A Hardware Architecture to Deploy Complex Multiprocessor Scheduling Algorithms

Renato Mancuso, Prakalp Srivastava, Deming Chen and Marco Caccamo
University of Illinois at Urbana-Champaign, USA, {rmancus2, psrivas2, dchen, mcaccamo}@illinois.edu

Abstract—**An increasing demand for high-performance systems has been observed in the domain of both general purpose and real-time systems, pushing the industry towards a pervasive transition to multi-core platforms. Unfortunately, well-known and efficient scheduling results for single-core systems do not scale well to the multi-core domain. This justifies the adoption of more computationally intensive algorithms, but the complexity and computational overhead of these algorithms impact their applicability to real OSes.**

We propose an architecture to migrate the burden of multi-core scheduling to a dedicated hardware component. We show that it is possible to mitigate the overhead of complex algorithms, while achieving power efficiency and optimizing processors utilization. We develop the idea of "active monitoring" to continuously track the evolution of scheduling parameters as tasks execute on processors. This allows reducing the gap between implementable scheduling techniques and the ideal fluid scheduling model, under the constraints of realistic hardware.

I. Introduction

An increasing demand for high-performance systems has been observed in the domain of both general purpose and real-time, safety-critical systems. Overall, the systems involved range from desktop computers to data-centers, and smartphones to embedded platforms. Within the range of different system configurations, purposes and platforms, the problem of scheduling the workload on multiple processing units in parallel is a common and crucial point. A plethora of advanced scheduling techniques has been proposed, extensively studied, and implemented in real-time systems. However, while heavy duty processing algorithms (like video compression) have been promoted to first-class system components and often implemented as a co-processor, schedulers have been traditionally implemented in software at an OS-level. As a result, little work has been done on exploring the potentials of using dedicated hardware as system scheduler.

The increasing demand for high-performance and energy efficient systems has pushed the industry toward a pervasive transition to multi-core platforms. Optimal and efficient scheduling algorithms (namely, RM and EDF) exist for the problem of scheduling real-time tasks on uniprocessor platforms. Unfortunately, many well-known scheduling results for single-core systems do not scale well to multi-core, and often exhibit sub-optimal performance if applied outside the single-core domain due to scheduling anomalies [1]. For example, a scheduling anomaly occurs when an increase in the period of a task (resulting in a lower overall CPU utilization) makes a previously schedulable taskset not schedulable anymore. Due to scheduling anomalies, no algorithm with job-level dynamic priorities is optimal on multiprocessor [2], [3]. It follows that no optimal algorithm for multiprocessor can be easily implemented in an event-driven fashion (i.e. performing scheduling decisions at the job level).

Conversely, in order to achieve optimal resource usage on multiprocessor systems, less restrictive and more computationally intensive algorithms need to be implemented [4]. These belong to the class of those algorithms with unrestricted dynamic priorities and full migrations. However, the complexity and computational overhead of these algorithms impact their applicability to real OSes. For example, consider a system with m processors and n tasks: the most efficient implementation of Pfair [5], known as PD^2 [6], has a complexity of $O(\min(n, m * \log\, n))$. Similarly, POGen [7] - a scheduling algorithm for bus transactions - has a complexity of $O(n^2 * h)$, with h being the length of hyperperiod for the considered n bus transactions.

In this work, we explore the possibilities offered by modern architectures to implement multi-core scheduling policies with unrestricted dynamic priorities on a dedicated hardware module freeing the embedded processors from the scheduling duty. In fact, by migrating the scheduling responsibilities to a specialized hardware module, we show that it is possible to significantly mitigate the undesirable overhead of mentioned algorithms, while achieving power efficiency and optimizing processors utilization. We propose an architecture for hardware schedulers that revolves around the idea of "active monitoring" to achieve a fine granularity on the way task priorities are calculated and updated. This is possible because our architecture continuously tracks the evolution of scheduling parameters as tasks execute on processors. We show how employing active monitoring simplifies the implementation of complex scheduling algorithms with unrestricted dynamic priorities; furthermore, under the constraints of realistic hardware, the proposed hardware scheduling engine helps reducing the gap between implementable scheduling techniques and the ideal fluid scheduling model [8].

The rest of the paper is organized as follows. First, Section II introduces some background concepts about task models and multiprocessor scheduling. Section III reviews prior research work that proposed the adoption of hardware schedulers. Next, in Section IV, a detailed description of the proposed architecture is provided. Section V carries out a case study on the implementation of a LLF hardware scheduler as an instance of the proposed architecture, while an extensive evaluation is provided in Section VI. The paper concludes in Section VII, where we provide the plan for our future work.

II. Background and Motivation

Processor scheduling is the problem of allocating processing time to tasks in a system. In real-time systems, tasks are commonly defined according to the periodic task model, also known as Liu and Layland model [9]. The periodic task model represents the simplest formulation for real-time tasks. Each periodic task is described by two parameters: a worst-case execution time (WCET) e and a period p. At each instance of time that is multiple of p, a new job of the considered task is released. Thus, it becomes active and available to the scheduler for execution. Each released job has to execute for at most e units of time and has to complete before a new job of the same task is released (this means e is strictly smaller than p).

A collection of periodic tasks (a taskset) is said to be feasible on a given number of processors m if there exists an assignment of active tasks to processors, for each instant of time, so that each task meets its timing constraints. A taskset is said to be schedulable by a given algorithm A if A ensures that all the timing constraints for the taskset are met when scheduling on m processors. A scheduling algorithm A is said to be *optimal* if any feasible taskset is determined to be schedulable by A.

The periodic task model is the simplest abstraction of processing tasks that needs to obey strict timing constraints. Less constrained models have been proposed in literature. For instance, in the sporadic task model [10], [11], the period of a task defines the minimum inter-arrival time between two jobs of the same task. An additional parameter d encodes the relative deadline for each released job. Further relaxations have been proposed in [12], [13], [14], [15]. Often, adding expressiveness to a task model directly impacts how tight the schedulability results can be.

A. Multiprocessor Scheduling

Since the manufacturing industry is heavily oriented towards multi-core and many-core processors [16], the problem of efficiently scheduling real-time tasks on multiprocessor platforms is not only related to a tangible need, but is also facing new issues in the scalability of scheduling algorithms. Traditionally, scheduling algorithms for multiprocessors have followed two directions: *partitioning* and *global scheduling*. In partitioned algorithms, tasks are statically assigned to processors, so that each processor can run a local uniprocessor instance of a scheduling algorithm. In this case, optimality results for uniprocessor scheduling are valid in a single partition. However, task allocation to partitions is an NP-hard problem [17]. Moreover, systems with a strict task-to-processor assignment can enter a configuration in which the load on different cores is extremely unbalanced. Global scheduling algorithms, instead, assign priorities to all the tasks in the system and dispatch the top-priority ones on the available processors. Unfortunately, however, it has been shown that optimality results for scheduling algorithms on uniprocessor systems are not valid when multiprocessor platforms are considered [17].

