EARL: Energy-Aware Reconfigurable Locks

Stefan Reif, Phillip Raffeck, Heiko Janker, Luis Gerhorst,
Timo Hönig, and Wolfgang Schröder-Preikschat
{reif,raffeck,janker,gerhorst,thoenig,wosch}@cs.fau.de
Friedrich-Alexander University Erlangen-Nürnberg

ABSTRACT

As system complexity grows, embedded operating systems increasingly face the challenge to adhere to various non-functional constraints, such as response times and power limits. These requirements sometimes contradict and, often, no solution satisfies all constraints under all conditions. Changes in environmental conditions, application-level requirements, and user response time expectation hence demand for system-wide adaptions to resource management. We find that process synchronization constitutes a simple yet effective leverage point to balance between timing-related and energy-related constraints. This paper presents EARL, an implementation of reconfigurable locks in Linux that enables seamless transitions between high-performance and low-power operating modes.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Hardware** → **Power estimation and optimization**; • **Software and its engineering** → Software organization and properties.

KEYWORDS

Energy-aware systems, Dynamic reconfiguration, Mutual exclusion

ACM Reference Format:
Stefan Reif, Phillip Raffeck, Heiko Janker, Luis Gerhorst, Timo Hönig, and Wolfgang Schröder-Preikschat. 2019. EARL: Energy-Aware Reconfigurable Locks. In *EWiLi 2019 – The Embedded Operating Systems Workshop, October 17, 2019, New York, USA.*. ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

With breathtaking speed, multiprocessor systems-on-chips (MPSoCs) are pervading many different fields of application. For example, large-scale Internet-of-Things (IoT) systems [14] employ MPSoCs just like cyber-physical systems [16, 30] do. With increasing cost pressure, MPSoCs additionally become state of the art in the embedded domain, such as the automation and automotive industry [5, 28]. Configurations with 10^3 and more processors on a chip arise on the horizon [4]. These devices will be heterogeneous in terms of on-chip processors, communication facilities, and memory organization. Shared and distributed memory will coexist on a single chip. At a certain level, cache coherence is no longer implemented in hardware analog to Intel's single-chip cloud computer.

As to different applications and operational domains, embedded systems are exposed to varying requirements and therefore must operate under a multiplicity of different constraints. For example, real-time constraints must be adhered to while considering additional non-functional system properties such as power[1] demand [27]. As system constraints are often dynamic, the non-functional properties of the systems must be adapted at run-time. This puts the system software into the spotlight of dynamic resource management for MPSoCs: available system resources (e.g., processor cores, memory) must be allocated dynamically, and corresponding hardware devices must be configured efficiently (e.g., DVFS, sleep states, heterogeneity [20, 31]). Hence, the enablement of massively-parallel applications that efficiently utilize MPSoCs becomes a non-trivial challenge to the system software.

A key problem in those applications is *contention* of interacting processes, that is, simultaneous processes that (directly or indirectly) interact with each other through a shared variable or by accessing a shared resource. The number of contentious processes is application-dependent and has a major impact on the effectiveness of their coordination at all levels of a computing system. Overhead, scalability and degree of specialization of the synchronization functions are decisive performance-influencing factors. This influencing variable not only causes varying process behavior, but also different power demands. The former leads to noise or jitter in the program flow: non-functional system properties, which are particularly problematic for highly parallel or real-time-dependent processes. The latter has, on the one hand, economic weight (higher costs for the power supply and cooling) as well as ecological effects (greater environmental impact due to increased power demand for the computing system) and, on the other hand, affects the limits in the scaling of multi-core processors: *dark silicon* [11]. These limits are ultimately set by measures to prevent possible overheating or even "meltdown" of the processor.

To solve this problem, various options are under discussion [32]: (1) shrinkage of the semiconductor chip, (2) down-regulation of circuit areas (*dim silicon*), (3) use of otherwise idle circuit areas for specialization by coprocessors and (4) the "Deus ex Machina" in the form of an entirely new semiconductor technology. Except for (1) and (4), the other options also require software measures, especially in operating systems. With regard to (2), for example, speed control of processes [34] or their "evacuation" to a colder processor core [25]. The extreme case of circuit area down-regulation is the deactivation of entire areas of processor cores, in order to leave them lying idle and thus to create "channels" for a better heat transfer. With it, processor allocation techniques and partitioning methods known from very large HPC systems [21] make their way into operating systems for high-scale MPSoCs.

