Saving Energy by Means of Dynamic Load Management in Embedded Multicore Systems

Matthias Becker*, Adriaan Schmidt†, Martin Orehek‡, Thomas Nolte*
*MRTC / Mälardalen University, Västerås, Sweden
{matthias.becker, thomas.nolte}@mdh.se
†Fraunhofer Institute for Embedded Systems and Communication Technologies ESK, Munich, Germany
adriaan.schmidt@esk.fraunhofer.de
‡University of Applied Sciences Munich, Munich, Germany
martin.orehek@hm.edu

***Abstract*—Load balancing is widely used to optimize response times and throughput of software systems. When considering embedded systems, however, additional optimization goals like energy consumption become relevant. In this paper, we explore the use of load balancing in embedded multicore applications. We present extensions to three prominent load balancing schemes, enabling them to dynamically scale the number of active cores. We integrated the algorithms in a proprietary operating system targeting multicore embedded systems. Our evaluation, which is based on a telecommunication (VoIP) scenario, shows that a significant reduction in energy consumption is possible.**

I. INTRODUCTION

One important problem in the context of multicore systems is the scheduling of computational tasks onto the available cores [1]. In contrast to single core systems, where the only decision is *when* to execute which software, the scheduling problem is now two-dimensional: in addition to the time domain, there is a space domain. The decision becomes *when* and *where*, that is on which core, software is executed. The goal is to optimally utilize the available processing power, maximize throughput, and minimize system response times. Modern operating systems with Symmetric Multi Processing (SMP) architectures address both dimensions simultaneously, performing dynamic scheduling and load balancing. Tasks are automatically assigned to cores, and are migrated when the utilization of the cores changes. This way, some of the complexity of the platform is hidden from the software developer.

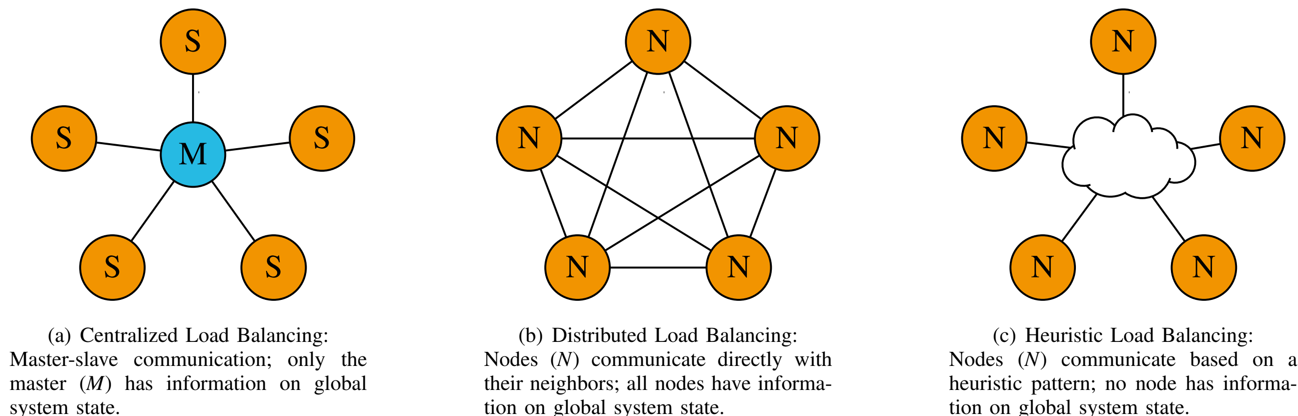
While this approach is successful in desktop and server applications, its use in embedded systems is limited. Applications with real-time requirements often cannot tolerate the non-determinism associated with SMP architectures. The sources of non-determinism include locks on shared data structures within the kernel [2], as well as the dynamic scheduling itself. In addition, SMP is not suited for some of the hardware platforms encountered in embedded systems. The shared memory programming model commonly used in the implementation of SMP systems requires a coherent cache infrastructure. In our work, we target a specific embedded platform that omits a cache coherence fabric to save chip area and costs, and can therefore not execute SMP operating systems. Heterogeneous hardware platforms form another class of systems that do not easily support SMP architectures.

Instead, many embedded systems use Asymmetric Multi Processing (AMP), where cores operate independently, either running different instances of the same operating system, or completely different environments altogether. This splits the two-dimensional scheduling problem, with the scheduler on each core only handling the time domain by performing local scheduling. In strict AMP systems, placement of tasks onto cores is performed at design time and does not change at runtime. While this approach can give the developer better control over the system behavior and permits analyses to verify that timing requirements are met, it also creates more static systems than SMP.

In this paper, we explore the benefits of load management in embedded multicore systems. We target soft real-time systems, i.e., systems that are optimized for throughput, but while there are real-time requirements, violations of deadlines are permitted to a certain degree. Examples of such systems are devices that process media (audio/video) streams. The concrete use-case on which our evaluation is based is a system that performs encoding and decoding of multiple Voice-over-IP (VoIP) streams in real-time.

To achieve management of load, we extend a single-core operating system with algorithms for inter-core communication, task migration and load balancing. To maintain independence of cores, our approach is designed to solely communicate via asynchronous message passing; no shared data structures are used. Our extensions introduce no additional synchronization, which means that all advantages of the AMP architecture remain. The resulting architecture can be regarded as a multikernel operating system [3]. These design decisions make it possible to use existing load balancing schemes from the area of distributed systems as the starting point of our work [4].

The common optimization goal in load balancing is an even distribution of computational load to achieve minimal response times and highest possible throughput. In many embedded systems, this goal can be relaxed: often a system has to meet certain deadlines, but there is no gain in finishing execution ahead of time. So the response time does not need to be "as fast as possible", it just needs to be "fast enough", in the context of the system in question. On the other hand, there may exist different optimization goals in embedded systems, e.g., low energy consumption to reduce cost and heat dissipation. With this in mind, we extend existing load balancing algorithms to make them "energy aware". This means that during system



(a) Centralized Load Balancing: Master-slave communication; only the master (M) has information on global system state.

(b) Distributed Load Balancing: Nodes (N) communicate directly with their neighbors; all nodes have information on global system state.

(c) Heuristic Load Balancing: Nodes (N) communicate based on a heuristic pattern; no node has information on global system state.

Fig. 1. Communication structure of different load balancing strategies.

operation, the number of active cores is scaled dynamically according to system load, thus reducing power consumption of the processor.

