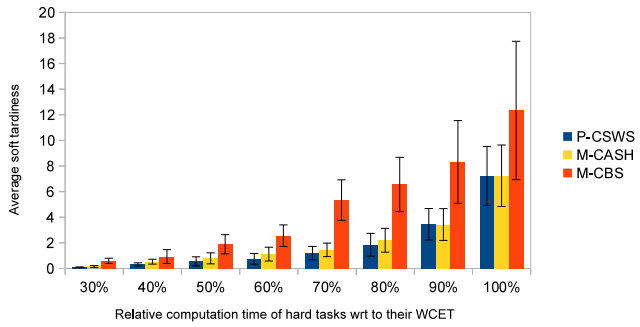

Figure 1: Average tardiness for parallel soft real-time tasks

As expected, the tardiness of soft tasks is smaller when hard tasks execute for less time, thus leaving more residual capacity. Thus, M-CASH significantly outperforms M-CBS, clearly justifying the use of a residual capacity reclaiming mechanism to significantly reduce the average tardiness of soft tasks. However, p-CSWS is always able to obtain better performance than M-CASH. One can conclude that p-CSWS: (i) is indeed able to exploit the available residual bandwidth, thus lowering the average tardiness of soft tasks significantly; and (ii) is able to exploit work-stealing to improve the performance of parallel programs, dynamically balancing the work load among processors. This is particularly for larger amounts of available residual capacity. Naturally, since no residual capacity is left free by hard tasks when they need to execute for the totally of their WCET, the tardiness of parallel soft real-time tasks grows rapidly.

A second study measured the impact of the chosen work-stealing policy on the tardiness of soft real-time tasks. The study considered two work-stealing policies applied to p-CSWS: (i) the classical random choice [7] of a stolen thread; and (ii) the proposed deterministic priority-based selection of the stolen thread. Figure 2 shows the obtained results.

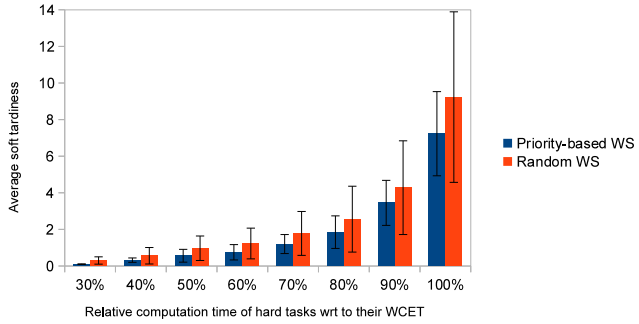

Figure 2: Average tardiness as a function of the chosen work-stealing policy

Similar trends exists for the several relative computation times of hard tasks with respect to their WCET. The bound on average tardiness is by far tighter when stolen threads are chosen according to their deadline rather than randomly. One can conclude that random selection, while fast and easy to implement, may not always select the best victim to steal from. Furthermore, 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 [6]. 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% [7].

9. CONCLUSIONS AND FUTURE WORK

Multiple programming models are emerging to address an increased need for dynamic task parallelism in applications for multicore processors and shared-address-space parallel computing, both in the general purpose and real-time em-

bedded software development. Scheduling algorithms based on work-stealing are gaining in popularity but also have inherent limitations for real-time systems. This paper proposed and proved correct a novel scheduling approach that combines a priority-based work-stealing load balancing policy with a multicore reservation-based approach to support dynamic task-level parallelism in real-time systems.

p-CSWS is particularly suitable to open systems, where independently developed applications can enter and leave the system at any time but, nevertheless, it is important to achieve the goals of temporal isolation and real-time execution among tasks whose resource demands are only know at runtime. The performance of the proposed approach was demonstrated by extensive simulation studies. We are currently pursuing an evaluation of its efficiency in real-world scenarios by implementing it as new scheduling class in the Linux kernel.

Although it is possible to guarantee the schedulability of parallel hard real-time tasks with p-CSWS, in the worst-case we must consider all threads to execute sequentially on its dedicated server, since parallel execution is only possible when some other server releases residual capacity. This kind of guarantee is very pessimistic and leads to an overal-location of resources. We plan to consider the possibility to pre-allocate empty servers with some reserved capacity to immediately take care of spawned threads, so that it could be possible to provide less pessimistic guarantees to hard real-time tasks. Naturally, if only soft real-time tasks are considered, servers may miss their deadlines by bounded amounts, eliminating such restrictive utilisation limits. It has been shown that, when using global EDF to schedule sporadic real-time tasks on m processors, deadline tardiness is bounded, provided total utilisation is at most m [43].