EWiLi '19, October 17, 2019, New York, USA

[1] Throughout this paper we use the terms *power* and *energy* (i.e., power over time) according to the respective context.

Contention creates hot spots, in the true sense of the word. That is how the paper focuses on the energy-aware synchronization of interacting processes. Starting point of the approach is a series of measurements of the performance and power characteristics of locking parameters. In a second step, the obtained information is applied in a run-time system for *energy-aware reconfigurable locks* (EARL) that dynamically adapts to the current operating conditions and requirements (e.g., degree of contention, amount of available resources, or required responsiveness). Contribution is (1) a concept of dynamic lock reconfiguration for energy awareness, (2) a practical implementation of this concept in Linux, and (3) an evaluation of this implementation on Raspberry Pi 4 based on actual energy measurements to demonstrate trade-offs between performance and power demand and to motivate the design decisions.

The rest of the paper is organized as follows. Section 2 gives further background information and discusses related works. The concept of reconfigurable locks is explained in Section 3, followed by remarks on the opposing dependency of performance and power demand in Section 4. An evaluation of the prototype implementation is presented in Section 5, and Section 6 concludes.

2 BACKGROUND AND RELATED WORK

Process synchronization has been a research topic for more than half of a century [10]. However, most research focused on large-scale computers rather than embedded systems, since they adopted the principles of parallel processing on multi-core hardware much quicker. In consequence, a plethora of algorithms for mutual exclusion has been designed for server-class machines [8, 18, 19, 24, 26], rather than embedded systems.

Falsafi et al. [12] propose "MutexEE", an energy-efficient synchronization algorithm. While this paper is inspired by their findings, they focus on large-scale server platforms rather than embedded systems, with differences in system scale (regarding both power demand and performance [13]), instruction-set architecture, and benchmark applications.

To allow for mutual exclusion algorithms in hard real-time systems, research has worked towards analyzing blocking bounds [2, 3, 33, 35]. Most of this research is oblivious to actual lock implementations, except that they typically assume *first-in first-out* (FIFO) lock provisioning. However, FIFO interacts poorly with passive waiting, which is often energy efficient [29]. Hard real-time systems are usually tailored to guarantee latency constraints, rather than energy efficiency. This paper, therefore, focuses on soft real-time systems, which use operating systems like Linux [23].

To allow for both energy-efficient and high-performance operating modes, we propose system-wide lock reconfiguration. Typically, lock reconfiguration is performed on lock granularity or process granularity (e.g. [1]) to optimize for performance. As an example, `glibc` [17] provides "adaptive" mutexes that spin in user-space for a flexible amount of time to avoid the performance overhead of system calls. However, we argue that energy-saving behavior adaptions should be system-wide and mandatory. Otherwise, adaption-aware applications face a performance penalty while unaware or non-cooperating processes are not affected by a performance degradation. As an approach that is similar to EARL, `glibc` provides "tunables": These options allow fine-tuning of internal

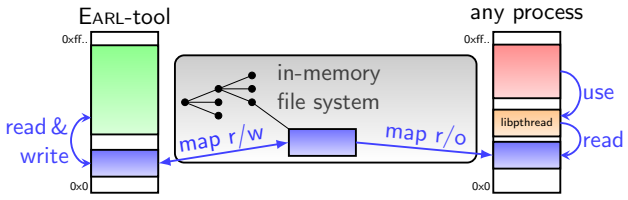


Figure 1: Overview of the memory mapping mechanism for dynamic reconfiguration via a shared configuration file.

`glibc` settings at process creation time via environment variables [6]. For tunables, modification of the behavior at run-time is not yet supported.

