**Appears in:** Proceedings of the 20th International Symposium on Real-Time Distributed Computing (ISORC 2017)
Toronto, Canada, 16–18 May 2017

# In the Heat of Conflict: On the Synchronisation of Critical Sections

Stefan Reif, Timo Hönig, and Wolfgang Schröder-Preikschat
Department of Computer Science, Distributed Systems and Operating Systems
Friedrich-Alexander University Erlangen-Nürnberg (FAU)

***Abstract*—Advances in semiconductor technology greatly extend the scope of special-purpose applications as multi-core processors find the way into embedded systems. The increasing number of processor cores makes it more important than ever to have real-time operating systems process parallel threads in the most efficient way. In doing so, they have to pursue multiple (often conflicting) goals: namely being predictable as to time and energy demand.**

**In shared-memory multi-core systems, contention at critical sections makes it inevitable for the operating system to execute competing threads with highly efficient synchronisation methods. Related research has primarily focussed on timing aspects of synchronisation methods, while the energy efficiency of the latter is an unexplored field, yet.**

**In this paper, we implement and evaluate five distinct synchronisation methods and analyse their run-time characteristics (i.e., time, energy) in-depth. We evaluate the overall demand at application level, and empirically prove that contention increases the energy demand significantly even when competing processes are temporarily suspended. Furthermore, the evaluation reveals that choosing the right synchronisation method can decrease the energy demand by more than a factor of 5. We come to the conclusion that it is mandatory to consider the effects of process synchronisation for energy analysis and energy-efficiency optimisations.**

## I. Introduction

Energy is an exceedingly valuable computing resource, both in economical and ecological respect. Everything counts in large amounts! This is especially true for embedded systems, which are so ubiquitous today and yet virtually imperceptible [1]. As a matter of fact, billions of processors of existing diversity [2] need to be continuously powered. Energy use as little as a $\mu$J in a small scale easily accumulates to an order of magnitude of megawatts in a large scale when considering the bulk freight of end devices. The *Internet of Things* (IoT) is one of the finest examples of such a scenario: from 6.4 billion IoT units in 2016, the number of connected devices is forecasted with 20.8 billion units in 2020. [1] Albeit each individual unit consumes little amounts of energy, only, the accumulated energy demand of all IoT devices is several times the energy demand of data processing centres or supercomputers, already today—such as the 17.6 MW spent by the Tihane-2 [3].

A significant share of those IoT devices is assigned to embedded systems, while a great many of which have to operate under real-time constraints. The hardware trend leads towards multi-core processors even for small individual devices, embedded systems are no exception. Real parallelism not only intensifies the already existing problem of non-deterministic operation, but it also amplifies interference with real-time scheduling decisions due to more difficult synchronisation methods needed to ensure consistent operation in the data as well as time domain. A fact that is not only limited to event-triggered systems but addresses time-triggered systems equally. As energy more and more plays a key role also for real-time embedded multi-core systems, it should also become an important factor when it comes to making scheduling decisions. In addition to ensuring that a process meets its timing deadline, energy-aware process schedulers also consider whether or not there is enough energy available in the system for the process to complete execution [4].

A key problem in forthcoming many-core processor designs is their inherent power limitation just to avoid overheating or even blowing of circuitry (*dark silicon* [5]). Constructive means as to the structuring of non-sequential programs are required to prevent *hotspots*, in the truest sense of the word, that would be caused by concurrent processes in the moment of accessing shared data structures or resources, respectively. Here, contention under processes forced to synchronise according to program instructions provoke *thermal stress* of the hardware. CPU throttling would be the consequence, which results in lower speed, risks deadline miss and, thus, affects the real-time behaviour of the whole system. Especially the processor cores that settle the contending processes are battered. It may be assumed that the thermal characteristic of such a hotspot depends, at least, on the degree of contention, the synchronisation principle (i.e., blocking or non-blocking), the respective synchronisation protocol, and the "hunger for energy" as well as frequency and interval of the atomic machine instructions to be executed, likewise.

Contribution of the paper is the investigation and analysis of the energy consumption of protocols used for the synchronisation of critical sections. Purpose is to figure out pros and cons of conventional and unconventional concepts (i.e., Pthread mutex and ticket lock on the one hand and so-called "guarded section" [6] on the other hand) for enforced sequential execution of single program sections in terms of energy balance and, thus, thermal characteristic. The measurements presented show that contention is the crucial point, which stands or falls with the structure of the respective non-sequential program

[1] www.gartner.com/newsroom/id/3165317

and the scope of application and use case. We also show that there is no panacea in the attempt of decreasing energy consumption other than providing a bunch of solutions that are each specialised to a certain level of contention. Key point then is to select from a *family of dedicated programs* [7] the most suitable variant as a function of the actual degree of contention.

The remainder of the paper is structured as follows: Section 2 gives some background information on the synchronisation methods that were evaluated and compared with each other. Focal point thereby are guarded sections, a means of synchronisation of usual critical sections without mutual exclusion of concurrent processes. Section 3 goes into implementation aspects, presents the evaluation, and analyses the measuring results. Section 4 discusses related work in the context of energy-aware system software and Section 5 concludes the paper.

## II. Background

Object of inquiry is blocking synchronisation of critical sections compared to serialised execution of guarded sections. Common to both approaches is that concurrent processes will “be constructed in such a way, that at any moment at most one of [them] is engaged in its [particular] section” (acc. [8, p. 11]) in order to prevent a *race condition*. Consequence of such a construction is the sequential execution of a specified program section. The difference is how this sequential execution is achieved: by means of mutual exclusion of concurrent processes, on the one hand, or successive processing of concurrent requests, on the other hand. Guarded sections follow the latter model, they never entail mutual exclusion of processes. This is in contrast to blocking synchronisation, widely used in terms of a *ticket lock* or *MCS lock*, respectively, [9] or a *Pthread mutex* [10].

In contrast to common multilateral synchronisation based on mutual exclusion, guarded sections [6] do not force contending processes to block. Instead, each process encapsulates guarded sections in closures, and submits them to the guard. Using the guard protocols, contending threads elect a control thread that sequentially executes all requests. Due the potentially asynchronous execution of guarded sections, the control thread has to signal availability of results in case of data dependencies using, for instance, a *future* [11] variable. Since the guard protocols are wait-free, they are particularly suited for time-critical systems.