Following the classification provided in [4], scheduling algorithms can be classified according to (A) the way priorities are assigned to tasks and (B) the degree of allowed migration. In this work, we consider only global algorithms with unrestricted migrations (i.e. jobs can be migrated at any time). The priority assignment schemes can vary from static assignment, dynamic at job level, up to dynamic and unrestricted. Rate-Monotonic (RM) [9], Earliest Deadline First (EDF) [9] and Least Laxity First (LLF) [18] are examples from each respective category. It is known that global scheduling algorithms with job-level dynamic priorities cannot be optimal on multiprocessor [2]. It follows that scheduling algorithms belonging to the least restrictive class (with dynamic and unrestricted priorities) are more powerful than algorithms belonging to the former class, since they can exploit multiprocessor resources more efficiently. In general, scheduling policies with unrestricted priorities can generate a high number of preemptions. As a result, tasks can experience poor cache performance. However, extensive studies [19], [20], [21] have shown how cache resources can be managed to be insensitive to process migrations and context-switches.

Since the number of cores in multiprocessor systems continues to grow, the size of schedulable tasksets can increase linearly too. In this case, algorithms that have sub-optimal performance but are implementable with a lower overhead have been largely preferred, since they scale well with the workload size.

For example, global EDF belongs to the class of job-level dynamic algorithms. Thus, it requires that the priority-based ordering of active tasks is maintained only at the boundaries of jobs. It follows that ordered insertions in a queue can be performed, achieving a complexity as low as $O(\log\ n)$. Conversely, let us consider Pfair: an optimal algorithm for multiprocessor with unrestricted dynamic priorities. An implementation for Pfair has been proposed in [6], which has a complexity of $O(\min(n, m * \log\ n))$. The problem of task scheduling has several similarities with scheduling of network flows and bus transactions. To understand how algorithmic complexity affects overhead, the case of POGen [7] - a bus transactions scheduler - represents a good example. POGen uses slotted time and the notion of scheduling interval to perform scheduling: the scheduler is activated at the beginning of each scheduling interval and, each time, a table driven schedule for all the slots within current scheduling interval is computed. A sufficient utilization bound for POGen is $U_{pogen} = \frac{L-1}{L}$, where L is the greatest common divisor (gcd) of all the transaction periods. The complexity of POGen is $O(n^2 * h)$, where h is the length of hyperperiod for the considered n bus transactions. In the case of POGen, if periods were chosen with $gcd = 1\ msec$, scheduling interval set to $L = 1\ msec$ and slot size between $10 - 100\ \mu sec$, the corresponding utilization bound is between 90% and 99% and the measured overhead is up to 3% for $n = 40$ transactions. However, the scheduling overhead becomes quickly unacceptable for a software implementation of POGen if larger transaction sets ($n > 40$) need to be scheduled at run-time. Similarly, the slot size (as in the case of Pfair) affects the run-time overhead of the scheduler.

B. The Need For A Hardware Scheduler

The high complexity factor has caused that algorithms with dynamic and unrestricted priorities have been neglected (from an implementation point of view) in favor of simpler ones sacrificing resource utilization. However, the performance loss in many-core architectures due to the deployment of a restricted scheduling algorithm can become significant and justify a shift in the role that schedulers play in multiprocessor systems. Specifically, we propose an innovative architecture with **hardware scheduling engine** for multiprocessor systems, delegating to a specific hardware circuit the responsibility of efficiently dispatching tasks to cores.

Since the development of silicon-based circuits has hit the frequency wall, resource specialization has been extensively used to optimize power efficiency and offload complexity from general purpose components. Graphic adapters as well as DSPs have already been going down the specialization road for years. Moreover, a task scheduler is a critical component that is mandatory on any system that needs to differentiate between different execution flows. Thus, specialization for schedulers is the natural evolution for architectures with a massive number of cores.

We envision a new paradigm to implement hardware schedulers using "active monitoring", which can have a number of benefits:

1) Through active monitoring, the previously neglected class of complex but powerful global scheduling algorithms with unrestricted dynamic priority becomes practically feasible. Thereby, the proposed architecture represents a pre-requisite toward the deployment of a class of optimal scheduling solutions.
2) RTOSes can be structured in a way that entirely relies on a dedicated circuit for scheduling. This offloads the burden of implementing software schedulers, simplifying their structure and making them less prone to programming bugs. In fact, the dedicated hardware module would represent an highly engineered architectural component, whose behavioral correctness would be extensively tested as a part of processor (family) design. Conversely, such validation effort would be systematically off-loaded from the development process of any scheduler-free RTOS.
3) Specialization can combine the advantages of complex and powerful scheduling algorithms with power efficiency, since processor time is not wasted anymore to multiplex tasks and can be almost completely allocated for useful computation. Similarly to the case of GPUs, a scheduler is a components that: (a) needs to be part of nearly every system; (b) since it performs a very specific task, can be assigned to a simpler unit (e.g. no floating-point unit is normally required); and (c) can be optimized with specialized resources (e.g. a hardware priority queue).

III. Related Work

Hardware schedulers have been largely employed in networking to address the challenge of scheduling packet flows according to QoS and timing requirements [22], [23]. Such systems employ simple fixed priority schemes and the main purpose of a hardware packet scheduler is usually ensuring that packet switching is performed with the minimum overhead. Thus, works in this directions have contributed in developing fast FIFO queues and priority queues that are able to remain consistently sorted upon insertion/deletion of entries.

Experimental schedulers implemented in hardware for real-time purposes have been studied in the past as a way to support accurate tick resolution in task handling and reduce the interrupt handling time by offloading scheduling tasks from CPUs. In the Spring kernel [24], system processors are included in the architecture to offload the scheduling algorithm from processors that have application workload. In [25], [26], SpringNet is presented, where a specific coprocessor that calculates the priority of the task to be scheduled according to adjustable metrics becomes part of the design of a real-time oriented system. Similarly, FASTHARD [27] proposes an implementation of a scheduler for single processor systems in which the scheduling algorithm is entirely computed in hardware using a specialized processor.

In [28], a reconfigurable hardware scheduler that allows choosing among three scheduling algorithms (static priority, RM and EDF) is proposed. The implemented scheduler is used to dispatch periodic tasks on a single-core system. Moreover, a high-level user interface is provided to create the desired RTOS/scheduler configuration to be deployed on target systems. Simulation experiments are carried out to validate the implementation of scheduler and RTOS.