3 LOCK RECONFIGURATION

To enable embedded real-time systems to react and adapt to environmental changes and their effect on application constraints, we propose lock reconfiguration that works system-wide and at run-time. First, performing reconfiguration at system level comes with several advantages, compared to application-level approaches. The most important benefit is the simultaneous effect on every application (i.e., "system-wide"), forcing applications to comply with the current system-wide power/performance settings. Otherwise, non-cooperative applications could harm the system by ignoring power-saving settings. Additionally, the reconfiguration mechanism in EARL is completely transparent to the applications in functional terms, as there is no need for changes in the system API. This obviates the need for the integration of specific adaption mechanisms in applications. Second, reconfiguration at run-time (i.e., "dynamic") removes the need to re-start applications, which is crucial for embedded systems with real-time constraints.

For dynamic reconfiguration, EARL uses the existing memory-mapping functionality in Linux. Thereby, configuration values are stored in files, which are located in an in-memory file system. As Figure 1 depicts, a configuration file is mapped as read-only into the address space of every process that uses the `pthreads` library. Hence, configuration parameters are easily accessible with minimal overhead.

The use of shared memory ensures that reconfiguration happens on the fly without restarting applications. In particular, real-time systems benefit from the negligible overhead and minimal, predictable interference of reconfiguration via shared-memory communications compared to, for example, signal-based mechanisms.

A dedicated configuration tool (`EARL-tool`) facilitates the reconfiguration process by adapting lock parameters at run-time and also provides means to display the configuration state. Thereby, all read and write accesses to the configuration variables happen atomically. The `EARL-tool` thus ensures that invalid intermediate configuration states cannot exist, which is an important invariant because it eliminates the need for explicit inter-process synchronization for the reconfiguration.

Our approach is extensible to multiple different configurations by allowing applications to specify an EARL *domain*, short "earldom" at process creation. Internally, each domain maps to an individual configuration file, so that parameter changes in one domain

Parameter Name	Range	Description
`retry_adaptive`	`bool`	force adaptive locking
`retry_max`	`uint`	lock-retry limit
`unlock_unfair`	`bool`	enable unfair unlocking mode
`unlock_iterations`	`uint`	delay for unfair-unlock

Table 1: Overview of the EARL configuration parameters.

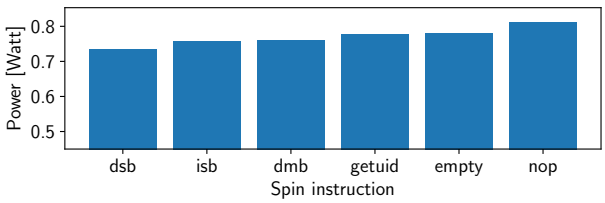


Figure 2: Average power demand of spinning instructions

do not affect other domains. The separation into different earldoms enables the representation of specific application requirements in their lock configurations. This enables distinct criticality levels, for example, for system-level background daemons without real-time requirements.

Inspired by the promising results of the "MUTEXEE" concept of Falsafi et al. [12] for Intel server architectures, we bring their approach to the domain of embedded systems. We integrate EARL into `nptl`, the `libpthreads` implementation within `glibc`, and extend the existing mutex implementation in a similar fashion to the MUTEXEE approach. Table 1 summarizes the configuration options provided by EARL. All type ranges listed for the different parameters refer to the atomic versions of these types defined by the C11 standard. Most importantly, we introduce an unfair unlocking mode and an adaptive locking mode.

First, the `unlock_unfair` parameter enables an unfair mode. In this mode, the unlocking thread spins for a configurable amount of time right after marking a mutex as free, allowing another thread to immediately reclaim the lock, before issuing a system call to wake up sleeping threads. As the expectation is that another thread instantaneously claims the mutex, the costly wake-up system call can be omitted. According to [12], this strategy improves the energy efficiency. In EARL, the duration of the delay is configured via the `unlock_iterations` parameter.

Second, the `retry_max` parameter allows threads to repeatedly try to acquire a mutex in user-space, before issuing a system call for sleeping. If critical sections are short, this strategy avoids costly system calls for both sleeping and waking up, and mutex ownership can be transferred much faster.