None of these *explicit synchronisation* techniques is free of charge. Cost factors are the actual implementation, the programming language, the compiler, and finally the instruction set architecture (ISA) of the underlying processor. Just to give a feeling: using GCC 4.2.1[2], the number of instructions in the absence of contention is 6 for a ticket lock [9, p. 27], 12 for a guarded section [6], 17 for a mutex [10, p. 106], likewise excluding the respective subroutine call but relying on atomic read-modify-write operations; for various fast mutual exclusion algorithms that only use atomic read-write operations, it is 9 for Merritt’s [12], 12 for Michael’s [13], and 15 for Lamport’s [14] solution. The lower contention, the more relevant becomes this *shortest path* (i.e., the minimum number of instructions) for performance as well as energy efficiency. But it also means that the shortest path of a critical/guarded section should be reasonably longer in order to lose unnecessary synchronisation overhead in the “background noise” of the actual functional code of that particular section.

[2]`gcc -O3 -static -m32 -fomit-frame-pointer`, x86

The other extreme is the *longest path* through the entry protocol, which also refers to a key property of algorithms for mutual exclusion as far as *liveness* is concerned. Namely, every waiting process is eventually granted access to the critical section (*starvation freedom*), if and only if the longest path is finite. This property particularly holds for guarded sections. There, processes that contend for running a guarded section proceed not only *non-blocking*, but even *wait-free* [15]. In consequence, they cannot starve. This is an important aspect when energy—but also timing—assessments are made (refer to Sec. III). Without trying to get ahead of the result, but executing less instructions often lowers the energy consumption or heat dissipation. This property is strengthened as the processing model of a guarded section also facilitates strong *process locality*, namely as to the process that controls the particular guarded section.

## III. Implementation and Evaluation

Synchronisation is a means for the coordination of cooperation and communication of *interacting processes*, thus concurrent processes that access common variables or share resources. For the methods investigated as to energy demand in the following, synchronisation applies to critical sections in a wider sense and operates blocking or non-blocking on a process. The difference is in the entry protocol, which either delays (blocking) or detours (non-blocking) processes arriving at such a section. In any case, however, the instruction sequence building this section is guaranteed to be executed as a whole one at a time. Thus, *sequential execution* of a specific program section is common denominator, but achieved in different ways and tainted with different non-functional properties (i.e., performance, energy demand).

In the study, blocking characteristic comes with Pthread mutex [10] and an own variant of a ticket lock [9], while non-blocking characteristic is given with guarded sections [6]. Blocking may be a *passive* or an *active* measure of the particular process to await admission to the critical section. If passive, the current process changes to “blocked” state by instructing the scheduler to release control of the processor, allowing it to enter a sleep state. If active, the current process remains in “running” state (*busy waiting*).

### *A. Synchronisation Methods*

Pthread mutex (q.v. Fig. 2 and 3, `PM`) is included in the evaluation for its widespread use. This method causes the current process to wait passively until its turn to enter the critical section that is synchronised by the respective mutex.

```
procedure TICKET_PASSIVE_LOCK($lock$)
    $t \leftarrow$ FAA($lock.ticket, 1$)
    loop
        PREPARE( )                ▷ be sensitive to wake-up call
        if $lock.cur = t$ then
            UNDO_PREPARE( )       ▷ ignore wake-up call
            return
        end if
        SLEEP( )                  ▷ await wake-up call
    end loop
end procedure
procedure TICKET_PASSIVE_UNLOCK($lock$)
    FAA($lock.cur, 1$)
    NOTIFY( )                     ▷ wake up all sensitive threads
end procedure
```

**Fig. 1:** Passive-mode (i.e., sleeping) ticket lock.

Ticket locks come in passive and active mode. If the critical section is vacant and in the absence of contention, the current process follows a beeline in both variants to occupy that section. Thereby, the shortest path of the passive-mode variant (i.e., 11 instructions, x86) is longer than the one of the active-mode variant (6 instructions): there is no need for the latter to interact with the scheduler (cf. Fig. 1). When catching an occupied critical section, the current process waits either passively or actively, depending on the ticket-lock variant used. Ticket-lock passive (TLP) mode resembles a *sleeping lock*: the process asks the scheduler for immediate release of the processor. In contrast, ticket-lock active (TLA) means a *raw lock*. In particular, the process performs *busy waiting* on the appearance of its ticket number.

Fig. 1 sketches a TLP variant. The code sets a *sensitivity flag* stored in a thread-local variable to indicate (PREPARE) that a processor core is possibly going to sleep, before actually checking the sleeping condition. In sleep state, a wakeup signal (i.e., *inter-processor interrupt*, IPI) is needed to resume normal processor-core operation. The notification routine (NOTIFY) first checks for sensitivity in order to avoid sending a spurious IPI. In the case where the executions of lock and unlock overlap, marking a core as sensitive before checking the waiting condition can cause unnecessary notifications, but prevents the lost-wakeup problem. In the case where waiting is not required, the process clears the sensitivity flag (UNDO_PREPARE) and continues.

Similar to ticket locks, guarded sections also come in passive and active mode. However, in this case, the particular mode does not address the entry protocol of a critical section synchronised in this vein: by concept, processes do not block at entrance to a guarded section. Rather, the mode applies to the protocol to synchronise on the event of the assignment of a value to a *future variable*.

Implementation of a guard raises the question of internal synchronisation. As threads are free to call guard functions concurrently, they have to ensure consistency of the internal state, such as an occupation indicator, and the job queue. The original concept [6] uses a wait-free solution, which has relatively complex entry, exit, and thread notification protocols. Additionally, we compare this solution against a cut-down version, which uses locks for protection of internal guard data structures: the lock does not protect the guarded section of the application, but critical sections inside guard protocols. In particular, the sequencer (i.e., control process) does not hold this lock while running a guarded section, and thus allows other processes to enqueue jobs in parallel. Most importantly, the lock-based entry protocol does not force conflicting threads to wait until the sequencer finishes a guarded section. For the application, the internal synchronisation method is entirely transparent in functional terms.

As variation of the internal guard synchronisation spans a large diversity of guard implementations, the evaluation narrows down to two representative versions. First, the GWP variant implements the entry and exit protocol in the original wait-free manner, and uses passive-mode futures. Second, the GTA alternative uses active-mode ticket locks for internal synchronisation of these protocols and active-mode futures for signalling. In our experiments, other guard variants show similar properties. In summary, the presented synchronisation methods allow distinction not only between guards and mutexes, but also between active and passive waiting operations.