We select three different load balancing schemes: one centralized, one distributed, and one heuristic approach. Their different communication structures are shown in Fig. 1. We implement each of them, first based on the original concept, and then with our extensions for energy awareness. We evaluate the implementations in an embedded multicore system, with different types of load, representative of applications in the area of telecommunication. To quantify the difference between the algorithms, we measure execution times and deadline misses of our load applications as well as energy consumption of the processor cores. Finally, we provide an estimate of the possible energy savings when using energy aware load balancing in a real-world system.

The system architecture we propose, i.e., asymmetric multiprocessing extended with dynamic, energy aware load balancing, offers several benefits. Being based on AMP concepts, the approach maintains its suitability for embedded and real-time applications. At the same time the dynamic load balancing enables high throughput, with no need for a static allocation of tasks to cores. In addition, the measures to scale the number of active cores according to system load can significantly reduce the energy consumption of the embedded system.

The rest of this paper is structured as follows. Section II summarizes the related work, including the existing load balancing schemes we use as basis for our own implementations. Section III describes in detail the selected load balancing approaches. In Section IV we present our extensions that scale the number of active cores according to system load, and thus make the algorithms energy aware. Section V describes our evaluation methodology and results. Finally, Section VI concludes the paper.

II. Related Work

Much research has been conducted in the field of power aware scheduling. Most proposed approaches apply Dynamic Voltage and Frequency Scaling (DVFS) to adapt the processor speed to the workload. This is done either at design time or during runtime. The power aware scheduling by Aydin et al. [5] is one example for a single core DVFS scheduler. They use an offline based static worst case solution to set an initial processor speed which is then modified during runtime based on the actual workload and on speculative adaption based on future executions. Muthukaruppan et al. [6] propose a hierarchical power management framework. They focus on asymmetric multicore platforms and an evaluation is performed on real hardware. Different control loops are used to assign tasks to cores and frequencies to clusters. Annamalai, et al. [7] propose a dynamic thread scheduling for AMPs. They use hardware counters as feedback while keeping the OS unaware of the concrete microarchitecture used. If the power consumption of one thread changes during execution it is detected by the monitor, and thread swapping is applied. For our case, where we assume identical cores, such an AMP based approach could be considered if we set different frequencies for each core like done in [8]. However, changing the operation frequency at runtime comes with drawbacks. A different frequency leads to a different worst case execution time, affecting latency and introducing jitter. Considering these effects in the algorithms can lead to more complexity and larger overhead. Therefore we restrict our approach to using only two states *active* and *idle*. Switching between the two states is done based on thresholds. Pinheiro et al. [9] use load unbalancing to scale the number of nodes in a cluster of workstations or PCs. If the system load increases, additional cluster nodes are turned on, while at times of low load, cluster nodes are turned off. Jeon et al. [10] use a similar principle on an embedded multicore processor. They use rate-monotonic priority assignment and exploit the utilization threshold to concentrate the periodic tasks on as few cores as possible. For tasks with unknown duration, shortest-queue scheduling is applied, and thus they are distributed to all available cores. Liu and Yang [11] propose a two level hierarchical scheduling scheme for hard real-time tasks on a homogeneous multicore embedded processor. Similar to [10], they consider the two task types periodic and aperiodic.

With increasing number of cores on one processor, there is need for operating system support to fully utilize the processors' potential. In contrast to traditional operating system concepts, like the monolithic kernel of Linux, distributed operating system designs gain in popularity. Monolithic kernel operating systems require locking of kernel data structures. With rising core numbers, the wait time for those locks increases, and they can become bottlenecks [2]. Baumann et

al. [3] propose a multikernel operating system. They show how ideas from distributed systems and networks can be used in such a design to reduce problems of traditional designs. Wetzlaff and Agarwal [12] use a similar approach for their factored operating system (*fos*). The main objective of *fos* is the scalability to future manycore systems with hundreds of cores. *fos* uses message passing between the cores with Internet-like communication structures. Holmbacka et al. [13] extend FreeRTOS with a task migration mechanism, allowing it to run individual instances of the operating system on the cores of an embedded multicore processor. The cores use message passing to communicate, and thus implement a multikernel architecture. Like argued in [4], well known algorithms from distributed systems can be transfered to such operating systems.

Willebeek-LeMair and Reeves [14] compare five different dynamic load balancing algorithms and illustrate the trade-off between the accuracy of information the different nodes have on the overall system state and the overhead introduced by the algorithm. Their simulations indicate that, from the algorithms they compared, the Receiver Initiated Diffusion approach performs best for a large variety of applications and a large range of system topologies. Lan and Yu [15] propose a centralized load balancing algorithm using two thresholds to split the load levels into *LOW*, *NORMAL* and *HIGH*. Their algorithm reduces the average task response time compared to previously proposed centralized load balancing algorithms. Additionally, Lan and Yu explore different invocation strategies, comparing instantaneous load balancing, that reacts immediately to load changes, to periodically invoked load balancing. Their work shows that instantaneous load balancing reduces average task response times at the cost of greater message overhead compared to periodic load balancing. More recently, a non-blocking work-stealing algorithm has been proposed by Arora et al. [16]. This approach becomes increasingly popular especially in industry since it shows good results for a range of applications [17]–[19].

III. Load Balancing in Embedded Multicore Systems

Load Balancing is the process of distributing workload to a number of resources, in our case the cores of a multicore processor. The main goal of load balancing in distributed systems is to achieve an equal workload distribution over all cores to obtain minimal response times and maximize throughput. The load balancing process can be divided into four steps [14]:

1) **Processor load evaluation:** To measure the workloads of the cores a common metric is defined, and the load is evaluated according to this metric.
2) **Load balancing profitability determination:** During this phase the decision is made whether load balancing is profitable at the current time.
3) **Task migration strategy:** If load balancing is found profitable, the source and the destination cores are determined, along with the amount of load to exchange.
4) **Task selection strategy:** The source core selects the tasks for effective load balancing and migrates them to the destination core.

Phases two and three are addressed by the actual load balancing algorithm and represent the main focus of this paper.

For the work of this paper, we selected three prominent load balancing algorithms. One of the main criteria in selecting the algorithms is to cover a variety of different strategies to get an overview of their benefits and downsides.

We selected one centralized, one distributed and one heuristic algorithm (Fig. 1). The centralized approach makes all load balancing decisions at a central point in the system, whereas the decentralized approach lacks such a component; instead all elements are equally allowed to make load balancing decisions. The last approach initiates load balancing based on a heuristic principle, which sets it apart from the first two.