Third, the `retry_adaptive` parameter enforces the adaptive[2] mode already present in `nptl`. In the adaptive mode, the number of retries is continuously adjusted using heuristics, within the limit provided by the `retry_max` parameter. In contrast, if adaptive is set to `false`, the heuristics are disabled, and the number of retry operations is set to the value of the `retry_max` parameter (where the value zero is equivalent to the default `glibc` behavior). Contrary to the `glibc` implementation, enabling and disabling of the adaptive mode and modifying the retry limit are possible at run-time.

We decided to only adjust internal locking parameters, in contrast to exchanging locking algorithms, to ensure that all variants are compatible. The reason is that, if the algorithm changes, a large number of intermediate states can co-exist, which makes it difficult for the `EARL-tool` to guarantee the absence of invalid states. As an example, the `glibc` mutex implementation has a dedicated state representing a mutex that is taken but certainly no waiters exist, to safely avoid wake-up system calls under *some* specific conditions. However, the unfair mode does not work with this optimization because it breaks internal assumptions. This implies that state transitions of the `unlock_unfair` parameter have the potential of lost-wakeup situations. We solved this issue by enforcing a wake-up system call in the next call to `pthread_mutex_unlock()` whenever `unlock_unfair` changes.

Our current implementation of EARL operates under the assumption that applications share locks only between threads of equal priority. Otherwise, a priority inversion scenario [22] could arise, where a higher priority thread is forced to wait for the release of a lock by a lower priority thread. Common techniques dealing with priority inversion are, however, orthogonal to the lock-behavior modifications of EARL. In particular, additional spinning of the higher priority thread in said priority inversion scenario could potentially even aggravate the problem. Only allowing the sharing of locks between threads of equal priority bypasses these difficulties and obviates the need to take thread priorities into account during lock acquisition. Extending EARL to beneficially interoperate with priority-inversion mechanisms is considered future work.

4 TRADE-OFFS

When a lock is taken, threads can generally either use system calls for sleeping, or spin in user-space. While the former comes with the overhead of system calls, which has costs in terms of latency and energy; the latter tends to be inefficient because it cannot utilize sleep states. This means that, typically, a combination of both is optimal. For long waiting times, sleeping is more efficient due to low-power sleep states, but for short waiting times, polling is more efficient since it avoids the system-call overhead.

Since such trade-offs have no trivial solution, we motivate design decisions and trade-offs in EARL using measurements. The measurement setup is described in Section 5.1.

4.1 Spinning Method

To minimize the power demand during waiting times, we compare multiple spinning methods. The goal is to spend time but minimize the power demand. For this, we measure the average power demand of a micro-benchmark that spins in a tight loop. The spinning methods we compare are a `nop` instruction, memory barriers (`dsb`, `isb`, `dmb`), no instruction at all (`empty`), and the `getuid` system call. The results, summarized in Figure 2, show that the `dsb`

[2] mutex initialized with `PTHREAD_MUTEX_ADAPTIVE_NP`

memory barrier minimizes the power demand. We, therefore, use this instruction in Earl when spinning.

4.2 User-Space Lock Acquisition Retries

To measure the influence of Earl parameters, we develop micro-benchmarks where we can precisely control the degree of contention. These micro-benchmarks alternatingly execute critical and non-critical sections of parameterizable length. Configurations with shorter non-critical sections thereby lead to higher lock contention. By comparing executions with different parameter sets, we are thus able to examine the impact of Earl on applications featuring different degrees of contention.

We vary the `retry_adaptive` and `retry_max` parameters of Earl, fixing `unlock_unfair` to `false`. We evaluate the throughput-per-power (TPP) metric, which calculates a trade-off between performance and power demand. In the micro-benchmark, the throughput is the number of successful lock acquisitions per second, the TPP thus the number of lock acquisitions per joule. Figure 4 visualizes the results. In particular, the optimal number of user-space lock acquisition retries strongly depends on the degree of contention. Thereby, the adaptive configuration which uses heuristics to estimate the optimum number of retries performs slightly worse than the non-adaptive configuration. This might be an artifact of the micro-benchmark where the length of critical sections is fixed. It is nevertheless possible that in real-world applications with varying critical section lengths the adaptive lock version performs better.