## B. Energy Measurement Procedure

A general problem of energy evaluation of concurrent threads is the intrinsic unpredictability of thread overlapping and interaction patterns. In particular, inter-thread dependencies occur at mechanisms for mutual exclusion. However, interacting threads mutually influence their run-time behaviour and, consequently, energy demand. For instance, execution time and energy demand of a lock-acquire function depend on the lock state—if the lock is free, a lock-acquire function needs only little time and energy. However, if the lock is occupied, resource usage depends on the waiting time, and thus, on other threads. Hence, concurrent threads mutually influence their energy and performance characteristics. In consequence, the naive approach to evaluate the energy demand of code fragments is not applicable to interacting parallel control flows. In summary, concurrency obfuscates the origins of energy issues—only imprecise information about activities on concurrent threads is available, but they influence energy consumption.

To mitigate this energy attribution problem, we measure the energy demand at coarse granularity. First, the energy measurement encompasses all cores. Second, every measurement includes multiple critical sections to abstract from specific overlapping patterns. Thus, the evaluation combines all threads and inter-thread effects.

## C. Experimental Setup

The experiments were executed on an Intel Xeon processor (Intel Xeon E3-1245v3) that fulfils both prerequisites for our experiments. First, the processor provides four cores (with two hardware threads each) running at 3.4 GHz. The processor thus allows parallel execution of eight application threads. Second, the processor offers a built-in mechanism for energy evaluation through the RAPL [16] interface. Monitoring the *package* domain, RAPL measures the energy consumption of all cores, caches, and the memory controller, but it excludes irrelevant off-chip components (e.g. network interfaces). On top of this platform, a Linux environment provides a reliable tool-chain for energy and execution time evaluation. For performance measurement, Linux evaluates the elapsed wall-clock time of an entire application execution; in parallel, RAPL evaluates the energy demand.

Most importantly, we want to compare experimental results at different degrees of contention, and different synchronisation methods, in order to identify the influence on the energy demand of the application. In consequence, experiment results need to be comparable despite different parameters. We therefore evaluate a micro-benchmark application where we can freely manipulate the degree of contention and the synchronisation method. Lozi et al. have experimentally examined the proportion between critical sections and the overall application execution time [17], for popular applications and benchmarks. Similarly, we define the degree of contention $C$ as the relative amount of time that the application spends in the critical or guarded section, excluding waiting times:

$$C = \frac{t_{critical}}{t_{critical} + t_{non\text{-}critical}} \tag{1}$$

This definition of contention determines the micro-benchmark design. We control the contention and, consequently, the probability that synchronisation requests conflict at run-time, by the duration of critical and interjacent sections. In particular, the benchmarking application consists of equivalent threads, which each executes a loop that contains a critical and a non-critical section. Both sections are made of time-controllable lag elements. Hence, the benchmark application structure allows experimental manipulation to the degree of contention, and thus provides information how conflict for shared resources affects energy consumption. The number of threads is thereby fixed to eight, matching the number of processor cores. In consequence, the total amount of work performed by the application remains constant, despite variation of the contention and the synchronisation method, and thus allows comparison of evaluation results. Our experiments focus on the range of 20 % to 80 % contention. This range balances the degree of contention in contemporary applications [17] and the trend that, in future many-core systems, the conflict for shared resources is expected to increase [18].

### D. Total Energy Demand

In the first experiment, the measurement covers a full application execution, where each application thread executes 1,000 critical and non-critical code sections in a loop. The plot in Fig. 2a depicts the measured energy demand of the entire application, and Fig. 2b visualises the execution time. Fig. 2c depicts the *energy-delay-product* (EDP) [19]. All three plots

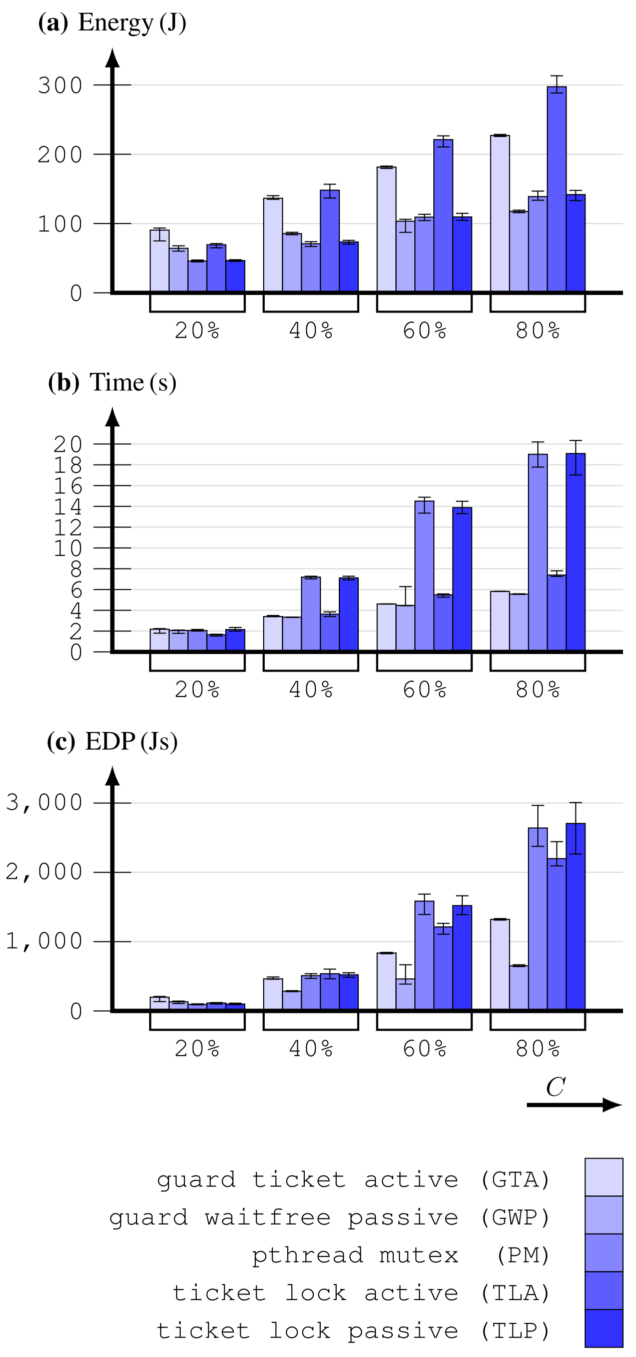

**Fig. 2:** Energy consumption, execution time, and energy-delay-product of an application, depending on the synchronisation method and the degree of contention.

show the arithmetic mean of 100 repetitions, along with the minimum and maximum of all observed values. The horizontal axis groups data by the degree of contention, as defined in (1). The following paragraphs analyse the evaluation results.