During operation, all three algorithms rely on a message passing infrastructure provided by the OS. Incoming messages are handled at each operating system tick, with the period defined as $t_{\text{period, OS}}$. The load balancing algorithms themselves are invoked periodically, at a lower frequency than the operating system tick: $p_{\text{LB}} \cdot t_{\text{period, OS}}$.

Table I lists all messages used by the three algorithms, including the arguments passed as message data. In the descriptions of messages and algorithms, we refer to processor cores as P_i, and to their load as L_i.

A. Centralized Load Balancing Approach

We selected the Central Scheduler Load Balancing (CSLB) algorithm as representative of the centralized load balancing schemes. The CSLB algorithm was proposed by Lan and Yu [15]. It is originally designed for distributed systems, and we adapted it to fit the needs of the embedded multicore platform. Each participating core is a so called local node. A central component, the central scheduler, keeps track of the different load levels of the system and performs load balancing if needed.

When the load level on a local node changes, a Load Update message is sent to the central scheduler. To lower the volume of these messages, the algorithm uses two thresholds, T_{LOW} and T_{HIGH}, to split the load levels of the local nodes into three states: *LOW*, *NORMAL* and *HIGH*. Messages are only sent when the load level crosses one of the thresholds.

The central scheduler is the main component of the algorithm. It consists of two main parts, the load list and the decision maker. Additionally, it has a queue to receive update messages from the local nodes. The load list is implemented as ordered list and sorted by load. Each element of the list represents one local node consisting of its ID and its current load.

The profitability of load balancing is determined by the decision maker, based on information received in the Load Updates from the local nodes. If the decision maker detects an imbalance in load levels it initiates load balancing. This is the case if the load list contains at least one node in state *LOW* and one node in state *HIGH*.

For the third phase, the task migration strategy, the algorithm takes the entry with the smallest load and the entry with the biggest load from the load list. Those entries are called P_{low} and P_{high}. It then attempts to balance the load

TABLE I. MESSAGES USED BY THE LOAD BALANCING ALGORITHMS, INCLUDING THEIR ARGUMENTS

Central Scheduler Load Balancing

New Threshold	Notification of new thresholds, sent by central scheduler
T_{high}	new threshold for HIGH state
T_{low}	new threshold for LOW state
Balance	Load balancing request, sent by central scheduler
L_E	amount of load to migrate
$P_{\text{destination}}$	destination core
Load Update	Notification of change in load level, sent by local nodes
P_i	sending core
L_i	load level of core P_i

Receiver Initiated Diffusion

Load Update	Notification of change in load level
P_i	sending core
L_i	load level of core P_i
Load Request	Request to migrate load
P_i	core requesting the load
L_{request}	amount of load requested
Request Handled	notification that load request was handled
—	(no arguments)

Load Stealing

Steal Request	Request to migrate load
P_{thief}	core requesting the load
L_{thief}	load level of core P_{thief}

between those two nodes. This is done until no more high/low combinations are found in the list. Load balancing algorithms that first balance the lowest with the highest node in the system are called fair as stated by Lan and Yu [15].

To balance the load between P_{low} and P_{high} the excess load, L_E, is computed from their respective load levels, L_{low} and L_{high}:

$$L_E = \frac{L_{\text{high}} + L_{\text{low}}}{2} - L_{\text{low}}$$

The central scheduler then sends a Balance message to P_{high}, requesting it to migrate its excess load to P_{low}.

The central scheduler is also responsible to manage the thresholds T_{low} and T_{high}. These are computed dynamically and broadcast to the local nodes whenever they change. The threshold values are based on the average system load, $L_{\text{AV}} = \frac{1}{N}\sum_{i=0}^{N-1} L_i$, where N is the number of local nodes and L_i is the load of node i. They are calculated as:

$$T_{\text{high}} = \max\{T_{\text{high_min}}, h \cdot L_{\text{AV}}\}$$
$$T_{\text{low}} = \max\{T_{\text{low_min}}, l \cdot L_{\text{AV}}\}$$

The parameters l and h are used to configure the behavior of the algorithm. With the following restrictions: $l \leq 1$ and $h \geq 1$.

B. Distributed Load Balancing Approach

Here we describe a distributed approach to load balancing based on the Receiver Initiated Diffusion (RID) algorithm, originally proposed by Willebeek-LeMair and Reeves [14]. Receiver initiated approaches have the advantage that the load balancing overhead occurs on cores with low load in contrary to sender initiated approaches where the highly loaded cores perform most of the load balancing calculations. RID achieves load balancing through a fully distributed asynchronous approach. The same algorithm runs on each core independently.

In the original algorithm, the target system is viewed as a graph, with cores represented by nodes and their communication channels by edges. Each node balances its load among a set of neighbors, its "domain". As the algorithm originates from distributed systems, the domains are defined by the given hardware topology. We can assume that, on an embedded multicore processor, all cores can directly communicate with each other. Thus, the graph of communication channels is always fully connected, and we can simplify the algorithms accordingly. Luque et al. [20] showed that under these conditions, a distributed diffusion approach reaches the equilibrium after just one load balancing step.

To determine if load balancing is profitable, two steps are taken. First the load of the node has to be less than a user defined threshold T_{low}. If low load is detected, the node computes the average load of its domain as $L_{\text{AV}} = \frac{1}{N}\sum_{i=0}^{N-1} L_i$.

To avoid unnecessary load migrations, a second threshold, T_{diff}, is used by the algorithm. This threshold describes the minimal needed difference between L_{AV} and L_i to proceed with the algorithm.

If $L_{\text{AV}} - L_i \geq T_{\text{diff}}$, the third load balancing phase is started, and a weight h_k is assigned to each neighbor:

$$h_k = \max\{0, L_k - L_{\text{AV}}\}$$

The weights are added up to determine the total surplus H_i of all nodes in the domain of core P_i:

$$H_i = \sum_{k=0}^{N-1} h_k$$

After each neighbor has been assigned a weight portion we can compute the load amount the node P_i requests from those neighbors:

$$\delta_k = (L_{\text{AV}} - L_i)\frac{h_k}{H_i}$$

Because the algorithm operates asynchronously on the different cores, messages can be based on old information. Specifically, cores can request more load than the respective node can spare. To avoid this situation, we introduce a load budget, $L_{\text{budget}} = \frac{L_i}{2}$. In one load balancing cycle, a node does not migrate more than half its load; additional Load Request messages are ignored.