An external scheduler, called Booster, is developed in [29]. The proposed scheduler runs on an external peripheral connected to the CPUs through a communication bus. Although the scheduling is offloaded, the communication delay still makes a fine-grained control of tasks unfeasible and its accuracy is subject to interferences from the I/O subsystem. Similarly, in [30], a comparison between a standard Linux kernel scheduler and a peripheral hardware scheduler connected through a PCI bus is presented, which highlights the potential of a dedicated scheduling circuit for systems composed of interconnected CPU boards.

Some works have investigated the benefits of hardware schedulers in symmetric multiprocessor platforms like the one considered in this work. Specifically, in [31] a many-core architecture is proposed which includes a hardware scheduler to dispatch workload across 256 RISC cores. The results derived from a real implementation underline the benefits for general purpose computing, in terms of reduction of overhead and load balance, coming from a specialized scheduler. However, little investigation is carried out about the benefits of such an architecture for real-time systems. Conversely, [32] provides an accurate description on how a Pfair [5], [33] scheduling algorithm for multiprocessor can be implemented in hardware. The work is particularly relevant for real-time systems since a class of algorithms derived from Pfair is known to be optimal on multiprocessor. Even though a detailed hardware complexity evaluation is provided, this work does not aim at providing an overall system perspective on hardware schedulers for real-time systems and their broader impact.

In the HPC domain some studies [34], [35] have demonstrated how scheduling duties can be offloaded to a specific processor, allocating almost full resources for computation of application workload on the remaining units. Although this approach can be beneficial in application domains where hardware resources are over-provisioned and power efficiency is not a strict constraint, it has a limited applicability to real-time embedded systems. First because if an entire general purpose processor can be dedicated to scheduling, the need for an optimal scheduling algorithm is not justified. Second, because such assignment would determine a waste of all those processor resources (floating point units, cache levels and so on) that are not frequently used for scheduling computation. Third because the performance of a specialized scheduler would be comparable to those of a general purpose processor, requiring lower voltage, frequency and thus achieving higher power efficiency.

Our work is set apart from the mentioned literature because, to the best of our knowledge, we bridge the gap between theoretical studies on complex unrestricted-priority scheduling algorithms and massive multiprocessor hybrid systems that can easily integrate specialized circuits to optimize workload distribution and resource utilization. As such, we investigate and evaluate the possibility of using "active monitoring" to implement in hardware a class of scheduling algorithms that: (A) feature near-optimal or optimal performance on multiprocessor platforms; (B) become easy to implement when relying on active monitoring; (C) ultimately offload the burden of computing scheduling from cores allocated to application workload. Finally, we propose as a case study about the hardware implementation of a Least Laxity First (LLF) scheduler, normally considered as one of the algorithms with highest overhead.

IV. Proposed Architecture

In this section, we provide an overview of the proposed architecture, detailing how it can allow using complex real-time scheduling algorithms to improve resource utilization on multiprocessor platforms.

A. Overall Structure

The structure of the proposed architecture is shown in Figure 1. As can be seen from the figure, the architecture is essential yet general in its components and allows a straightforward implementation of nearly any scheduling algorithm that involves complex priority calculations and frequent queue reordering operations. We propose an architecture that is suitable for multi-core platforms and that can be adapted to many-core architectures with a limited number of changes to that portion of the communication interface that is responsible for delivering scheduling decisions.

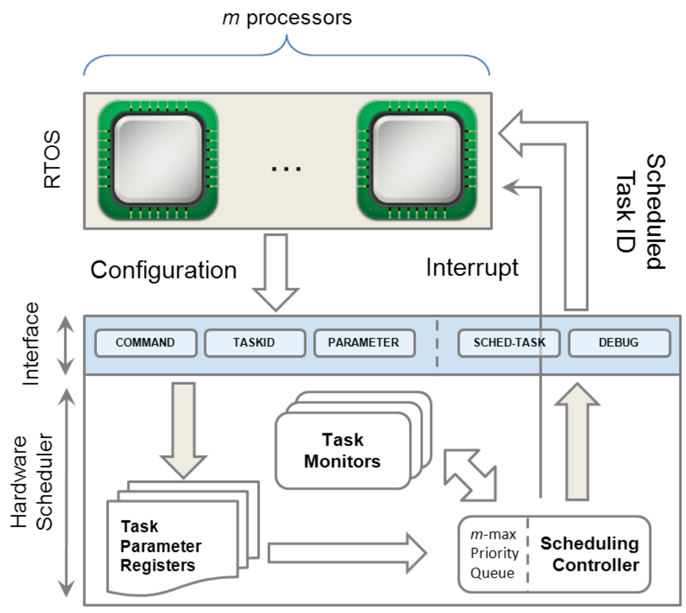

Fig. 1. Overview of the proposed architecture.

As can be seen in the figure, the scheduler is interfaced with a bulk of general purpose processors on which the application workload is dispatched. The pool of general purpose processors runs a stripped version of a RTOS which completely relies on the hardware module to schedule tasks. The RTOS and thus the processors communicate with the hardware module through an interface composed of global and banked registers. On the other end, the hardware scheduler stores task parameters and configuration options in internal registers. These internal registers are used by the *scheduling controller* to compute the scheduling algorithm, as well as to keep its internal state. In addition, the hardware scheduler provides a number of *active task monitors*: one per each task that can be supported.

B. Hardware Scheduler Architecture

We propose an architecture for hardware schedulers that use the idea of "active monitoring" to achieve a fine granularity on the way task priorities are calculated and updated. Specifically, what makes active monitoring a powerful technique is that it allows us, under the constraints of a realistic hardware, to bring the scheduling model closer to the idea of fluid scheduling. In fact, our architecture continuously tracks the evolution of task parameters as tasks execute on processors. As we show in Section V, the achievable granularity is proportional to the achieved synthesis frequency.

1) Communication Interface: The communication interface represents a bidirectional gateway for data flowing from and to the processors that carry either configuration parameters or pieces of scheduling information. The interface is divided into three main parts.

The first part comprises a set of global memory mapped registers that are seen coherently from each processor. This set of registers is used for system-wide configuration purposes: at system start-up or at task creation, the host RTOS can program the hardware scheduler with the parameters of the considered taskset[1]. In our architecture design, the number of supported tasks represents an adjustable parameter that will assume a fixed value in the final hardware synthesis. As such, the configuration parameters of the tasks are sent serially to the scheduler using only a pair of interface registers. A specific command is sent to validate the transmitted set of parameters. Since the taskset configuration is only performed at bootstrap time, having a serialized interface does not represent a scalability issue. Instead, the set of registers described below are used for potentially concurrent task handling commands.

The second part consist of the banked register: all the cores access them in the same range of addresses, but each of them sees a different copy of the mentioned registers. This structure has the advantage of allowing the RTOS code to be simpler and more generic. The set of banked registers is used to carry two main pieces of information: first, the scheduler puts in each banked register the task ID to be scheduled on the targeted processor. Second, a processor can signal the termination of a task to the scheduler using its own banked register.