*1) Active vs. Passive Waiting:* Comparing the energy demand of the synchronisation methods shows that waiting passively reduces the energy consumption significantly. The only exception is GWP at low contention: the method is relatively inefficient in this situation due to its complex wait-free protocols. However, when increasing contention, GWP is the most energy efficient synchronisation method, because the control process switches between guarded sections efficiently without inter-core communication. In general, synchronisation methods that wait actively have a relatively high energy demand, at all levels of contention. For instance, TLA locks require 49 %[3] more energy than TLP at 20 % contention. Similarly, the active-mode GTA requires 41 % more energy than the GWP alternative, which lets processor cores sleep in the case of contention. In summary, waiting actively increases the energy demand by nearly 50 %.

*2) Guards vs. Locks:* When comparing active guards (GTA) to locks (TLA), the latter only needs less energy in the case of low contention. In numbers, GTA needs 30 % more energy than TLA at 20 % contention. At high contention, the effect reverses, so TLA consumes more energy than GTA. For instance, at 60 % contention, TLA needs 22 % more energy than GTA. Similarly, passive-mode guards (GWP) are the most energy efficient synchronisation method at high contention. In summary, guards are generally more energy efficient than locks at high contention.

*3) Performance:* Regarding execution time, passively waiting locks and mutexes are relatively slow. In contrast, active-mode locks (TLA), and both guard variants, are faster. At 20 % contention, TLP and PM need 30 % and 25 %, respectively, more time than TLA. These performance results are in line with previous research of the run-time performance of synchronisation methods [10].

When contention is 60 %, TLP is 153 % slower than the actively waiting counterpart, TLA. However, the passive locks consume less energy. In numbers, TLA needs 101 % more energy than TLP. In summary, waiting actively is faster, but waiting passively consumes less energy.

*4) Energy to Performance Trade-off:* In order to evaluate the energy-to-performance trade-off, we use the EDP as metric. The EDP represents energy efficiency by combining execution time and energy consumption. It tolerates a non-optimal energy consumption, as long as it is justified by a corresponding performance gain. Fig. 2c shows the EDP for all evaluated synchronisation methods at different contention levels.

Evaluation of the EDP shows that GWP is the most efficient synchronisation method at high contention. At 60 % contention, this synchronisation method has a relatively low energy demand and is also fast. In consequence, the EDP of TLA is 163 % higher, compared to GWP. Similarly, the EDP of PM is 243 % above. This result shows that energy efficiency does not contradict high performance, but finding an efficient synchronisation method depends on the use case.

In this evaluation, the execution time differences dominate the energy differences between synchronisation methods. In consequence, waiting actively can be more efficient than passive waiting with respect to the EDP. For instance, the EDP of TLP is 25 % higher, compared to TLA, at 60 % contention. Regarding guarded sections, however, the EDP of the passive variant (GWP) is better than its actively waiting counterpart (GTA), because both have similar execution time, and the passive version consumes less energy. In summary, waiting actively can be reasonable, if the performance gain outweighs the energy overhead. However, guards have a better EDP than all evaluated locks, except for low contention.

[3]Comparison of experiment result data always references the difference, in relation to the smaller value. For instance, comparing 20 J to 21 J yields an increment of 5 %.

## *E. Isolated Application Energy Demand*

In a second experiment, we have taken into account that the system consumes energy even in idle state. For instance, the hardware, Linux kernel threads, and system daemon processes consume a certain amount of energy even when no application runs. Motivated by systems that run in non-stop mode, we re-evaluate the micro-benchmark application, using a different perspective. In this scenario, we use the energy demand of an idle system as baseline, and evaluate the energy difference caused by the application execution.

In order to compute the application energy overhead, we evaluate the energy demand of idleness. This idle energy resembles the static energy demand [20], but it also takes system processes into account. For experimental evaluation of the energy demand of doing nothing, RAPL measures the energy consumption of an application that sleeps using `nanosleep`. The results of this experiment show a linear relation between the sleeping time and the corresponding energy demand, with a relatively small constant offset. This proportionality matches the expectation that an idle system consumes a constant wattage. A least squares approximation of 1,000 measurement iterations yields the following linear formula:

$$E_{idle}(t) = 5.28\,\mathrm{W} \cdot t + 0.60\,\mathrm{J} \tag{2}$$

Using this relation, we reconsider the results of the experiment in Section III-D. We insert the observed execution time into (2) and thus calculate the amount of energy consumed by the hardware and the operating system. Fig. 3a depicts both the overall energy consumption of the application and, as the lower blank part of each bar, the corresponding idle energy demand. Fig. 3b shows the energy overhead of the application, which is the difference between the overall application energy, and the idle energy. This particular energy difference is caused by the application execution and therefore in focus of the discussion below. Fig. 3c displays the product of the energy overhead and the application execution time. The following paragraphs discuss the results in detail.