The load update strategy, i.e., the time and frequency of load updates, is one crucial factor in order to obtain good results. This is especially important in distributed load balancing algorithms, because too many updates increase the overhead and may slow down the performance of the system, while too few updates result in many requests based on old information that can not be satisfied. Willebeek-LeMair and Reeves [14] propose a variable update interval that yields a

constant error percentage in the load information but reduces the overhead due to the update messages.

A factor F_{update} is introduced. In combination with the last update value sent, L_{last}, two thresholds are computed:

$$T_{\text{update_low}} = L_{\text{last}} \cdot F_{\text{update}}$$
$$T_{\text{update_high}} = \frac{L_{\text{last}}}{F_{\text{update}}}$$

This leads to update messages on the order of $\log_{F_{\text{update}}}(L_{\text{last}})$ and the error percentage of the load information will be at most $1/F_{\text{update}}$ [14].

The frequency of update messages increases as the load decreases. This means that information is most accurate for nodes with low load, which is where it is most important for the load balancing algorithm.

C. Heuristic Load Balancing Approach

The non-blocking work-stealing algorithm of Arora et al. [16] takes a heuristic approach in load balancing and is the basis for all work-stealing algorithms. Work-stealing is one of the most popular load balancing techniques in both industry and academia [21] due to its simplicity and effectiveness.

Each core acts independently, and work stealing is only initiated once a core detects that it is underloaded. Unlike the previously described algorithms, in work-stealing no core has any information about the load levels on the other cores. If a core detects that it is underloaded, another core is chosen at random, and the core attempts to steal load from this core. The algorithm described by Arora et al. was designed to steal only one element with each steal attempt. However, several authors argue that the algorithms can be improved if more than one element is stolen at a time [17], [22]. The basic load balancing algorithm we implement is similar to the one described by Hendler and Shavit [21]. Their algorithm allows stealing of up to half the load from other cores. They assume a local work queue for each core. Stealing cores access the other cores' queues directly. Thus the case where two parties access one work queue at the same time must be considered. The multikernel architecture prevents such situations. If one core attempts to steal, it has to use the message passing mechanism provided by the operating system, which provides an implicit synchronization of the queue accesses.

The load balancing profitability determination is spread over two cores. One core (the thief) initiates load stealing if its load is lower than a predefined threshold T_{low}. If this is the case the core selects one of its neighbors at random (the victim) and attempts to steal load by sending a Steal Request message. The receiving core now has to decide whether load balancing is profitable, and the steal attempt should succeed, or whether the request should be ignored. Load balancing is considered profitable, if the load of the stealing core is below the load of the victim by at least a margin of T_{diff}: $L_{\text{thief}} \leq L_{\text{victim}} - T_{\text{diff}}$. If this is the case, the victim migrates as much load to the thief as needed to balance those two:

$$L_{\text{E}} = \frac{L_{\text{thief}} + L_{\text{victim}}}{2} - L_{\text{victim}}$$

If the victim core discards the steal attempt, no reply message is sent. The thief detects a discarded steal attempt if the victim core did not send any load until the next load balancing period. In this case the thief selects a new victim core.

In the Load Stealing algorithm, the distinction between the second and third phases of load balancing is not as clear as for the CSLB and RID algorithms. With the selection of a random node for the steal attempt, the migration strategy is already partly decided: the source and destination cores are determined, even before it is known whether load balancing is profitable.

IV. Energy Aware Load Balancing

Load balancing itself does not provide support to adapt the number of active cores to the current computational demand of the system. This is not always optimal for embedded systems because even a small number of tasks is distributed to all cores. Thus all cores are running and use the maximal power.

The basic idea to resolve this problem is the consolidation of work on a subset of cores as long as the load on those cores is tolerable. As the introduced load balancing algorithms differ in their information and communication patterns, different solutions were applied, as described in this section. The additional messages that are sent to manage the number of active cores are listed in Table II.

A. Central Scheduler Load Balancing

We used an approach similar to the load unbalancing strategies proposed by Jeon et al. [10] and Pinheiro et al. [9]. A new threshold T_E is introduced. T_E is assumed to be the level of load tolerable on one core. The central scheduler computes the hypothetical average load $L_{\text{AV}-1}$ if one core were to be turned off:

$$L_{\text{AV}-1} = L_{\text{AV}} \cdot \frac{N}{N-1}$$

If $L_{\text{AV}-1}$ is lower than T_E, we know that we can turn off one core and still have a tolerable average load per core. If L_{AV} increases and is greater than T_E we need to turn on a sleeping core. A hysteresis, T_{hyst}, is introduced to avoid oscillation of core states. Both parameters, T_E and the hysteresis, are configurable and can be changed to affect the performance characteristics.

The process of turning cores off and on is usually not instantaneous. The processor hardware may need time to adjust to a new clock frequency, and there may be additional software necessary to handle power management. Therefore it is not profitable to send cores to sleep for short periods. To prevent situations like this, a minimal sleep time, $t_{\text{min_sleep}}$, is used and measured in load balancing periods.

B. Receiver-Initiated Diffusion

To extend the RID algorithm, we use the same approach as described in Section IV-A. However, several details were adapted to meet the requirements of RID. The principle of distributed load balancing schemes stating that all nodes are equal and act independently and asynchronously conflicts with the objective to concentrate the load on fewer nodes. Independent decisions on when to reduce the number of active nodes can lead to situations where multiple cores are switched off simultaneously. This is avoided by introducing a so called

TABLE II. ADDITIONAL MESSAGES INTRODUCED BY THE LOAD ENERGY AWARENESS EXTENSIONS, INCLUDING THEIR ARGUMENTS

Central Scheduler Load Balancing

Message	Description
Sleep Request	Request to enter sleep state, sent by central scheduler
$P_{\text{destination}}$	core that receives remaining load
Sleep Response	Response to sleep request
R	response (*confirmed* or *denied*)

Receiver Initiated Diffusion

Message	Description
Sleep Request	Request to enter sleep state, sent by current owner of sleep token
$P_{\text{destination}}$	core that receives remaining load
Sleep Notification	Broadcast notification that core enters sleep state
P_i	core that enters sleep state
Awake Notification	Broadcast notification that core leaves sleep state
P_i	core that leaves sleep state
Move Token	Message to transfer the sleep token from the current to the new owner
—	(no arguments)

Load Stealing

Message	Description
—	(no additional messages)

sleep token. Only the current owner of the token is allowed to make sleep decisions.