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, ref. FCOMP-01-0124-FEDER-020447 and FCOMP-01-0124-FEDER-015006

10. REFERENCES

[1] 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.

[2] B. Andersson and J. Jonsson. Preemptive multiprocessor scheduling anomalies. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 271, April 2002.

[3] O. ARB. Openmp. Available at `http://www.openmp.org/`.

[4] 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*, pages 119–129, New York, NY, USA, 1998. ACM.

[5] 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.

[6] A. Bhattacharjee, G. Contreras, and M. Martonosi. Parallelization libraries: Characterizing and reducing overheads. *ACM Transactions on Architecture and Code Optimization*, 8(1):5:1–5:29, February 2011.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[8] 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.

[9] G. Buttazzo and E. Bini. Optimal dimensioning of a constant bandwidth server. In *Proceedings of the 27th IEE International Real-Time Systems Symposium*, pages 169–177, Rio de Janeiro, Brasil, December 2006.

[10] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of 21th IEEE RTSS*, pages 295–304, Orlando, Florida, 2000.

[11] M. Caccamo, G. C. Buttazzo, and D. C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, February 2005.

[12] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28, 2005.

[13] S. Collette, L. Cucu, and J. Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106:180–187, May 2008.

[14] I. Corporation. Parallel building blocks. Available at `http://software.intel.com/en-us/articles/intel-parallel-building-blocks/`.

[15] M. Corporation. Task parallel library. Available at `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.

[16] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, October 2011.

[17] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE RTSS*, page 308, Washington, DC, USA, 1997.

[18] D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 90–99, July 2010.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.

[20] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems Journal*, 25:187–205, September 2003.

[21] 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, pages 1–12, April 2010.

[22] D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18:189–207, February 2006.

[23] S. Kato and Y. Ishikawa. Gang edf scheduling of parallel task systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 459 –468, December 2009.

[24] S. Kato, R. Rajkumar, and Y. Ishikawa. Airs: Supporting interactive real-time applications on multicore platforms. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 47–56, July 2010.

[25] O.-H. Kwon and K.-Y. Chwa. Scheduling parallel tasks with individual deadlines. In *Algorithms and Computations*, volume 1004 of *Lecture Notes in Computer Science*, pages 198–207. Springer Berlin / Heidelberg, 1995.

[26] 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*, pages 259 –268, December 2010.

[27] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

[28] W. Y. Lee and H. Lee. Optimal scheduling for real-time parallel tasks. *Transactions on Information and Systems*, E89-D:1962–1966, June 2006.

[29] C. Lin and S. A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE RTSS*, pages 410–421, 2005.

[30] G. Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the 12th EuroMicro Conference on Real-Time Systems*, pages 193–200, Stockholm, Sweden, 2000.

[31] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):40–61, 1973.

[32] 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*, 15:39–60, July 1998.

[33] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 211, Toronto, Canada, 2004.

[34] 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.

[35] 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.

[36] 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*, pages 517–518, New York, NY, USA, 2009. ACM.

[37] D. Neill and A. Wierman. On the benefits of work stealing in shared-memory multiprocessors. Technical report, Department of Computer Science, Carnegie Mellon University, 2009.

[38] L. Nogueira and L. M. Pinho. A capacity sharing and stealing strategy for open real-time systems. *Journal of Systems Architure*, 56(4-6):163–179, 2010.

[39] R. Pellizzoni and M. Caccamo. M-cash: A real-time resource reclaiming algorithm for multiprocessor platforms. *Real-Time Systems*, 40:117–147, 2008.

[40] 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*, 41(3):73–86, June 2007.

[41] 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*, pages 217 –226, Vienna, Austria, December 2011.

[42] K. Taura, K. Tabata, and A. Yonezawa. Stackthreads/mp: integrating futures into calling standards. *ACM SIGPLAN Notices*, 34(8):60–71, 1999.

[43] 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*, pages 311–320, December 2005.

[44] 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*, pages 280–299, May 2009.

[45] 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*, pages 925–930, February 2010.