*1) Active vs. Passive Waiting:* Isolating the application energy overhead decouples energy demand from execution time. In the evaluation in Section III-D, the power consumption of idleness penalises slow synchronisation methods. In contrast, excluding the idle energy demand rewards the usage of low-power sleeping modes regardless of their performance implications. Consequently, synchronisation methods that wait

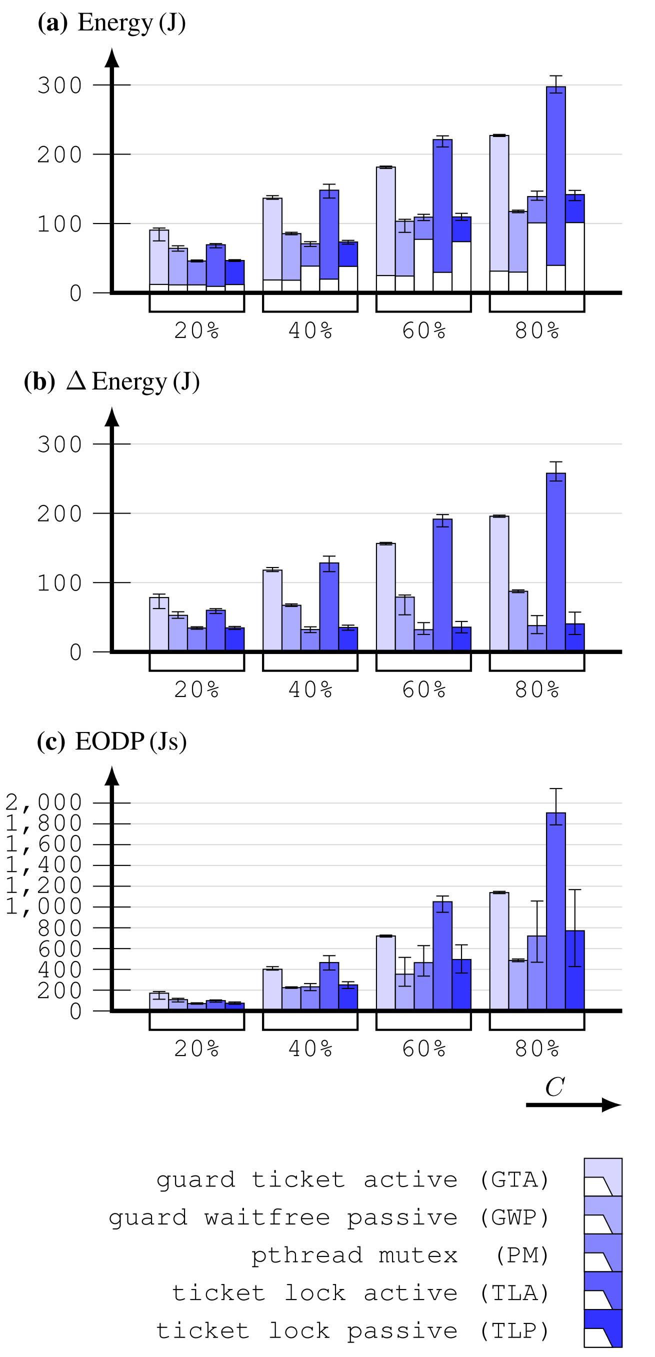

**Fig. 3:** Reconsidering the application energy overhead, taking the energy demand of idleness into account.

actively cause a higher energy overhead than passive alternatives. For instance, the actively waiting `GTA` consumes 128 % more energy than `PM` at 20 % contention. When increasing the degree of contention, the energy differences between synchronisation methods grow. At 80 %, `TLA` demand 195 % more energy than `GWP`. In summary, waiting actively is highly inefficient regarding the application energy overhead.

Similar to the evaluation in Section III-D, contention has a negative effect on the energy consumption. More precisely, a comparison of different contention levels, while leaving the synchronisation method constant, shows that the energy overhead grows when contention increases. For instance, the energy demand of `PM` grows by 10 %, between 20 % and 80 % contention. Alternatively, the energy demand of `TLP` increases by 17 %. However, both of these factors appear to be only a minor effect, considering that the energy overhead of `TLA` and `GTA` increases by 150 % and 330 %, respectively. In summary, contention increases the energy overhead in all evaluated scenarios.

*2) Mutexes vs. Guards:* Regardless of the contention, mutual exclusion has the lowest energy overhead, if waiting passively. Namely, `PM` and `TLP` show the highest efficiency amongst all evaluated synchronisation methods. For instance, active `TLA` consume 74 % more energy than their passively waiting counterpart at 20 % contention. At 80 % contention, the energy overhead difference is even 538 %. Thus, passively waiting locks are much more efficient than their competitors.

*3) Energy to Performance Trade-off:* To combine performance and energy demand, we multiply the energy overhead with the execution time. The resulting *energy-overhead-delay-product* (EODP) resembles the EDP, but it excludes the idle energy consumption. Fig. 3c displays the results for all five evaluated synchronisation methods at different degrees of contention. Regarding the EODP, waiting actively is less efficient than waiting passively, in all observed cases. For instance, the EODP of `TLA` is 86 % above their passive counterpart at 40 % contention. Similarly, `GTA` have a 79 % higher EODP than the alternative `GWP`.

While the passive-mode mutex and lock (`PM` and `TLP`) have the lowest energy overhead of all evaluated synchronisation methods, regardless of the degree of contention, they do not have the lowest EODP. More precisely, their EODP suffers from the relatively high execution time. In the experiments, their execution time is 242 %, or 244 %, above the `GWP` variant, at 80 % contention. In consequence, the product of energy overhead and execution time is therefore not optimal for passive-mode locks. In numbers, the EODP of `PM` is 48 % above `GWP`, and the EODP of `TLP` is 59 % higher. However, `TLA` has the highest EODP. In the evaluation, the EODP of `TLA` is 292 % higher than `GWP`, at 80 % contention. Thus, choosing the correct synchronisation method can increase energy efficiency drastically.

Similar to other energy efficiency metrics, the degree of contention is vital for the EODP. For `GWP`, increasing the degree of contention from 20 % to 80 % increases the EODP by 346 %. In the same scenario, the EODP of `PM` and `TLP` grow by 917 % and 938 %, respectively. For `TLA`, the EODP even grows by $1,823$ %. Even though higher contention implies longer execution time for our setup, because longer code sections are serialised, the EODP increase is over-proportional. In summary, all evaluated synchronisation methods become either much slower, or energy hungry, in the case of conflict. Therefore, reducing the degree of contention is crucial for energy optimisation.

### F. Analysis

The evaluation results show that energy optimisation of critical sections is a complex challenge, where a multitude of factors have to be considered. For instance, guards tend to be more energy efficient than similar mutex-based synchronisation methods at high contention. Similarly, synchronisation methods that wait actively consume more energy than their passive counterparts. However, none of these statements holds for all evaluated situations. Since the energy demand of synchronisation is significant, but complex to predict, we propose adaption of existing tool-chains for energy-aware software development [21] to parallel applications. Then, optimising synchronisation for energy can be automatised and hence assist the programmer to optimise multi-threaded applications.