4.3 Unfair Unlock

We use the same micro-benchmarks as in Section 4.2 to evaluate the influence of unfair unlocking. We fix the `retry_max` parameter to `800` and `retry_adaptive` to `false`. The results, summarized in Figure 5, show that in most scenarios, the unlock-unfair strategy is more power-efficient than the default variant.

For applications with (soft) real-time constraints, unfairness is generally problematic because it can imply severe synchronization delays which harm response times. For applications without real-time constraints, in contrast, the unfair variant can improve energy efficiency. This power efficiency increase strengthens our motivations for earldoms, since they allow for a co-existence of both fair and unfair configurations.

5 EMPIRICAL EVALUATION

5.1 Measurement Setup

An overview of the measurement setup is outlined in Figure 3. The device under test (DUT) is a Raspberry Pi 4 Model B [15] with 4GB RAM running the Yocto Linux Reference Distribution (Poky) [7]. The DUT was modified by lifting the inductance in the core voltage rail and inserting a shunt resistor. The resulting voltage drop over the shunt is measured by an LTC2991 [9] monitoring IC featuring an 14-Bit ADC. The energy usage is calculated and forwarded to the controlling Linux PC via an AVR microcontroller. To ensure repeatable results without interference from thermal throttling, the operating frequency of the DUT was limited to 600 MHz.

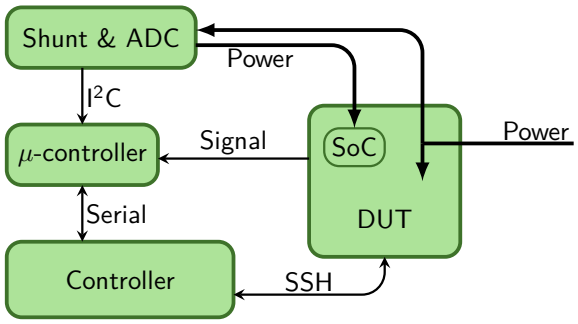


Figure 3: Overview of the measurement setup

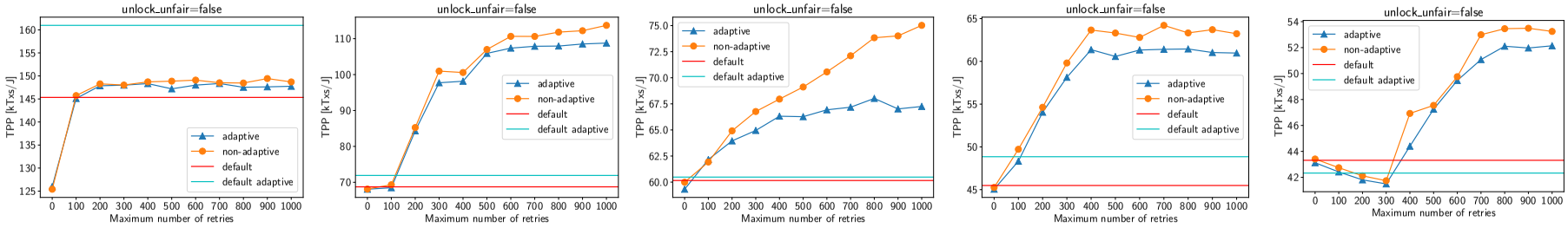
5.2 Measurement Results

We use the "MUTEX" benchmarks of the synchrobench [18] suite in the `C/C++` version to evaluate the power efficiency of Earl. In particular, these benchmarks are implementations of data structures of different complexity (hashtable, hoh-list, lazy-list, RCU-tree, and skiplist) that use `pthread` mutexes for synchronization. These generic data structure benchmarks are representative examples of real-world workloads in embedded multi-core systems. We measure the power and throughput of these benchmarks with different parameter sets. We use the benchmark-supplied throughput metric (i.e., number of transactions per second) and measure power demand. All benchmarks execute with standard (i.e., fair) or unfair unlocking with 6 iterations before the wake-up call is performed. We also evaluate both adaptive and non-adaptive locking with `retry_max` ranging from 0 to 1000 retries.