The interrupt lines that wire the hardware scheduler to the processors represent the third portion of the interface. As previously mentioned, each processor keeps computing application workload until the scheduler module instructs the processor to perform a context-switch to a different task. As soon as the scheduler, according to its policy, determines that a different task needs to be executed on a given processor, it places in the corresponding banked register the task ID of the new task and asserts the interrupt line for the targeted

[1]The parameters sent to the scheduler should be inclusive of all the pieces of information needed to schedule the taskset and depend on the considered task model. For instance, period, WCET and deadline would be sufficient for sporadic tasks

processor. It is responsibility of the RTOS to correctly handle the interrupt and perform the context-switch.

2) Active Task Monitors: The ability of the proposed scheduler to perform active monitoring on the scheduled real-time tasks is offered thanks to the addition of a series of active task monitors. A hardware scheduler which follows the proposed architecture internally implements a number of task monitors that is the same as the number of supported tasks. In their simplest formulation, active monitors are counter registers whose status can be dynamically configured by the scheduling controller.

Each task monitor is updated at the boundary of an internal clock cycle. For each scheduling monitor, three aspects can be dynamically configured: (A) the status of the monitor (counting or stopped); (B) the counting step, as the difference between two consecutive values of the monitor; (C) the counting direction, i.e. if the counter is being incremented or decremented. The scheduling controller, thus, has the responsibility to regulate the behavior of task monitors so that the contained value is reflective of the task priority. For example, as we show in Section V, to implement a LLF scheduler, it is enough that a task monitor is configured to be monotonically decreasing for active, non-scheduled tasks and stopped for active and currently scheduled tasks. The task monitors are logically independent hardware modules, and, as such, their update can progress in parallel at the clock boundary.

3) Scheduling Controller: The scheduling controller is the main block of the module which implements the actual logic of the desired scheduling algorithm. In its simplest formulation, it has three main responsibilities. First, it is responsible for pre-processing and storing the configuration parameters passed through the global registers. This is done to initialize the scheduling mechanism. Second, it directly controls the behavior of the task monitors configuring their step, status and direction at each scheduling decision. Third, it is responsible for selecting the m top-priority tasks to be scheduled on the corresponding m processors and communicating the scheduling decisions through the previously described interface. The selection of the top-priority tasks can be performed in a range of different ways.

For a limited number of tasks and processors, simple combinatorial logic can be set in place to query the status of the task monitors and find those tasks with the lowest (or highest) value of the monitored quantity. This is the approach followed in the proposed case study in Section V. However, as the number of tasks and processors to be supported increases, a more efficient hardware priority queue can be implemented in which each item is a task and whose ordering is kept consistent as the update of task monitors progresses. Similar queues have been largely studied and implemented for network applications [36], [37] and can scale with logarithmic factor with respect to the number of contained items.

Although we propose a structure for a hardware scheduler that is rather generic, some optimizations can be placed to reduce context-switches and migrations on the RTOS side. The first obvious optimization is ensuring that a scheduling interrupt is sent to a target processor only in the case in which the new task to be scheduled is different from the one selected at the previous decision. Moreover, it may happen that the pool of tasks selected to be scheduled at a given time includes some tasks that were already been selected at the previous scheduling decision. In this case, under the assumption that all the processors are equivalent, it is important that said tasks continue executing on the currently assigned processor. Once again, for a limited number of processors and tasks, this optimization can be implemented using simple combinatorial logic, but a dedicated *mix & match* module can be added to the scheduler to perform dispatching decisions on top of scheduling decisions.

4) Overhead and Scheduling Granularity: The overhead of a traditional operating system corresponds to the amount of processor time spent to update task priority queues, perform context-switches, update internal structures and so on. Thus, if we consider an OS using a periodic interrupt with a period of T_{intr}, with a handling time of σ, its overhead can be computed as a utilization factor $U_t = \frac{\sigma}{T_{intr}}$. Thus, in a system with n tasks, the overall overhead utilization factor will be [38]:

$$U_{ov} = \delta \sum_{i=1}^{n} \frac{N_i}{p_i} + U_t \tag{1}$$

Where δ is the cost of a context switch and N_i is the maximum number of context switch that a given task with period p_i can be subject to. The value of N_i strictly depends on the considered scheduling algorithm.

The presence of a dedicated hardware scheduler eliminates the need of a periodic interrupt for scheduling purposes. In fact, the hardware scheduler forwards interrupts to the main processor(s) only when a new scheduling decision needs to be taken. Thus, in our architecture, the amount of overhead only depends on the number of context switches performed by the considered scheduling algorithm, leading to an improved overhead factor of:

$$U'_{ov} = \delta \sum_{i=1}^{n} \frac{N_i}{p_i} \tag{2}$$

In a system that adheres to a fluid scheduling model, the minimum distance between two successive decisions tends to zero. However, this cannot be achieved in real systems for two main reasons: (A) events can only occur at the boundary of a clock cycle; (B) allowing scheduling decisions (and thus context-switches) with an arbitrarily low inter-arrival time, would result in an unacceptable overhead. We call this parameter the *scheduling granularity*. It is clear that, for real implementations, a trade-off needs to be found between scheduling granularity and resulting overhead, which in turns depends on the selected algorithm. In the next section, we show how a LLF scheduler can be implemented with a high granularity and a fixed 1% overhead.

5) Advantages: The advantages of the proposed architecture range from scalability benefits to flexibility and are summarized below.

The first and most important benefit is that a hardware scheduler that follows the proposed architecture makes practical to use complex scheduling algorithms that would introduce an undesirable overhead if implemented in software at an OS-level. Such algorithms often exhibit improved resource utilization with respect to simpler or more restricted ones. The set of implementation options include algorithms from the less restrictive class mentioned in [4], which features tasks with dynamic unrestricted priorities. In Section V we detail how a LLF scheduler can be implemented and provide the basic ideas on how EDZL [39], MLLF [40] and EDF could be implemented as well using the proposed architecture.

The resulting hardware complexity, moreover, can scale well with the number of cores and supported tasks. In fact, the number of required internal registers grows linearly with the number of cores. Similarly, the number of required task monitors linearly increases with the number of supported tasks. Similarly, the number of required interrupt lines scale linearly with the number of cores. Finally, as previously mentioned, hardware priority queues can allow implementations where the number of required hardware resources scale logarithmically with the number of entries [36].

Furthermore, the proposed architecture features all the benefits of providing a real-time system, as well as a general purpose system with a dedicated scheduler module. Besides offloading the application processors from expensive computation of scheduling and increasing the power efficiency of the system, other benefits include: simplifying the design of OSes; increasing their robustness; as well as removing the dependency between the overhead of context-switches and the number of handled tasks; and increasing the power efficiency through specialization.