Analog to Section IV-A, a threshold T_E is introduced, describing the tolerable load for one core. The node holding the sleep token computes $L_{\text{AV}-1}$. If this load is less than T_E one node can be turned off. If L_{AV} is bigger than T_E one sleeping node is turned on. To reduce negative effects of the migration of large amounts of load, the algorithm always turns off the node with the smallest load. To prevent oscillation of the number of active nodes, a hysteresis T_{hyst} is introduced.

All nodes participating in the RID algorithm need to be aware of the current domain, i.e., the set of active nodes, in order to make appropriate load balancing decisions. Turning nodes on and off changes the domain. Thus nodes notify other nodes in their domain if they wake up or go to sleep, using Sleep Notification and Awake Notification messages.

To ensure that the nodes are always aware of their current domain, even after waking up from a sleep state, we use a distributed stack like the one described by Aridor et al. [23]. Each node keeps track of the nodes it turned off using a stack. If the owner of the sleep token finds itself as the node with the smallest load in the system, it transfers the sleep token to another node randomly chosen from the active nodes and goes to sleep. If nodes need to be turned on, the current owner pops one node from its stack and wakes it. If the stack is empty it wakes the node it received the sleep token from and returns the token. This way the nodes wake up in reverse order as they went to sleep, and after waking up their information on the current members of their domain will still be correct.

C. Load Stealing

In load stealing no core has information about the load and state of other cores in the system. Therefore, the extensions to adapt the number of active cores according to the current computational demand of the system cannot be based on the load unbalancing schemes as done for the previous algorithms. To our knowledge, no previous work has addressed this problem for the load stealing algorithm. The extension proposed is based on a heuristic principle like load stealing itself.

In addition to the load stealing algorithm described in Section III-C, all cores count the number of consecutive rejected load stealing attempts. If this number exceeds a user defined threshold, T_{denied}, it is assumed that the average system load is low enough to reduce the number of cores, and the core goes to sleep. The other cores are not informed of this, which means that they can continue to send Steal Request that will obviously go unanswered. However, this also means that the core has to wake itself after a certain time, as the other cores are not aware of its sleep phase. The sleep time, T_{sleep}, can be configured by the user. After waking up, the core resets the reject counter, discards Steal Requests that were received during its sleep phase, and tries to acquire load by sending out new Steal Requests.

A core can still have load after the number of denied steal attempts reaches T_{denied}. In this case, the core needs to migrate all its load before going to sleep. As it is generally unknown which cores in the system are currently active, we introduce a so called base core. This core does not participate in the energy awareness protocol, thus it is always active and able to receive excess load from cores going to sleep.

V. EXPERIMENTAL EVALUATION

To evaluate the algorithms, we implemented them on a target system consisting of a 6-Core MIPS32 processor [24]. Each core has a private first level cache, and all cores share one second level cache. No cache coherence fabric is implemented. The target system provides no DVFS support, however, each core provides the functionality to disable its clock if the computational power is not needed.

As mentioned in Section I, we base our work on a multikernel operating system. Scheduling in the space domain is handled by the individual load balancing algorithm and scheduling in the time domain is handled on each core locally. Each core schedules its task set using a round robin scheduler. Ready tasks are managed in a so called ready queue and currently inactive tasks are managed in a so called suspended queue.

The presented algorithms only cover phases two and three of the four load balancing phases. We used the current number of tasks in the run and suspended queue as load descriptor (phase one), as it is the most trustworthy and easiest to measure workload descriptor. Despite the simplicity of this parameter it also shows the best results in the experiments performed by Kunz [25]. Since the load balancing algorithms compute the amount of load to migrate from one core to another as real number and tasks only exist as discrete quantities, we migrate the integer part of the computed values. To select tasks for migration (phase four), we use a simple method. Tasks are always taken from the end of the suspended queue, or, in case this queue is empty, from the end of the run queue.

A. Methodology

The system needs to be under changing load in order to evaluate the implemented load balancing algorithms and their

TABLE III. CONFIGURATION VALUES FOR THE TEST SYSTEM AND ALL ALGORITHMS

System and Load Tasks

$t_{\text{period, OS}}$	1	Operating system tick period in ms
p	20	Load task period in ms
e	4	Load task execution time in ms
t_{call}	2	Simulated call duration in s

CSLB, Basic

$T_{\text{low_min}}$	1	Minimum allowed value for threshold T_{low}
$T_{\text{high_min}}$	3	Minimum allowed value for threshold T_{high}
l	0.95	Factor to compute threshold T_{low}
h	1.40	Factor to compute threshold T_{high}
p_{LB}	25	Load balancing period in OS ticks

RID, Basic

T_{low}	2	Threshold to initiate load balancing
T_{diff}	0.5	Minimal needed difference to L_{AV}
F_{update}	9/10	Factor to compute update interval
p_{LB}	5	Load balancing period in OS ticks

LS, Basic

T_{LS}	3	Threshold to initiate load balancing
T_{diff}	2	Minimal difference needed
p_{LB}	15	Load balancing period in OS ticks

CSLB, Energy Aware

T_E	4	Threshold for load on one node
T_{hyst}	2	Hysteresis used in EA mode
$t_{\text{min_sleep}}$	2	Minimal sleep time in $t_{\text{period,CSLB}}$

RID, Energy Aware

T_E	5	Threshold for load on one node
T_{hyst}	1	Hysteresis used in EA mode

LS, Energy Aware

T_{denied}	6	Number of denied requests before node goes to sleep
t_{sleep}	250	Time one core sleeps in ms

extensions to scale the number of active cores. The target platform we used to implement the algorithms is designed to address next-generation high-density VoIP applications in telecommunication systems. We therefore use artificial tasks with a frequency and duration representative of those used in the respective codecs in telecommunication systems. The tasks used to simulate the behavior of the codec are periodic with implicit deadlines. Each task invokes a series of jobs and each job requires a certain amount of work to perform before the end of the period. Tasks are allowed to preempt or migrate between cores at any time. As we simulate VoIP codecs we use a packet period of 20 ms [26]. The execution time for one packet was computed according to the data given in [27] (for a high complexity codec), and the performance of one core of the target system, resulting in an execution time of 4 ms for each packet of one call.

To obtain clean results, without side effects from tasks other than the load task, only Core 1 to Core 5 are used for measurements. Core 0 is not included in the measurements and used as tester responsible for load generation, logging and trace coordination.

Load generation itself is done in two different patterns, “load step” and “random load”. For the load step pattern a number of tasks enter an empty system at the same time. This represents the worst case scenario for a load balancing algorithm as stated by Willebeek-LeMair and Reeves [14]. The number of tasks, i.e., the height of the load step, can be changed, while all tasks have the same duration.