As a general result, contention is an enemy of energy efficiency. In the experiments, decreasing the degree of contention reliably reduces the energy demand of the application. Therefore, energy-aware application design should focus on reducing contention for shared resources. Furthermore, the results imply that energy analysis tools depend on an accurate prediction of waiting times.

In the experiments, synchronisation methods that wait actively consume relatively much energy, especially at high contention. Both of the `TLA` and `GTA` synchronisation methods require considerably more energy than their passively waiting counterparts. For instance, an application using `TLA` consumes $49\,\%$ more energy, compared to using `TLP`, at $20\,\%$ contention. Meanwhile, the execution time shows only minor differences in this scenario. However, waiting actively can be energy efficient, when the performance gain justifies the energy costs. Such a system can profit from energy savings by shutting down the hardware early (*race-to-halt* [22]).

When taking into account that the system needs energy even in idle state, the energy differences between synchronisation methods increase. While passively waiting locks, guards and mutexes have a relatively low energy overhead, compared to an idle system, waiting actively is inefficient. For instance, `TLA` consumes $195\,\%$ more energy than `GWP` at $80\,\%$ contention. Similarly, passive waiting is generally more efficient than active waiting when regarding the EODP.

Contrary to our expectations, increasing the degree of contention leads to a higher energy demand even for synchronisation methods that wait passively. Despite no application code is executed while waiting passively, sleeping causes a measurable energy consumption. For instance, `PM` consumes $10\,\%$ more energy when increasing the contention from $20\,\%$ to $80\,\%$. In consequence, passive waiting is not energy neutral, and energy analysis tools must consider sleeping times. Elimination of this energy overhead would generally require the operating system to utilise hardware sleep states more efficiently.

As a bottom line of our results, energy evaluation of parallel applications has to consider concurrency aspects. Only an analysis that includes the effect of contention can provide reliable results. In the experiments, contention increases the energy overhead by up to $330\,\%$, and the EODP increases by up to $1,823\,\%$. Thus, the degree of conflict for shared resources is a crucial aspect for energy consumption analysis and optimisation.

## IV. Related Work

Improving the energy efficiency of system software has been subject of research works ranging from energy-aware synchronisation techniques [23], [24] over run-time optimisations [25], [26] to constructive measures which, for example, provide analysis tools [4], [21] that help programmers to lower the energy-demand of systems. The shared objective among the work related to the research presented in this paper are to better interlock software and hardware mechanisms with the common goal to reduce unnecessary frictional losses which are responsible for the dissipation of energy. Especially the advent of multi-core processors has introduced new opportunities and challenges to reduce the energy footprint of computing systems.

Moreshet et al. compare the energy efficiency of transactional memory with the use of traditional spin-locks [23]. The work shows that transactional memory outperforms traditional locking (i.e., spin locks) with regards to performance and energy in cases of low contention. If the degree of contention rises, however, transactional memory requires an increasing amount of transactions which occurs at the expense of energy demand. The authors conclude their work that neither of the explored approaches has been designed with a focus on energy consumption. To mitigate the negative effects (i.e., increased energy demand in the case of high contention), Kim et al. propose the hardware extension *C-Lock* [24]. *C-Lock* combines transactional memory techniques with traditional locking mechanisms and detects memory-access conflicts to shared data by analysing the addresses of memory-access operations. In case of conflict, *C-Lock* stops a subset of the competing processor cores and reduces their energy demand by clock-gating the cores for the time of the conflict. In absence of conflicts, *C-Lock* achieves a degree of parallel execution which is comparable to systems which use transactional memory, only.

To increase the energy efficiency at run-time, Nishtala et al. explore an energy-aware thread-to-core assignment policy for heterogeneous multi-core processors [25]. The proposed strategy incorporates memory and performance demands of individual threads and creates predictions about the prospective thread behaviour from empirically monitoring thread demands (i.e., by means of performance counters). On basis of the predictions, co-located threads (so-called co-runners) are being selected for future thread-to-core assignments. The assignment algorithm attempts to spread contention for shared resources uniformly across the available processor cores and aims to optimize energy efficiency. Santinelli et al. [26] explore *energy-aware scheduling* techniques for embedded systems by exploiting energy-saving features at hardware-level such as dynamic voltage and frequency scaling (DVFS) and dynamic power modes (DPM). The authors propose an on-line energy-aware scheduling algorithm which schedules task and messages under real-time constraints. Aligned to timing constraints, the on-line scheduler calculates and enforces optimal processor speeds (controlled by DVFS) and sleep durations (controlled by DPM) to minimise the energy demand of the system at run-time.

Constructive measures address the design of energy efficient system software. Program analysis approaches [4], [21] determine the energy demand of programs and help programmers to identify and resolve energy faults. In practice, such energy-aware programming methods need to be tightly coupled with tool-chains [27] in order to effectively help programmers at designing energy-efficient system software.

Our research results (cf. Section III) suggests that the design of system-software components (i.e., synchronisation methods) need to be accompanied by appropriate constructive measures. Only the combination of different optimisation approaches (i.e., system design, run-time support, and tool infrastructure) makes it possible to reduce the energy of computing systems by the highest possible extent.

## V. Conclusion

Synchronisation of interacting processes has a huge influence on the energy demand of applications. The presented empirical study showed that misconception in the use of a synchronisation method for critical sections increases the energy demand by up to $538\,\%$. Well established methods such as the Pthread mutex or a ticket lock scale poorly when regarding execution time and energy consumption. Measures for "obviation of congestion" as introduced with guarded sections using passive-mode future variables outperform ticket locks at high contention.

Emphasis of the examination in this paper was on the synchronisation of critical sections, that is, on procedures that ensure sequential execution of a specified sequence of instructions. Functional identical but structural disparate representations of the respective program sections in terms of non-blocking synchronisation were not considered. However, this topic will be addressed by future work, along with an evaluation of already existing real-world applications, including hard and firm real-time requirements. Furthermore, emerging multi-core platforms for embedded systems will not only increase the relevance of energy-aware parallel programs, but also give the opportunity to diversify our experiments. The long term goal are infrastructure operations that aim at *proactive energy-aware programming* particularly of non-sequential software.