6) Current Limitations: Our current implementation presents some limitations that can be overcome by extending the set of parameters exchanged through the interface and/or increasing the complexity of the scheduling controller.

First, in this work we focus on real-time tasks that obey the periodic model. Periods can be interpreted as minimum inter-arrival times and explicit deadlines can be added to the list of parameters to implement a scheduler that adheres to the sporadic task model. Similar changes can be made to implement algorithms for more expressive task models. Furthermore, we plan to extend the proposed architecture, as a part of our future work, showing how a shared resource model and different types of servers for aperiodic tasks can be supported by the architecture.

Finally, additional extensions to the set of considered task parameters can allow deploying hardware schedulers for mixed criticality systems and non-real-time systems. For example, direct control can be given to either a user or a peripheral in the system to selectively and dynamically prioritize a subset of tasks.

V. Case Study: Bare Metal OS + LLF

This section presents a case study to demonstrate how the proposed architecture can be instantiated to realize hardware real-time schedulers that implement complex algorithms. Specifically, we focus on the implementation of a Least Laxity First (LLF) [41] scheduler.

A. Why LLF?

LLF scheduling algorithm, which assigns a higher priority to a task with smaller laxity (slack time) is an optimal scheduling algorithm on single processor platform. Unfortunately, it has been proven to be non-optimal [18] on multiprocessor systems. In an extensive study [42], a schedulability test for LLF has been developed. As summarized in Figure 2, the study highlighted how, on multiprocessor, LLF (and other variants based on laxity calculation) can (A) achieve similar performance to Pfair, which is known to be optimal, and (B) largely outperform both Pfair and global EDF on instances of sporadic taskset with density λ^2 greater than the number of processors m.

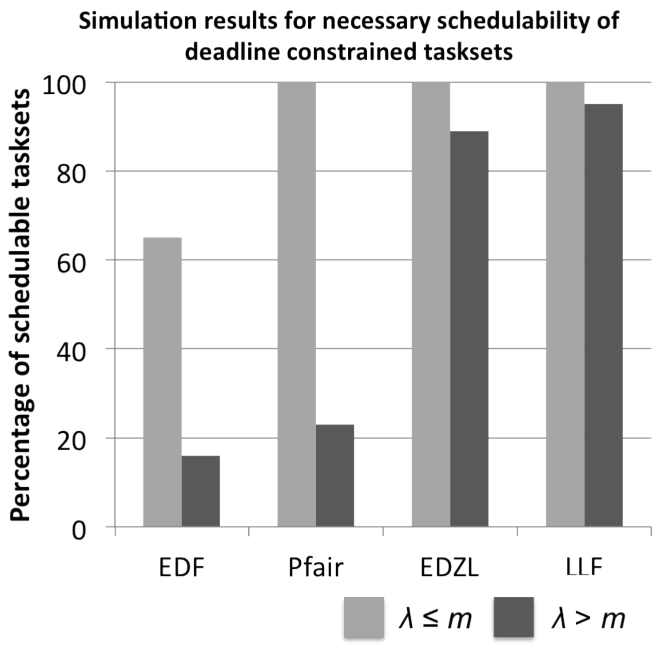

Fig. 2. Simulation results for EDF, Pfair, EDZL and LLF over 100,000 random tasksets on a four-processor platform, as presented in [42].

Unfortunately, however, in its theoretical formulation, a laxity tie determines that an infinite number of context-switches is produced by the algorithm. Moreover, since LLF adheres to a fluid scheduling model in which the priority of a job can change in an unrestricted manner throughout its execution, it has been traditionally considered as unpractical for real applications.

By choosing LLF for our case study, we demonstrate that it is possible to build realistic systems that employ a class of scheduling algorithms that have been reputed unpractical from an implementation point of view, despite their promising theoretical properties. Specifically, we show that the proposed architecture is able to: (A) implement algorithms that follow a fluid scheduling model at a fine granularity by using active monitoring; (B) engineer the implemented scheduler to have a fixed and customizable overhead.

B. Platform for Proposed Architecture

In this case study, we use a hybrid platform to benefit from a multi-core embedded processor while at the same time being able to implement a dedicated hardware LLF scheduler. Specifically, we have used a ZedBoard development board based on the Xilinx Zynq Z-7020 SoC which features a dual-core ARM Cortex-A9 MPCore operating at a frequency of 667 MHz. The application cores are coupled with an Artix 7 FPGA featuring 85k logic cells and including 140 blocks of RAM of 36 Kb each. The block of programmable

[2]In a sporadic taskset, the density is calculated as $\lambda = \sum_i \frac{e_i}{min(d_i,p_i)}$.

logic can be used to both change the interconnections of the processor subsystem with the I/O peripherals and to implement custom user-defined additional logic. Thus, we deploy our LLF scheduler inside the FPGA block, having a fully operating dual-core embedded processor.

1) Host OS: Linux could be selected as the OS that runs on top of the embedded ARM cores. This would have the great advantage of enabling a fast development of user-space benchmarking applications and full interactivity with a wide range of I/O peripherals. Unfortunately, however, the Linux kernel heavily relies on periodic timer interrupts to perform scheduling, book-keeping of resources and interact with I/O devices. Since our aim is to explore the benefits of a system design that does not interrupt the CPU to perform scheduling, the option of using a Linux OS presents several disadvantages. In particular, it would require a non-trivial effort to modify the Linux kernel so as to exclude the scheduling features while implementing a backward-compatible scheduler in hardware.

For this reason, we have developed an essential bare-metal RTOS, called PicOS, on top of the hardware initialization code provided by Xilinx as part of the board support package (BSP). This bare-metal OS provides basic primitives for the definition of periodic tasks, interaction with devices and debugging, but it does not implement any scheduling primitive. In fact, it completely relies on interactions with the hardware scheduler to schedule/de-schedule tasks.

2) Observed Granularity and Overhead: Once a scheduling algorithm is synthesized (in our case on an FPGA), the obtained synthesis frequency f_{hw} can result relatively close to the clock speed f_{cpu} of the main processors. This indicatively means that scheduling decisions can be forwarded to the processors every $T_{sched} = \frac{f_{cpu}}{f_{hw}}$ main processor clocks. A lower value of T_{sched} implies that the fluid scheduling module is followed with a finer granularity. However, since in a real system the context-switch overhead cannot be null, a low value of T_{sched} would determine an unacceptable overhead in a LLF scheduler. In fact, in case of a laxity tie and $T_{sched} = 1$, the LLF hardware scheduler would trigger a context-switch per each clock cycle of the main processor.