Figure 6 summarizes the results of these measurements starting each benchmark with four threads (i.e., one thread per core), depicting the performance achieved at a specific power consumption using different `retry_max` values with and without adaption heuristics. Experiments with different numbers of threads yielded comparable results, which we do not discuss here for space reasons. In general, we see different patterns for each benchmark, underlining the dependency of the achieved results on the contention behavior of the specific applications. Overall, the unfair unlocking protocol decreases the throughput significantly, but it also improves the power demand, for three of the five benchmarks. The results mean that the `unfair_unlock` parameter enables transitions between low-power and high-performance operating modes.

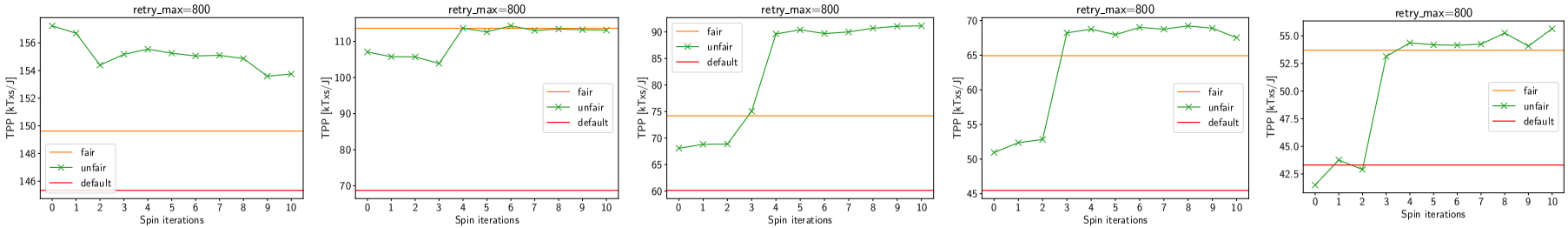
Both the adaptive and non-adaptive approach show similar results with no indication of one approach being absolutely better. This matches the behavior of the micro-benchmark results presented in Section 4. For most benchmarks, the `retry_max` parameter has only a minor influence on the benchmark performance, but it often can improve the power demand.

In contrast to the results obtained by Falsafi et al. [12], our experiments show no clear indication for a universally better approach. In particular, we were not able to confirm their conjecture that both throughput and energy efficiency can be simultaneously improved, in the domain of embedded systems. Since no parameter set is the universally best configuration, the generation of application-specific profiles that predict the power and performance characteristics could be helpful in embedded systems.



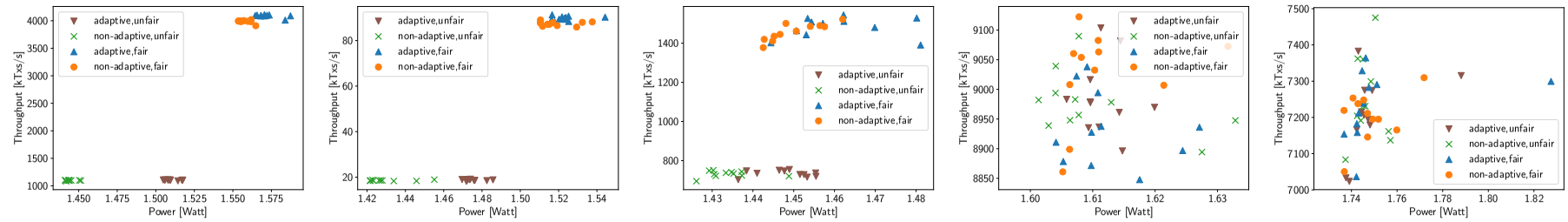
(a) very low (b) low (c) medium (d) high (e) very high

Figure 4: Power efficiency depending on contention and user-space lock acquisition retries



(a) very low (b) low (c) medium (d) high (e) very high

Figure 5: Power efficiency depending on contention and unlock unfairness



(a) hashtable (b) hoh-list (c) lazy-list (d) RCU-tree (e) skiplist

Figure 6: Power/Performance evaluation of synchrobench benchmarks

6 CONCLUSION

In this paper, we have presented Earl, an approach for dynamically reconfigurable and energy-aware locks. We have demonstrated that, for locking, no configuration is the universally best. In particular, we were not able to reproduce the results presented by Falsafi et al. for server platforms [12] in the domain of embedded systems. Instead, Earl provides a run-time system that allows the reconfiguration of locking parameters at system level without restarting processes or adapting applications. The evaluation shows that, by using the Earl parameters, applications can switch seamlessly between high-performance and low-power modes. While the results yield no universally beneficial approach, Earl facilitates such mode-configuration switches for different application classes.

Future work will extend Earl in various ways. First, an integration into battery or temperature monitoring frameworks can reconfigure the system dynamically depending on the amount of available energy or temperature headroom. Second, Earl can interact with application profiles that model the power and performance characteristics of specific applications. Our evaluation has shown that applications react differently to configuration changes. Since Earl allows for dynamic reconfiguration, such profiles can be generated on-the-fly while the application is running. Third, future versions of Earl can support the dynamic replacement of the underlying locking algorithm. Additionally, investigating in-kernel throughput optimizations, as opposed to the `glibc`-based approach of Earl, can reveal potential synergies. Lastly, Earl can be extended to achieve the aforementioned interoperation with priority-inversion avoidance techniques, for example, by using the Earl extensions only if the locking threads priority allows it and falling back to the default `glibc` mechanisms otherwise.

7 ACKNOWLEDGEMENT

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 146371743 – TRR 89 "Invasive Computing" as well as the individual research grants SCHR 603/10-2, /13-1, and /15-1.

REFERENCES

[1] Jelena Antić, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. 2016. Locking Made Easy. In *Proceedings of the 17th International Middleware Conference (Middleware 2016)*. ACM, Article 20, 14 pages.

[2] Alessandro Biondi, Björn Brandenburg, and Alexander Wieder. 2016. A Blocking Bound for Nested FIFO Spin Locks. In *Proceedings of the 37th Real-Time Systems Symposium (RTSS 2016)*. IEEE, 291–302.

[3] Aaron Block, Björn Brandenburg, James Anderson, and Stephen Quint. 2008. An Adaptive Framework for Multiprocessor Real-Time System. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS 2008)*. IEEE, 23–33.

[4] Shekhar Borkar. 2007. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC 2007)*. ACM, 746–749.

[5] Aaron Carroll and Gernot Heiser. 2014. Mobile Multicores: Use Them or Waste Them. *ACM SIGOPS Operating Systems Review* 48, 1 (May 2014), 44–48.

[6] The GLIBC community. 2019. [GLIBC] Tunables. https://www.gnu.org/software/libc/manual/html_node/Tunables.html. Acc. 2019-08-01.

[7] Yocto Project Community. 2019. Poky. https://www.yoctoproject.org/software-item/poky/. Acc. 2019-08-01.

[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating System Principles (SOSP 2013)*. ACM, 33–48.

[9] Analog Devices. 2019. LTC2991. https://www.analog.com/en/products/ltc2991.html/. Acc. 2019-08-01.

[10] Edsger Wybe Dijkstra. 1968. The Structure of the "THE"-Multiprogramming System. *Commun. ACM* 11, 5 (May 1968), 341–346.

[11] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA 2011)*, Ravi Iyer, Qing Yang, and Antonio González (Eds.). ACM, 365–376.

[12] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. 2016. Unlocking Energy. In *Proceedings of the USENIX Annual Technical Conference (ATC 2016)*. USENIX, 393–406.

[13] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Barroso. 2007. Power Provisioning for a Warehouse-sized Computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA 2007)*. ACM, 13–23.

[14] Gerhard P. Fettweis. 2016. 5G and the future of IoT. In *Proceedings of the 42nd European Solid-State Circuits Conference (ESSCIRC 2016)*. IEEE, 21–24.

[15] Raspberry Pi Foundation. 2019. Raspberry Pi 4. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/. Acc. 2019-08-01.

[16] David Geer. 2005. Chip Makers Turn to Multicore Processors. *IEEE Computer* 38, 5 (May 2005), 11–13.

[17] The glibc community. 2019. The GNU C Library (glibc). https://www.gnu.org/software/libc/. Acc. 2019-08-01.

[18] Vincent Gramoli. 2015. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, 1–10.

[19] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2016. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the USENIX Annual Technical Conference (ATC 2016)*. USENIX, 649–662.

[20] Marcus Hähnel and Hermann Härtig. 2014. Heterogeneity by the Numbers: A Study of the ODROID XU+E big.LITTLE Platform. In *Proceedings of the 6th Workshop on Power-Aware Computing and Systems (HotPower 2014)*. 1–6.

[21] Fulya Kaplan, Jie Meng, and Ayse Kivilcim Coskun. 2013. Optimizing Communication and Cooling Costs in HPC Data Centers via Intelligent Job Allocation. In *Proceedings of the 4th International Green Computing Conference (IGCC'13)*. IEEE Computer Society, 1–10. https://doi.org/10.1109/IGCC.2013.6604521

[22] Butler Lampson and David Redell. 1980. Experience with Processes and Monitors in MESA. *Commun. ACM* 23, 2 (1980), 105–117.

[23] Hannu Leppinen. 2017. Current Use of Linux in Spacecraft Flight Software. *IEEE Aerospace and Electronic Systems Magazine* 32, 10 (2017), 4–13.

[24] John Mellor-Crummey and Michael Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *Transactions on Computer Systems (TOCS)* 9, 1 (Feb. 1991), 21–65.

[25] Andreas Merkel and Frank Bellosa. 2006. Balancing Power Consumption in Multiprocessor Systems. In *Proceedings of the First EuroSys Conference (EuroSys'06)*, Yolande Berbers and Willy Zwaenepoel (Eds.). ACM, New York, NY, USA, 403–414.

[26] Adam Morrison. 2016. Scaling Synchronization in Multicore Programs. *ACM Queue* 14, 4 (Aug. 2016), 56–79.

[27] Trevor Mudge. 2001. Power: A First-Class Architectural Design Constraint. *IEEE Computer* 34, 4 (April 2001), 52–58.

[28] Bikash Poudel, Naresh Kumar Giri, and Arslan Munir. 2017. Design and comparative evaluation of GPGPU- and FPGA-based MPSoC ECU architectures for secure, dependable, and real-time automotive CPS. In *Proceedings of the 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2017)*. IEEE, 29–36.

[29] Stefan Reif, Timo Hönig, and Wolfgang Schröder-Preikschat. 2017. In the Heat of Conflict: On the Synchronisation of Critical Sections. In *Proceedings of the 20th International Symposium on Real-Time Distributed Computing (ISORC 2017)*. IEEE, 42–51.

[30] Santanu Sarma, Nikil Dutt, Puneet Gupta, Alexandru Nicolau, and Nalini Venkatasubramanian. 2014. On-chip self-awareness using Cyberphysical-Systems-on-Chip (CPSoC). In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES 2014)*. ACM, 1–3.

[31] Claudio Scordino, Luca Abeni, and Juri Lelli. 2018. Energy-aware Real-time Scheduling in the Linux Kernel. In *Proceedings of the 33rd Annual Symposium on Applied Computing (SAC 2018)*. ACM, 601–608.

[32] Michael B. Taylor. 2013. A Landscape of the New Dark Silicon Design Regime. *IEEE Micro* 33, 5 (Sept./Oct. 2013), 8–19.

[33] Bryan Ward and James Anderson. 2012. Supporting Nested Locking in Multiprocessor Real-Time Systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*. 223–232.

[34] Andreas Weissel and Frank Bellosa. 2002. Event-Driven Clock Scaling for Dynamic Power Management. In *Proceedings of the 2002 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002)*, Shuvra S. Bhattacharyya, Trevor N. Mudge, Wayne Wolf, and Ahmed Amine Jerraya (Eds.). ACM, 238–246.

[35] Maolin Yang, Alexander Wieder, and Björn Brandenburg. 2015. Global Real-Time Semaphore Protocols: A Survey, Unified Analysis, and Comparison. In *Proceedings of the 36th Real-Time Systems Symposium (RTSS 2015)*. IEEE, 1–12.