The random load pattern simulates different calls entering and leaving the system. The calls themselves are modeled using a Poisson distributed random variable for the call arrival and an exponentially distributed random variable for the call duration [28]. To obtain a predefined average number of tasks, n, measurements start out with a load of n calls, and the arrival rate of additional calls is chosen accordingly.

To compare the different algorithms, several properties are measured. This is done using minimally intrusive instrumentation of the software. To achieve this, each core records trace messages for each measurement in private memory which are then collected by the test core and submitted to a host PC for further evaluation. Missed deadlines, the start and end times of tasks and jobs, and all scheduling events are recorded this way.

One important metric used in most evaluations of load balancing algorithms designed for distributed systems is the average job execution time [15], [29], [30]. This is done by instrumenting the start and end time of the job. The average execution time is expected to be larger than the theoretical execution time since operating system overhead and preemption of the task has to be considered. Despite its popularity in distributed systems, this metric is not the most important one for our use case. Since we consider streaming oriented load, the execution time is not important as long as the computation finishes before the deadline. Thus we use a second metric, the percentage of deadline misses during one measurement, to evaluate the algorithms.

To evaluate our energy awareness extensions, we measure the average power during execution. This is done using a power analyzer [31] that allows us to measure the supply voltage of the digital cores separately from the other components of the target system.

The various parameters of the algorithms are chosen according to Table III. The table also includes parameters of the operating system and the load generation.

B. System Utilization

This section compares the system utilization for a different number of load tasks. The utilization shown in the figures was measured for the experiments with load step, both with and without our energy awareness extensions. Fig. 2 compares the three different algorithms without extensions. As expected, the utilization is linear dependent on the number of tasks in the system. At a utilization of 80 %, with a load of $n = 18$ tasks, all algorithms reach saturation and the curves start to differ.

Fig. 3(a) shows the same scenario as Fig. 2 but with enabled extensions to scale the number of active cores. The computed utilization refers to the average number of active cores used by the respective algorithm for the different workload scenarios which is shown in Fig. 3(b). Despite the

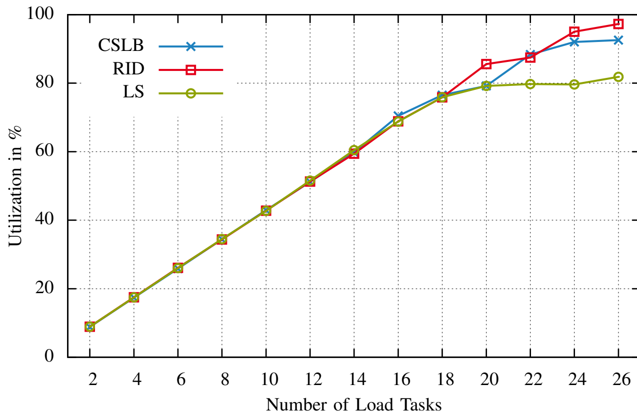
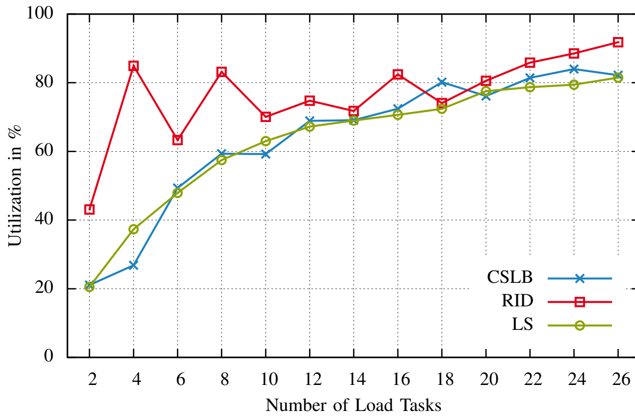
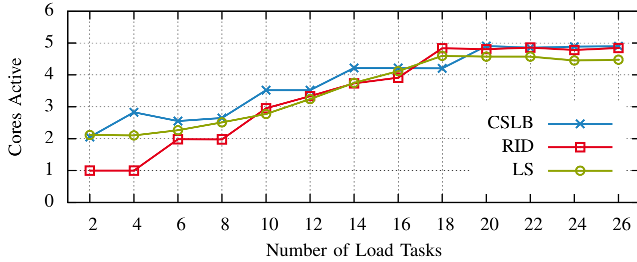


Fig. 2. System utilization during measurements with the “load step” pattern, and load balancing without energy awareness.



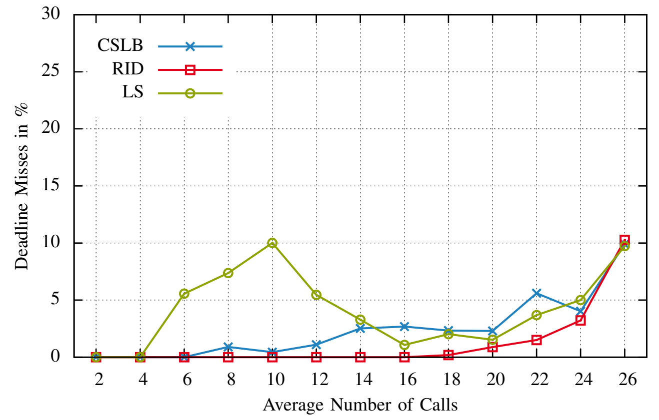
(a) System utilization



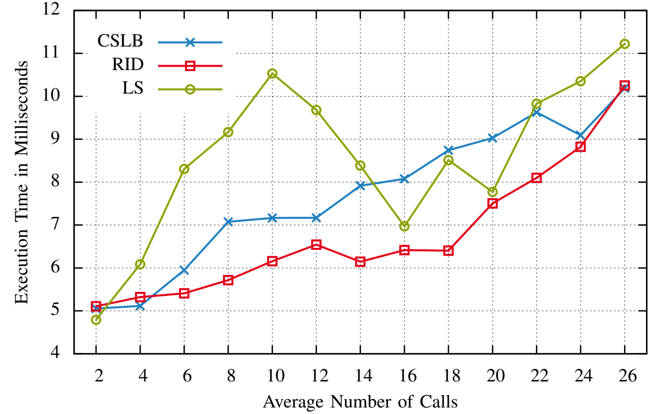
(b) Mean number of active cores

Fig. 3. System utilization and mean number of active cores during measurements with the “load step” pattern, and load balancing with energy awareness.

similarity of the extensions for the CSLB and RID algorithm, the utilization curves differ clearly. Partly because the CSLB algorithm has an additional task, the central scheduler. This task has little computational overhead, compared to the load tasks used to simulate the calls, but the current load descriptor does not distinguish between different task types. Thus the load balancing algorithm provides more computational power than is needed by the actual workload, which can be seen in the figure. The curve of the load stealing algorithm has a more moderate slope than the linear part in Fig. 2 but shows the most continuous curve of the three algorithms due to the heuristics used in the extensions.