In our implementation, we show that a good granularity for a LLF scheduler can be obtained, while at the same time a constant and limited overhead cost can be paid. Specifically, we enforce that our hardware module can forward scheduling decisions only at the boundary of a *scheduling tick*, defining that a given number of internal clock cycles must occur between two successive tick. In this case, $T_{sched} = \frac{f_{cpu}}{f_{tick}}$. It is easy to see that, given the price δ for a context-switch, the implemented (LLF) scheduler will have an overhead of at most $\frac{\delta}{T_{sched}}$. Thereby, f_{tick} is a parameter that can be adjusted to obtain a arbitrarily low overhead. In our experiments, we have observed that, thanks to the reduced context-switch time in the RTOS, we were able to obtain a 1% overhead for our LLF scheduler when f_{tick} =10 KHz[3].

3) Communication Interface: As mentioned in Section IV, the communication interface is implemented using banked and global registers. In our LLF instance, we have three 32 bit global registers per core which the host RTOS uses to send configuration commands to the LLF scheduler. Two additional 32 bit banked registers are used by the LLF scheduler to send the ID of the task to be scheduled to each core.

4) Active Task Monitors: Each task monitor is responsible for updating the state of each task. To implement LLF, each monitor keeps track of two metrics of the current job, (1) relative time to deadline at time t, i.e. $T\_Deadline = d_i - t$, and (2) the amount of computation left *T_Computation*. The values are kept in two 16 bit registers. Upon arrival of a new job, *T_Deadline* and *T_Computation* are initialized to the *Period* and *WCET* of the task respectively. Moreover, the monitor is configured to decrement *T_Deadline* each tick, while *T_Computation* is decremented for each tick if the task is scheduled. At any time the difference of the two metrics would give us the laxity of the current job. This information is used by the scheduling controller to decide the priority of each task. In case a task has expired its WCET i.e., *T_Computation* equals zero, the *T_Runnable* bit for that task is set to *DONE*, which means that the current job has completed and thus it cannot be scheduled anymore until the next period. Also, if a task completes earlier than its WCET, the RTOS sends a *TASK_DONE* command that sets the *T_Runnable* bit for the corresponding task to *DONE*.

Finally, whenever there is a change in the task ID that needs to be scheduled on a given processor, the interrupt signal is set and the new task ID is communicated to the host.

5) Scheduling Controller: The controller module is responsible for implementing the LLF logic. Specifically, the control on the task monitors is performed according to the criteria mentioned above. In case of a tie between two or more tasks, the logic gives preference to those tasks that were scheduled at the previous decision step. In this case, no interrupt is sent to the host and the cost of unnecessary context switches (as mentioned in Section IV-B3) and interrupt handling is avoided. When this condition is not met, the tie is broken according to the lowest task ID and an interrupt signal is sent to each core in which a context-switch has to be performed.

6) Implementation sketch for EDF, MLLF and EDZL: Since scheduling algorithms with job-level dynamic priorities are a subclass of those with unrestricted dynamic priorities, they can as well be implemented using our architecture. For instance, in order to implement EDF, it is enough to (A) initialize a task monitor with the value of relative deadline at each job arrival and (B) configure task monitors for active tasks to monotonically decrease the value of deadline. The tasks with minimum value of deadline are selected to be scheduled.

MLLF [40] is a variant of LLF which significantly reduces the number of generated context-switches. The main idea is that, when a set of m tasks is scheduled at t, it is possible to defer the next context-switch by $\alpha = d_{min}(t) - l_a(t)$, where: $d_{min}(t)$ is the time-to-deadline of the T_{min} task with the earliest deadline; and $l_a(t)$ is the laxity of the task with the smallest remaining execution time among those with minimum laxity. MLLF can be implemented using the proposed architecture by: (A) controlling the task monitors in the same way as the proposed LLF implementation; (B) calculating the new value of α after each scheduling decision; and (C) deferring the next scheduling decision for α ticks.

EDZL [39] is a variant of EDF that schedules tasks according to EDF until some of the non-scheduled tasks reaches a zero-laxity state. Tasks with zero-laxity are given highest

[3]The f_{tick} for a traditional Linux scheduler is 250 Hz

OS	OS Type	Sched. Overhead (μs)
PicOS	**RTOS, ext. scheduler**	**0.98**
Linux	Standard Linux	2.9
Litmus^{RT}	RT Linux	3.5
Preempt_RT	RT Linux	2.7
C Executive	RTOS	3
Delta OS	RTOS	23
ThreadX	RTOS	2

TABLE I. COMPARISON OF OSES SCHEDULING OVERHEAD.

priority. EDZL can be implemented following the proposed architecture by monitoring the amount of slack for each task as we do for LLF. Next, scheduling decisions select at most $k \leq m$ tasks with laxity equal to zero, and the remaining $m-k$ tasks with the earliest deadline.

VI. METHODOLOGY AND RESULTS

In this section, we describe the methodology followed in the experiments, as well as the results of the performed evaluation.

A. System Instantiation

A performance evaluation as well as an analysis of FPGA resource utilization have been done on a final version of the system that includes only the deployed components. Specifically, the ARM CPUs have been instantiated together with a central interconnect. Moreover, the scheduling interrupt pins of our LLF scheduler module have been routed to the Generic Interrupt Controller (GIC). A basic support for the UART has been included in our RTOS, while the discussed LLF module is the only hardware block configured on the programmable logic block.

B. Scheduling Overhead

We have measured the time required to handle the scheduling interrupt, which includes interacting with the LLF scheduler module and performing a complete context-switch of the preempted/preempting tasks.

The methodology used to perform the measurements involved using the ARM performance counters. Specifically, we have configured the PMCCNTR (clock cycles counter) register to monotonically count the CPU clock cycles with a granularity of a single cycle and to be accessible from user-mode. Next, we have placed two single assembler instructions to read the performance counter into a global variable, at the beginning of the interrupt handling procedure and at the end, respectively. This allows us to have an accurate measurement of the time required to handle a scheduling interrupt with a negligible measurement overhead (4 clock cycles).

The results of our measurements highlight two main aspects. First, that an external scheduling module can be successfully used to offload the complexity of computing the task schedule. In fact, the average number of clock cycles needed to handle a scheduling interrupt is 651 cycles, resulting in a handling time of about $0.98\mu s$. Such handling time is remarkably lower than a classic Linux, a Real-Time Linux[4] [43] and represents a significant improvement over existing RTOSes as well [44]. A complete comparison in reported in Table I.

Second, that since the complexity of the scheduling algorithm is hidden inside the hardware module, the resulting scheduling overhead has an excellent determinism. In fact, the observed variation on the scheduling overhead over 100000 samples has returned a minimum of 648 cycles and a maximum of 653 cycles, which, in turn, translates to a time fluctuation of $\pm 0.004\mu s$. This is a highly desirable feature for a RTOS designed for safety-critical applications.