The idea is to have *energy-aware non-sequential real-time systems* that prescribe parallel processes which are subjected to different synchronisation protocols dependent on the actual degree of contention. Thus, blocking and non-blocking synchronisation changes on the fly just as the respective method of the individual synchronisation category. As there is no panacea in the attempt of decreasing energy consumption other than providing a *family of programs* that are each specialised to a certain level of contention, issue will be to dynamically switch to the most suitable variant as a function of the actual degree of contention.

## Appendix
## Guarded Sections

At a bigger picture, in this model a process creates a closure for each "critical section" and encapsulates it in a *job*, that is, a unit of work that is scheduled and executed by a so-called *sequencer*. To request execution of that section, a process attaches the corresponding job to the guard.

To ensure sequential execution, the guard protocols guarantee that, at any time, at most one process can act as a sequencer. When a sequencer is already processing jobs, concurrent processes consequently fail to occupy the guard. These processes advance without blocking, while the sequencer executes their jobs asynchronously. In case a process depends on the computation result or state change caused by a guarded section, the sequencer provides a *promise* [28] and delivers the result of the corresponding job in the *future* [11]. In such a situation, the bypassing process commits *conditional synchronisation*: if necessary, it delays (i.e., blocks) until the sequencer indicates the availability of the promised event.

In order to give a rough picture of the "background noise" coming with guarded sections, the entry and exit protocols are briefly sketched at this point (Fig. 4). In contrast to the original [6], an improved version of these protocols is presented. The original version contains a race condition. As a potential consequence of this, a guard state can be left behind that pretends to execute pending guarded sections by some process (i.e., the sequencer) although no such process actually is in charge of it and never can be. In Fig. 4, the race condition is resolved by recording each attempt to enter a guarded section.

```
function VOUCH($guard, order$)
    ENQUEUE($guard.list, order$)
    if FAA($guard.load, 1) = 0$ then        ▷ become sequencer
        return DEQUEUE($guard.list$)
    else
        return 0                            ▷ indicate bypassing
    end if
end function
function CLEAR($guard$)
    if FAA($guard.load, -1) > 1$ then       ▷ remain sequencer
        return DEQUEUE($guard.list$)
    else
        return 0                            ▷ indicate leaving
    end if
end function
```

**Fig. 4:** Guarded section entry and exit protocol.

The entry protocol (VOUCH) authorises a process to become the sequencer of particular guarded section (*guard*). This sequencer is the single thread of control at a given point in time that is allowed to execute pending requests (*order*) to pass through such a section. The exit protocol (CLEAR), on the other hand, defers to that thread of control the execution of the next passing request, if any. This very thread then decides

```
procedure CRITICAL_MX($entity, input$)
    ACQUIRE($entity.mutex$)
    /* needed sequential execution of
       some task that must process
       some $input$ controlled by $entity$ */
    RELEASE($entity.mutex$)
end procedure
```

```
procedure CRITICAL_GS($entity, input$)
    $request \leftarrow$ ORDER($input$)
    if $job \leftarrow$ VOUCH($entity.guard, request$) then
        repeat
            /* needed sequential execution of
               some task that must process
               some $input$ controlled by $entity$ */
        until ($job \leftarrow$ CLEAR($entity.guard$)) $= 0$
    end if
end procedure
```

**Fig. 5:** Sequential execution using plain critical sections (left) and guarded sections (right).

in a self-contained manner and depending on its own potential to either process the request (if there is enough slack time) or hand it over to a spare sequencer (otherwise).

Pending passing requests are kept on a wait-free synchronised queue. In contrast to [6], which relies on a queue with a restricted utilisation profile, the implementation used in the experiments is based on an all-purpose wait-free queue [29] for an even-handed comparison with the Pthread mutex and ticket lock variants.

For comparison with conventional critical sections, assume some operation with reference to a complex control or information structure, referred to as $entity$, as shown left in Fig. 5. The usual "synchronisation brackets" are represented by ACQUIRE and RELEASE, namely to make a request for and to quit mutual exclusion.

The actual $entity$ parameter of this operation identifies the mutex or lock, respectively, variable used in the entry and exit protocols. In order to restrict potential parallelism as short as possible, synchronisation variables are used on a per-entity basis. The "bracketed" statements left in Fig. 5 surround the actual functional part within the critical operation. In case of contention at the critical section, that is, while the section is occupied by some process (holder), any contending process will be blocked upon request (ACQUIRE) to enter.

Synchronisation of the same functional code but, this time, using guarded sections is shown right in Fig. 5. Here, the "synchronisation brackets" are implemented using the language constructs **if**, **repeat**, and **until** for the *control function* and ORDER, VOUCH, and CLEAR for *order management*. The job for a guarded section is created using ORDER. By means of VOUCH, a process (i) registers a job and (ii) asks to occupy the guard, that is, operate the sequencer. If VOUCH succeeds, the current process becomes the sequencer and takes care for the successive execution of pending jobs. When a sequencer is already associated with the guard, VOUCH fails and the current process bypasses the guarded section. The bypassing process is also called *requestor*. In case of contention at the guarded section, that is, while the section is occupied by some process (sequencer), any contending process (requestor) will be never blocked upon request (VOUCH) to enter.

At end of a guarded section, the sequencer checks for pending jobs by using CLEAR. If there is at least one more job outstanding, CLEAR succeeds and the sequencer repeats itself, that is, loops through the guarded section using another parameter setting. The guarded section remains occupied until CLEAR fails, which indicates that either no more jobs are awaiting execution on the guard or the maximum allowable load for the particular sequencer is exhausted. In this case, the sequencer marks the guard as unoccupied and continues its control flow, that is, relinquishes sequencer functionality, or hands over further job processing to some other sequencer, respectively. The actual variant [7] depends on the *guard-family member* used.