(a) Deadline misses



(b) Mean execution time for one job

Fig. 4. Deadline misses and mean job execution times during measurements with the “random load” pattern, and load balancing with energy awareness.

C. Algorithm Comparison

This section compares the algorithms with their energy awareness extensions under realistic conditions. Load was generated according to the “random load” pattern. Fig. 4(a) shows the average percentage of missed deadlines in the measurements and Fig. 4(b) compares the respective average execution times of the jobs taken from the same measurements.

The peak in the curve of the Load Stealing algorithm is due to the nature of the energy awareness extension. If the system loaded is low, there is only a small probability for one steal request to succeed. It is thus likely for one core to count enough rejected requests to reach the threshold T_{denied} and go to sleep. After the system contains enough load and the steal requests are more likely to succeed the load is spread over a greater number of cores and thus the average execution times for the load stealing algorithm declines again. The execution time behaves analogously. The CSLB algorithm has few missed deadlines, starting at 50 % utilization. The RID algorithm again behaves best, with no deadline misses up to $n = 18$ tasks.

All average execution times and deadline misses start to increase after the system contains enough load to reach the saturated utilization levels discussed in Section V-B.

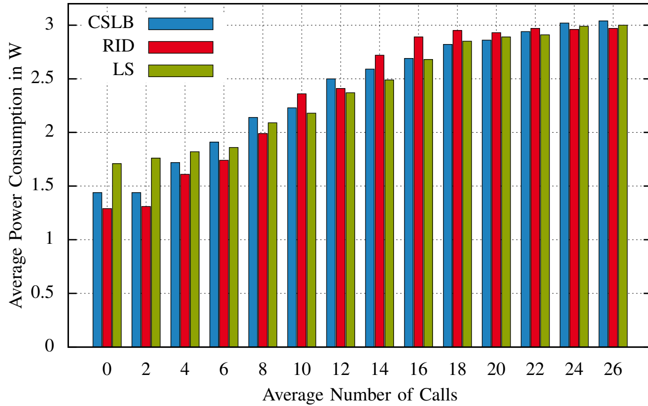


Fig. 5. Comparison of average energy consumption during measurements with the "random load" pattern, and load balancing with energy awareness.

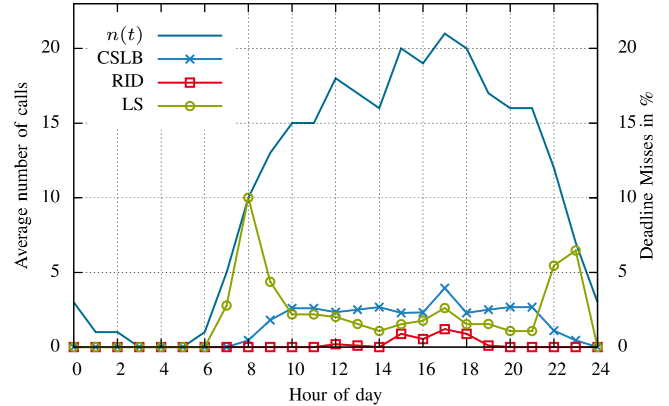


Fig. 6. Load profile for one sample day. Data was obtained from Willkomm et al. [28, Fig. 2].

D. Energy Consumption

Fig. 5 summarizes the results of the measurements with enabled extensions to scale the number of active cores. The diagram shows the average power consumption for the different load levels plotted as bar graph. All three algorithms perform similar. Since the load stealing algorithm constantly searches for load in the system it consumes more energy than its competitors if the system is under less load.

E. Real-World Example

In this section we show the effects of the implemented algorithms if they are applied in a realistic scenario. As we modeled VoIP calls we compare the energy savings of the proposed algorithms if applied to the call profile of one sample day. We used the data provided in Willkomm et al. [28, Fig. 2]. The figure shows the distribution of system wide average call arrival rates over one sample day.

The data was used to compute the utilization for each hour of the day. The results of the power measurements with artificial load at different average number of calls is used as performance data of our algorithms. Note that we used the same average call duration in all our measurements. This simplification is not consistent with real data, as Willkomm et al. [28] showed, but sufficient to show the possible effects of our algorithms. With this assumption we can use Little's Law [32] to compute the average number of calls in the system. This number is then scaled to fit the maximum number of calls in our system. The resulting data $n(t)$ is shown in Fig. 6. Additionally, the average number of deadline misses for each algorithm is depicted for the respective number of active calls. As discussed before, the RID algorithm outperforms its competitors.

We calculate the average power consumed during the sample day as:

$$P_{day} = \frac{1}{24} \sum_{h=0}^{h=23} p(n(h))$$

where h is the hour of the day, $n(t)$ is the average calls per hour, and $p(x)$ is the average power consumption for x calls in the system. Note that one call equals one task in our setup.

Table IV presents the resulting values for the average consumed power P_{day} on the sample day. We compared the three algorithms with extensions to scale the number of active cores and the CSLB algorithm without extensions as a representative of algorithms which do not adapt the number of active cores during runtime. Additionally, the savings compared to the algorithm without energy awareness are given in percent. As we used one core to manage the load generation and to record the trace data, this core is included in the raw measurements. Therefore the table contains two evaluations. The first columns ("Raw Values") show the average power consumption of all cores, measured using the power analyzer. Core 0 is part of the measurement setup, so it does not contribute to the actual application. As Core 0 never enters a sleep state, it is responsible for $1/6$ of the total power consumption without energy awareness, which amounts to $0.5\,\text{W}$. The rightmost columns of Table IV show the power savings, only considering the cores that actually execute the application. Also the CSLB algorithm consumes more power than its competitors in the range of 10 to 18 calls (see Fig. 5) it performs best in this experiment. This is due to the call profile of the sample day. Only minor part of the time is spent in the range of 10 to 18 calls where CSLB performs worse, most time is spent either in the fully utilized state where all algorithms perform almost equal or in the less utilized state where CSLB performs almost as well as RID.

VI. Conclusion

In this paper, we explored different load balancing strategies for embedded multicore systems. We adapted and extended three prominent algorithms from the area of distributed computing. Our extensions make the algorithms "energy aware", which means that they dynamically scale the number of active cores according to the current system load. We implemented the algorithms and evaluated them by measuring deadline misses and energy consumption of an embedded soft real-time application.

We showed that load balancing algorithms can be successfully applied in embedded applications, and that our energy awareness extensions can significantly reduce the energy demand of the system. Applied to a real-word example, a VoIP application processing telephone calls with a realistic load, up to 30% of the energy consumed by the cores was saved.

TABLE IV. POWER CONSUMPTION OF THE SYSTEM DURING THE SAMPLE DAY, COMPARING THE ENERGY AWARENESS EXTENSIONS OF ALL THREE LOAD BALANCING ALGORITHMS. THE POWER CONSUMED WHEN ALL CORES ARE ACTIVE AT ALL TIMES IS USED AS A REFERENCE TO COMPUTE THE OBTAINED SAVINGS.

	Raw Measurements		**Without Core 0**	
Algorithm	**Av. Power**	**Reduction**	**Av. Power**	**Reduction**
Central Scheduler LB	2.2625 W	24.6 %	1.7625 W	29.5 %
Receiver-Initiated Diffusion	2.2888 W	23.7 %	1.7888 W	28.5 %
Load Stealing	2.3358 W	22.1 %	1.8358 W	26.6 %
No Energy Awareness	3.0000 W	0.0 %	2.5000 W	0.0 %

The optimal choice of algorithm and its parameters depends on the requirements of the application. There is a trade-off between energy consumption and the expected number of missed deadlines. We found that Receiver Initiated Diffusion provides the best performance in terms of energy savings to missed deadlines ratio, out of the three examined algorithms.

REFERENCES

[1] J. D. Ullman, "Np-complete scheduling problems," *J. Comput. Syst. Sci.*, vol. 10, no. 3, pp. 384–393, Jun. 1975.

[2] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *9th USENIX conference on Operating systems design and implementation*, OSDI'10, pp. 1–8, 2010.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *22nd symposium on Operating systems principles*, SOSP '09, pp. 29–44, 2009.

[4] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs, "Your computer is already a distributed system. why isn't your os?" in *12th conference on Hot topics in operating systems*, HotOS'09, pp. 12–12, 2009.

[5] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Power-aware scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 584–600, May 2004.

[6] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *50th Annual Design Automation Conference*, DAC '13, pp. 174:1–174:9, 2013

[7] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu, "Dynamic thread scheduling in asymmetric multicores to maximize performance-per-watt," in *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pp. 964–971, 2012.

[8] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *20th Conference on Supercomputing*, SC '07, pp. 1–11, 2007.

[9] E. Pinheiro, R. Bianchini, E. V. Carrera, , and T. Heath, "Load balancing and unbalancing for power and performance in cluster-based systems," in *Department of Computer Science Rutgers University Technical Report DCS-TR-440*, 2011.

[10] H. Jeon, W. H. Lee, and S. W. Chung, "Load unbalancing strategy for multicore embedded processors," *IEEE Transactions on Computers*, vol. 59, no. 10, pp. 1434–1440, 2010.

[11] J. Liu and M. Yang, "Task scheduling of real-time systems on multi-core embedded processor," in *18th International Conference on Intelligent Systems and Knowledge Engineering*, ISKE '10, pp. 580–583, 2010.

[12] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, 2009.

[13] S. Holmbacka, W. Lund, S. Lafond, and J. Lilius, "Task migration for dynamic power and performance characteristics on many-core distributed operating systems," in *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '13, pp. 310–317, 2013.

[14] M. Willebeek-LeMair and A. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Transactions o Parallel and Distributed Systems*, vol. 4, no. 9, pp. 979–993, 1993.

[15] Y. Lan and T. Yu, "A dynamic central scheduler load balancing mechanism," in *14th Annual International Phoenix Conference on Computers and Communications*, pp. 734–740. 1995.

[16] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *10th annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pp. 119–129, 1998.

[17] P. Berenbrink, T. Friedetzky, and L. Goldberg, "The natural work-stealing algorithm is stable," in *42nd IEEE Symposium on Foundations of Computer Science*, FOCS '01, pp. 178–187, 2001.

[18] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *17th ACM symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pp. 21–28, 2005.

[19] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.

[20] E. Luque, A. Ripoll, A. Cortes, and T. Margalef, "A distributed diffusion method for dynamic load balancing on parallel computers," in *Euromicro Workshop on Parallel and Distributed Processing*, pp. 43–50, 1995.

[21] D. Hendler and N. Shavit, "Non-blocking steal-half work queues," in *21st annual symposium on Principles of Distributed Computing*, PODC '02, pp. 280–289, 2002.

[22] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, "A simple load balancing scheme for task allocation in parallel machines," in *3rd annual ACM symposium on Parallel Algorithms and Architectures*, SPAA '91, pp. 237–245, 1991.

[23] Y. Aridor, M. Factor, and A. Teperman, "cjvm: A single system image of a jvm on a cluster," in *28th International Conference on Parallel Processing*, ICPP '99, pp. 4–11, 1999.

[24] *MIPS32 24KEc Processor Core Datasheet*, MIPS Technologies, 2008.

[25] T. Kunz, "The influence of different workload descriptions on a heuristic load balancing scheme," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 725–730, 1991.

[26] *VoIP Bandwidth Calculation*, Newport Networks, 2005.

[27] *Cisco Unified Communications Solution Reference Network Design (SRND)*, Cisco Systems, Inc., Americas Headquarters Cisco Systems, Inc. 170 West Tasman Drive San Jose, CA 95134-1706 USA, 2008.

[28] D. Willkomm, S. Machiraju, J. Bolot, and A. Wolisz, "Primary user behavior in cellular networks and implications for dynamic spectrum access," *IEEE Communications Magazine*, vol. 47, no. 3, pp. 88–95, 2009.

[29] T. Suen and J. Wong, "Efficient task migration algorithm for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 4, pp. 488–499, 1992.

[30] S. Zhou, "A trace-driven simulation study of dynamic load balancing," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1327–1341, 1988.

[31] *Agilent Technologies DC Power Analyzer - Model N6705*, 9th ed., Agilent Technologies, Inc. 2007-2012, 2012.

[32] J. D. C. Little, "Or forum—little's law as viewed on its 50th anniversary," *Oper. Res.*, vol. 59, no. 3, pp. 536–549, 2011.