The specially low overhead paid for handling tasks according to the LLF schedule computed by the hardware module implies that, given a target context-switch overhead, our setup is able to sensitively increase the granularity of the scheduler. Specifically, for schedulers implemented in software, typical tick frequencies range from 100-250 Hz, which is the default tick frequency of a Linux kernel compiled for i386 platforms. Our experiments suggest that, while keeping an overall CPU overhead of 1%, PicOS can have a tick which is two orders of magnitude higher. Specifically, we have tested our implementation with a FPGA clock frequency of 48 MHz, with a tick defined as 4800 internal clocks. This leads to a scheduling period of $100\mu s$, which, in turn, determines a scheduling frequency $f_{tick} = 10$ KHz. It follows that a scheduling model that is feasible and much closer to the idea of fluid scheduling can be adopted. Clearly, another problem related to designing a system in compliance with a practically feasible fluid model concerns cache behavior of applications. However, cache resources can be administered in a way that they become insensitive to process migrations and context-switches, for example using lockdown mechanisms [19], [20].

C. Analysis of Hardware Resources

Figure 3 reports the trend of the FPGA resource utilization, in terms of flip-flops and Look-Up Tables (LUTs), for the developed LLF scheduler module. The measurements are reported as a function of the number of tasks supported by the system. This metric is useful to understand the complexity of the proposed scheduling mechanism. As can be noted, the resource usage trend is linear with the number of tasks.

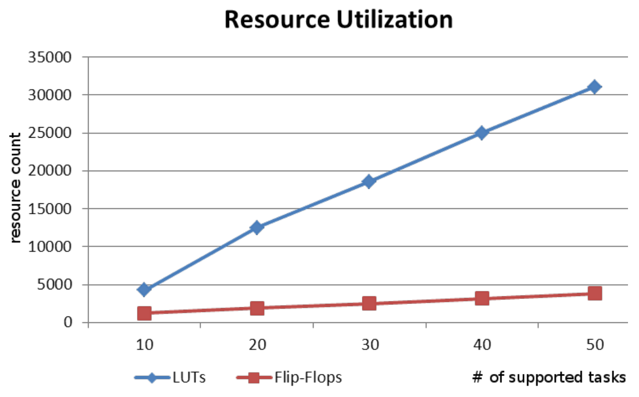

Fig. 3. Resources usage trend of our LLF scheduler as a function of supported tasks.

In this case, we compare our result with the Pfair scheduler in [32], whose gate occupation graph is reported in Figure 4. As can be noted, despite the differences in synthesis technology[5], the order of magnitude of utilized components is the same, which suggests a comparable hardware complexity of the considered schedulers.

[4]Note that the results reported on Linux and RT Linux systems have been performed on high-end machines.

[5]The results in Figure 4 have been obtained using a 90nm ASIC library.

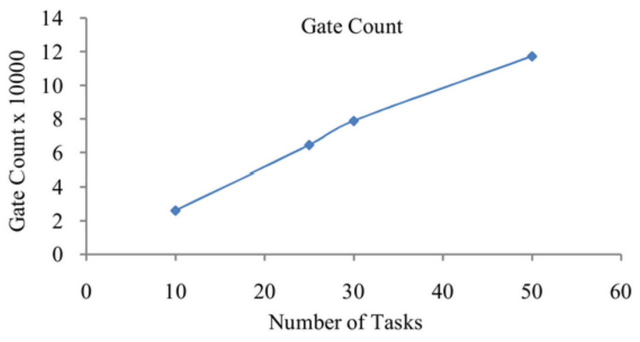

Fig. 4. Resource usage trend of Pfair ASIC scheduler presented in [32].

D. Maximum Frequency

We have evaluated how the internal maximum frequency of the synthesized LLF scheduler varies with the number of supported tasks. The synthesis objective has been set as speed optimization. The results are shown in Figure 5 and highlight that the maximum frequency decreases with a law that has a sharp decrease for a number of supported tasks between 10 and 20, while it follows a linear trend from 20 tasks ahead. It is worth to note, however, that, despite the undesirable decrease in the observed maximum frequency, a real-size taskset of 50 tasks leads to a frequency of about 48 MHz, which is still sensitively higher than what is required by our PicOS. In fact, to keep a scheduler tick frequency of 10 KHz, it is enough to raise a tick interrupt every 4800 FPGA clock cycles, i.e. f_{tick} =10 KHz.

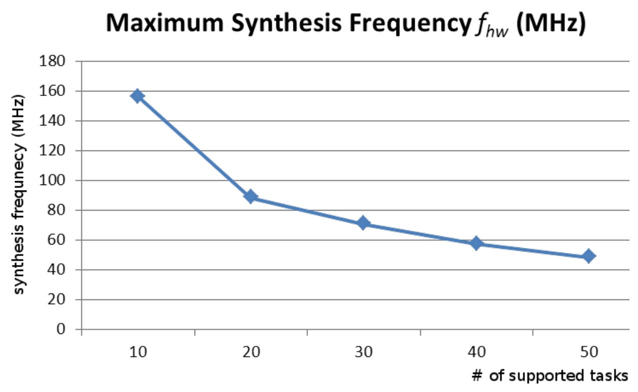

Fig. 5. Maximum Frequency (FMax) trend of our LLF scheduler as a function of supported tasks.

E. Power Usage

Finally, we have investigated the power consumption of our current FPGA implementation of the proposed scheduler as a function of the number of tasks supported by the system. This is because, in the current implementation, the parameters of all the tasks are stored inside the FPGA module for schedule computation. This impacts FPGA power usage in two ways: first, additional flip-flops are required as the number of supported tasks grows; second, the critical path of the scheduler involves updating slack values and finding two minimum slack values, which is proportional to the number of tasks running in the system. Figure 6 depicts the trend of the power usage as the number of tasks supported by the system grows.

As can be noted, the trend for the power consumption of

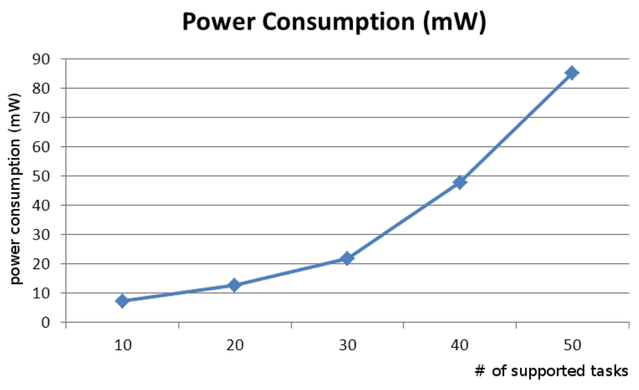

Fig. 6. Power usage trend of the hardware LLF scheduler as a function of number of supported tasks.

the LLF scheduling logic follows a growth law that - as already observed for the frequency - is piecewise linear. Specifically, a sharp increase can be observed between 30 and 40 tasks, while a linear trend is exhibited from 40 tasks ahead. This result is not surprising, since the synthesis objective is speed optimization. This means that, in order to achieve a higher frequency, the synthesis tools apply sub-optimal strategies from a power efficiency point of view.