An alternative setting of a guarded section, which then leads to a process behavior that is more strict to a conventional critical section, is shown in Fig. 6. This pattern uses the

```
procedure CRITICAL_GS_STRICT($entity, input$)
    $request \leftarrow$ ORDER($input$)
    if $job \leftarrow$ VOUCH($entity.guard, request$) then
        repeat
            /* needed sequential execution of
               some task that must process
               some $input$ controlled by $entity$ */
            PROVE($job.tobe$)                      ▷ cause signal
        until ($job \leftarrow$ CLEAR($entity.guard$)) $= 0$
    else
        EXACT($request.tobe$)                      ▷ await signal
    end if
end procedure
```

**Fig. 6:** Sequential execution using signalling guarded sections.

primitives PROVE and EXACT for logical synchronisation of the requestor with the sequencer. By calling EXACT, the requestor awaits completion of its order to be executed by the sequencer. The latter applies PROVE inside the guarded section to signal completion of the order. These two primitives manage the "future variable" ($tobe$) that communicates computation results from the sequencer to the requestor. The sequencer never blocks in the course of PROVE execution, that is, while assigning a value to $tobe$. Contrariwise, EXACT delays the requestor if and only if the future variable has not yet been assigned a value from the sequencer. The requestor thus may ensure that problem-specific dependencies to the guarded section are fulfilled. In case of contention at the guarded section, that is, while the section is occupied by some process (sequencer), any contending process (requestor) will never be blocked upon request (VOUCH) to enter, but can be blocked on retrieval (EXACT) of the processing status of an order.

Essentially, multilateral synchronisation (mutual exclusion) of concurrent processes at a conventional critical section is mapped to unilateral (conditional) synchronisation of a (data, causally) dependent process that proceeds to some extent parallel to a guarded section. In contrast to mutual exclusion, process delay becomes effective only if the data-delivery event on which the respective process depends is still pending, even in case of contention for running a guarded section.

As a matter of fact, the bypassing process (requestor) of a guarded section gets the opportunity for *latency hiding*, that is, doing meaningful work over the time needed by the sequencer to produce a value and, as the case may be, never blocks in the attempt to consume this very value. In contrast, when using mutual exclusion to synchronise critical sections, contending processes do not have such an option. Then, processes either actively or passively exercise useless work while awaiting admission, depending on the entry protocol (i.e., ACQUIRE). If the entry protocol of a conventionally synchronised critical section triggers a process switch when a waiting condition has been detected, then the "system" indeed may proceed in doing meaningful work, but not the individual process that competes with fellow sufferers for access to that section. All this never happens with a guarded section.

## Acknowledgements

This work is supported by the German Research Foundation (DFG) under grants no. SCHR 603/8-2, SCHR 603/13-1, 603/15-1, and the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR89, Project C1).

## References

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94–104, September 1991.

[2] D. Tennenhouse, "Proactive computing," *Communications of the ACM*, vol. 43, no. 5, pp. 43–50, May 2000.

[3] X. Liao, L. Xiao, C. Yang, and Y. Lu, "MilkyWay-2 supercomputer: System and application," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 345–356, June 2014.

[4] P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat, "Worst-case energy consumption analysis for energy-constrained embedded systems," in *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS '15)*. IEEE, 2015, pp. 105–114.

[5] H. Esmaeilzadeh, E. R. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA '11)*, R. Iyer, Q. Yang, and A. González, Eds. ACM, 2011, pp. 365–376.

[6] G. Drescher and W. Schröder-Preikschat, "Guarded sections: Structuring aid for wait-free synchronisation," in *Proceedings of the 18th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '15)*. IEEE, 2015, pp. 280–283.

[7] D. L. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, pp. 1–9, March 1976.

[8] E. W. Dijkstra, "Cooperating sequential processes," http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, Tech. Rep. EWD-123, 1965.

[9] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, February 1991.

[10] B. Lewis and D. J. Berg, *PThreads Primer: A Guide to Multithreaded Programming*. SunSoft Press, 1996.

[11] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. ACM, 1977, pp. 55–59.

[12] M. Merritt and G. Taubenfeld, "Speeding Lamport's fast mutual exlusion algorithm," *Information Processing Letters*, vol. 45, pp. 137–142, March 1993.

[13] M. M. Michael and M. L. Scott, "Fast mutual exclusion, even with contention," University of Rochester, Computer Science, Tech. Rep. 460, June 1993.

[14] L. Lamport, "A fast mutual exclusion algorithm," *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 1–11, May 1987.

[15] M. P. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 123–149, January 1991.

[16] *Intel Architecture Software Developer's Manual*, Intel Corporation, Santa Clara, California, USA, 2016.

[17] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*. USENIX, 2012, pp. 65–76.

[18] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the 8th Conference on Operating Systems Design and Implementation (OSDI '08)*. USENIX, 2008, pp. 43–57.

[19] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *Proceedings of the 1994 Symposium Low Power Electronics*. IEEE, 1994, pp. 8–11.

[20] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *IEEE Computer*, vol. 36, no. 12, pp. 68–75, December 2003.

[21] T. Hönig, C. Eibel, R. Kapitza, and W. Schröder-Preikschat, "SEEP: Exploiting symbolic execution for energy-aware programming," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 3, pp. 58–62, December 2011.

[22] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: A platform for OS-level power management," in *Proceedings of the 4th European Conference on Computer Systems (EuroSys '09)*. ACM, 2009, pp. 289–302.

[23] T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy-aware microprocessor synchronization: Transactional memory vs. locks," in *Proceedings of the 4th Workshop on Memory Performance Issues (WMPI '06)*. IEEE, 2006, pp. 1–7.

[24] S. H. Kim, S. H. Lee, M. Jun, B. Lee, W. W. Ro, E.-Y. Chung, and J.-L. Gaudiot, "Energy efficient synchronization for embedded multicore systems," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 1962–1974, August 2014.

[25] R. Nishtala, D. Mossé, and V. Petrucci, "Energy-aware thread co-location in heterogeneous multicore processors," in *Proceedings of the 11th International Conference on Embedded Software (EMSOFT '13)*. ACM/IEEE, 2013, pp. 1–9.

[26] L. Santinelli, M. Marinoni, F. Prosperi, F. Esposito, G. Franchino, and G. Buttazzo, "Energy-aware packet and task co-scheduling for embedded systems," in *Proceedings of the 10th International Conference on Embedded Software (EMSOFT '10)*. ACM/IEEE, 2010, pp. 279–288.

[27] T. Hönig, H. Janker, O. Mihelic, C. Eibel, R. Kapitza, and W. Schröder-Preikschat, "Proactive energy-aware programming with PEEK," in *Proceedings of the 2014 Conference on Timely Results in Operating Systems (TRIOS '14)*. USENIX, 2014, pp. 1–14.

[28] D. P. Friedman and D. S. Wise, "The impact of applicative programming on multiprocessing," in *Proceedings of the International Conference on Parallel Processing (ICPP '76)*. IEEE, 1976, pp. 263–272.

[29] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 2011 Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, 2011, pp. 223–233.