We envision that future implementations will involve the creation of dedicated ASIC circuits. However, the translation from FPGA structure to ASIC logic gates is not straightforward due to the profound differences in synthesis technology [45]. Thereby, as a part of our future work, we plan to implement and evaluate a class of laxity based scheduling algorithms using a 45nm ASIC library.

VII. Conclusion and Future Work

In this work, we have proposed an architecture that revolves around the idea of “active monitoring” to continuously track the evolution of tasks parameter according to the logic implemented scheduling algorithm. Such an architecture is general enough to allow a practical implementation of a class of algorithms that use unrestricted dynamic priorities. Since these algorithms are more powerful than those featuring job-level dynamic priorities, they are able to exploit processor resources more efficiently, especially as the number of processors grows.

We demonstrate that: (A) it is possible to offload the burden of scheduling to a dedicated hardware module; (B) doing so can mitigate the high complexity factor that needs to be paid to deploy scheduling policies with unrestricted priorities; and (C) the resulting low scheduling overhead helps reducing the gap between implementable scheduling techniques and the ideal fluid scheduling model.

For these reason, we have provided an example of how a LLF scheduler with fixed overhead and high granularity can be instantiated using the proposed architecture. We also provide the intuition on how algorithms belonging to the same class (EDZL and MLLF) can be implemented as well. Finally, we have performed an evaluation on a real multi-core hardware platform and RTOS.

As a part of our future work, we plan to extend the proposed architecture to include the support for shared resources and non-interruptible sections. Moreover, we want to investigate the benefits of performing the synthesis of hardware schedulers using ASIC technologies, as well as to further generalize the proposed architecture by developing a programmable scheduling processor.

Acknowledgment

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1302563, and CNS-1219064. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] B. Andersson and J. Jonsson. Preemptive multiprocessor scheduling anomalies. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, page 8, April 2002.

[2] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 193–202, 2001.

[3] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 205–217, New York, NY, USA, 1972. ACM.

[4] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004.

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

[6] A. Srinivasan. *Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors*. PhD thesis, 2003. AAI3112080.

[7] B. D. Bui, R. Pellizzoni, and M. Caccamo. Real-time scheduling of concurrent transactions in multidomain ring buses. *IEEE Trans. Comput.*, 61(9):1311–1324, September 2012.

[8] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 3–13, July 2010.

[9] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[10] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

[11] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190, Dec 1990.

[12] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Trans. Softw. Eng.*, 23(10):635–645, October 1997.

[13] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 63–72, Dec 2012.

[14] S. K. Baruah. Feasibility analysis of recurring branching tasks. In *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, pages 138–145, Jun 1998.

[15] M. Stigge, P. Ekberg, N. Guan, and Wang Yi. The digraph real-time task model. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 71–80, April 2011.

[16] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[17] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[18] J. Y. T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1-4):209–219, 1989.

[19] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 114–123, 2002.

[20] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems. *J. Syst. Archit.*, 57(7):695–706, August 2011.

[21] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54, 2013.

[22] S. W. Moon, K. G. Shin, and J. Rexford. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Trans. Comput.*, 49(11):1215–1227, November 2000.

[23] B. K. Kim and K. G. Shin. Scalable hardware earliest-deadline-first scheduler for ATM switching networks. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 210–, Washington, DC, USA, 1997. IEEE Computer Society.

[24] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time operating systems. *SIGOPS Oper. Syst. Rev.*, 23(3):54–71, July 1989.

[25] J. Stankovic, D. Niehaus, and K. Ramamritham. SpringNet: A scalable architecture for high performance, predictable, and distributed real-time computing. In *Predictable, Distributed, Real-Time Computing, Univ. of Massachusetts, Technical Report*, pages 91–74, 1993.

[26] W. Burleson, J. Ko, D. Niehaus, K. Ramamritham, J.A. Stankovic, G. Wallace, and C. Weems. The spring scheduling co-processor: a scheduling accelerator. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD '93. Proceedings., 1993 IEEE International Conference on*, pages 140–144, Oct 1993.

[27] L. Lindh. FASTHARD - a fast time deterministic hardware based real-time kernel. In *Real-Time Systems, 1992. Proceedings., Fourth Euromicro workshop on*, pages 21–25, June 1992.

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

[29] J. Furuns. Benchmarking of a real-time system that utilises a booster. In Hamid R. Arabnia, editor, *PDPTA*. CSREA Press, 2000.

[30] T. Klevin and L. Lindh. Scalable architecture for real-time applications and use of bus-monitoring. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 208–211, 1999.

[31] I. Avron and R. Ginosar. Performance of a hardware scheduler for many-core architecture. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 151–160, June 2012.

[32] N. Gupta, S. K. Mandal, J. Malave, A. Mandal, and R. N. Mahapatra. A hardware scheduler for real time multiprocessor system on chip. In *VLSI Design, 2010. VLSID '10. 23rd International Conference on*, pages 264–269, 2010.

[33] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 294–303, 1999.

[34] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Trans. Parallel Distrib. Syst.*, 16(11):1066–1077, November 2005.

[35] Jinhui Qin and Michael A. Bauer. Job co-allocation strategies for multiple high performance computing clusters. *Cluster Computing*, 12(3):323–340, September 2009.

[36] S. W. Moon, K. G. Shin, and J. Rexford. Scalable hardware priority queue architectures for high-speed packet switches. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 203–212, Jun 1997.

[37] S. W. Moore and B. T. Graham. Tagged up/down sorter - a hardware priority queue. *The Computer Journal*, 38:695–703, 1995.

[38] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[39] S. K. Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, pages 607–611 vol.2, Aug 1994.

[40] S. H. Oh and S. M. Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 31–36, Oct 1998.

[41] J. Y. T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1-4):209–219, 1989.

[42] J. Lee, A. Easwaran, and I. Shin. Laxity dynamics and LLF schedulability analysis on multiprocessor platforms. *Real-Time Systems*, 48(6):716–749, 2012.

[43] F. Cerqueira and B. Brandenburg. A comparison of scheduling latency in linux, PREEMPT RT, and LITMUS RT. *OSPERT 2013*, page 20, 2013.

[44] S. Baskiyar. A survey on real-time operating systems. *Proceedings on Networks, Parallel and Distributed Processing and Applications*, 2002.

[45] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM.