Hardware-Accelerated Energy-Efficient Synchronization and Communication for Ultra-Low-Power Tightly Coupled Clusters

Florian Glaser*, Germain Haugou*, Davide Rossi†, Qiuting Huang* and Luca Benini*†
*Integrated Systems Laboratory, ETH Zürich, Switzerland, {glaser, haugoug, huang, benini}@iis.ee.ethz.ch
†Electrical, Electronic, and Information Engineering, University of Bologna, Italy, davide.rossi@unibo.it

***Abstract*— Parallel ultra low power computing is emerging as an enabler to meet the growing performance and energy efficiency demands in deeply embedded systems such as the end-nodes of the internet-of-things (IoT). The parallel nature of these systems however adds a significant degree of complexity as processing elements (PEs) need to communicate in various ways to organize and synchronize execution. Naive implementations of these central and non-trivial mechanisms can quickly jeopardize overall system performance and limit the achievable speedup and energy efficiency. To avoid this bottleneck, we present an event-based solution centered around a technology-independent, light-weight and scalable (up to 16 cores) synchronization and communication unit (SCU) and its integration into a shared-memory multicore cluster. Careful design and tight coupling of the SCU to the data interfaces of the cores allows to execute common synchronization procedures with a single instruction. Furthermore, we present hardware support for the common barrier and lock synchronization primitives with a barrier latency of only eleven cycles, independent of the number of involved cores. We demonstrate the efficiency of the solution based on experiments with a post-layout implementation of the multicore cluster in a 22 nm CMOS process where the SCU constitutes less than 2 % of area overhead. Our solution supports parallel sections as small as 100 or 72 cycles with a synchronization overhead of just 10 %, an improvement of up to 14 × or 30 × with respect to cycle count or energy, respectively, compared to a test-and-set based implementation.**

I. Introduction

The trend for ever increasing single-core performance, mostly through faster clock frequencies, has long come to an end for desktop, server and portable computers [1] as the resulting power consumption and heat dissipation became impractical to handle. To still meet the demands for growing performance, the usage of multi-core processors has become a widely adopted standard. Their efficient utilization however introduced a new and significant aspect of complexity both for the surrounding hardware as well as software running on them. The turn to multi-core or parallel processors, often simply referred to as multicores, recently also reached the embedded computing domain: Rapidly evolving capabilities of mobile, handheld and wearable devices as well as the era of the internet-of-things (IoT) [2] push the performance requirements for both general-purpose and specialized computing to very challenging levels. While energy efficiency is a crucial metric for most emebedded devices, it is especially true for IoT endpoints where self-sustainability and battery lifetime are key features.

One of the predominant approaches to meet the stated high performance and feature demands while maintaining energy efficient operation is parallel near-threshold computing [3]–[6] which heavily relies on parallel architectures combined with digital voltage and frequency scaling (DVFS). Making shared resources such as memories or peripherals available to all PEs naturally incurs a non-negligible overhead in terms of area and power for arbitration, interconnect and routing hardware. In order to not jeopardize the intended energy-saving efforts, consequently as many PEs as possible must be (uniformly) utilized and the remaining ones aggressively power managed. How small an application can be in order to still profit from parallelization in terms of energy efficiency while taking into account both hard- and software overhead is therefore an important figure of merit for embedded parallel systems [3]. Similar and even more important is the granularity with which applications can be parallelized before any gain gets defeated by energy spent for parallelization [4]. This figure is not only relevant for data-dependent programs that require constant thread interaction but also for those where the required granularity is non-constant and varies during execution.

To support the execution of parallel programs, there must exist mechanisms that enable PEs to organize and synchronize execution, communicate with each other and to manage access to exclusive resources, implementing synchronization primitives typical of shared memory parallel programming models such as OpenMP [7]. In this work, we focus on *cache-less shared-memory multiprocessors* as they can provide DSP-class processing performance while maintaining microcontroller-class energy efficiency [4] when programmed with awareness of the memory hierarchy. The shared memory concept makes data exchange between cores and communication thereby trivial, however still a proper (atomic) handshake protocol is required to guarantee functional correctness. All synchronization mechanisms exhibit two important aspects, namely 1) on which basis they are implemented, both from a hard- and software perspective and 2) how PEs are treated that have to wait for the continuation of execution. While achieving near-ideal speedup is already extremely challenging when synchronization and communication overhead is completely neglected [8], both aspects contribute largely to overall achievable energy and performance gains; improper or straight-forward solutions can consequently have strong adverse effects [4], [9].

A universal approach is to use atomic memory access in the form of hardware-supported *test-and-set* (TAS) or *compare-*

978-3-9819263-2-3/DATE19/©2019 EDAA

and-swap (CAS) and the like in combination with spin locks on shared variables. While all synchronization mechanism can be implemented on this basis, spin locks as a form of *busy waiting* heavily violates the principle of aggressively power managing idle cores, with the repeated variable polling causing not only wasted energy through unnecessarily active cores and loaded interconnect systems, but also deteriorated performance for the working core(s) due to congestion on the processor-to-memory interconnect. Additionally, as each memory location can only concurrently be accessed by a single core, even best-case latency for synchronization primitives grows with the number of contestants. The limitation of busy waiting can be avoided by employing interrupt-based approaches [10], [11] and power managing cores after the first unsuccessful access to an atomic variable. While the scalability issues remain unresolved, the handling of interrupts usually incurs large software overhead due to the associated context switches. Furthermore, complex interrupt controllers that support nesting and (multi-level) prioritization (e.g., [12]) are often not required in computation-centric clusters, targeted to handle DSP kernels with maximum energy efficiency.

In this paper, we therefore propose a hardware-supported solution for heterogeneous multicore synchronization and communication that is completely based on idle waiting on events, i.e., halting execution when necessary and continuing after a signaling event without context change. We focus on aggressively reducing the overhead devoted to entering and returning from *restful wait-states* as well as for PE-to-PE signaling and communication in terms of cycles and therefore energy. We thereby enable 1) energy efficient fine-grain parallelization and 2) power managing during very short idle phases in the order of 10s of cycles. Contrary to the majority of previous art, we do not rely on atomic access to shared memory with possibly adverse effects on system performance while also not requiring large message buffers or content addressable memories (CAMs) solely devoted to synchronization.

II. Related Work & Contribution

The topic of energy efficient synchronization for embedded multicore systems has become of great interest simultaneously with their rise as neglecting it can significantly hinder energy and performance gains [4], [8], [9]. Previous art related to the analysis of different software implementations of synchronization primitives can be found in [9]; a wide range of works follows the approach of employing specialized hardware blocks, aimed to accelerate synchronization and ultimately improving performance and energy efficiency of multicore systems [9]–[11], [13], [14], which we also do. A multitude of works [15]–[17] raises the question whether any kind of power managing should at all be employed immediately in case a core is blocked at a synchronization point due to the overhead associated with entering and leaving idle modes. The authors of [16], [17] further propose to equalize execution speeds among all cores by automatically tuning the clock frequency of each one. Ultimately, the assumption of asynchronous clocks implies loosely coupled cores, which in turn implies significant inter-core communication overhead (simply going across dual-clock FIFOs causes several cycles of latency). Hence, all the solutions presented in the literature are suitable for parallel synchronization-free regions of thousands of instructions; we however target one to two orders of magnitude smaller regions.

We follow a fundamentally different and more regular approach where the whole cluster operates from a single clock source and workload is distributed as equally as possible in the application itself. The focus is set on making fine-grain power management in the form of clock gating amenable by reducing the latency to enter and leave idle mode in a deterministic way to three cycles. We provide hardware support for the typically required *barrier* and *lock* primitives, reducing the number of required bus transactions per core for both to one. We present the central enabler for these features, a light-weight, modular and flexible hardware synchronization and communication unit (SCU) designed for the usage in embedded multicore clusters as the central components of computing-centered embedded systems. We detail architectural aspects that enable synchronization primitives with very small overhead in terms of latency, code size as well as both hardware- and software complexity. Furthermore, the integration of the unit into a tightly-memory coupled RISC-V based cluster with minimal adaption of its cores and interconnect systems is shown.

In particular, we propose 1) support for endogenous event-based signaling between cores and hardware support for synchronization primitives such as barriers or locks, 2) support for exogenous events from both tightly coupled accelerators as well as remote PEs and peripherals and 3) coupling of the unit to the core data interfaces, enabling restful wait states with fused information exchange in a single load operation.We present a technology-independent hardware implementation of the SCU, analyze the scalability of the design and give experimental results obtained from the integration into the RISC-V based cluster and post-layout simulations in 22 nm CMOS. The efficiency of the architecture is demonstrated through speedup and energy analysis of synchronization primitives in a parallel programming model [7] for synthetic benchmarks.

III. Architecture

A. Heterogeneous multi-core cluster

The employed parallel computing cluster is based on a configurable number of RISC-V cores (typically up to 16) that belong to the microcontroller-class, however are extended with several powerful DSP extensions, increasing the achievable computation performance for typical kernels significantly [18]. To cope with tasks that require even higher processing power but are specialized, high-bandwidth tightly memory-coupled heterogeneous processing elements such as neural network accelerators or DMAs are additionally incorporated [19], [20]. The architecture is depicted in Fig. 1, showing all important building blocks. All cores request their instructions from a shared cache; the L1 data memory serves as a local, shared scratchpad memory and is single-cycle accessible by all PEs through a dedicated logarithmic interconnect. The L1 interconnect further implements test-and-set on a small address subset

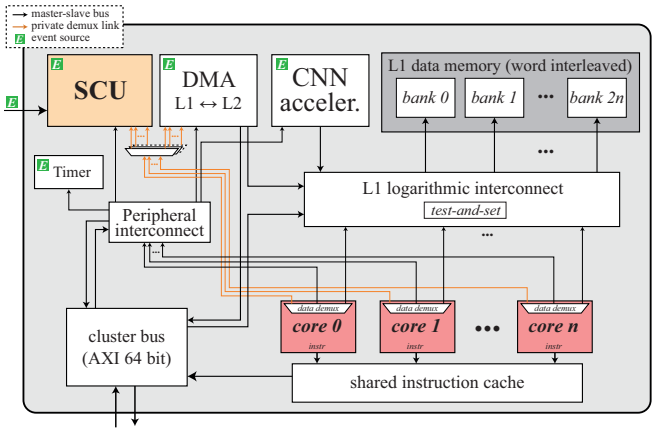

Fig. 1. Heterogeneous multi-core cluster, incorporating the synchronization and communication unit (SCU).

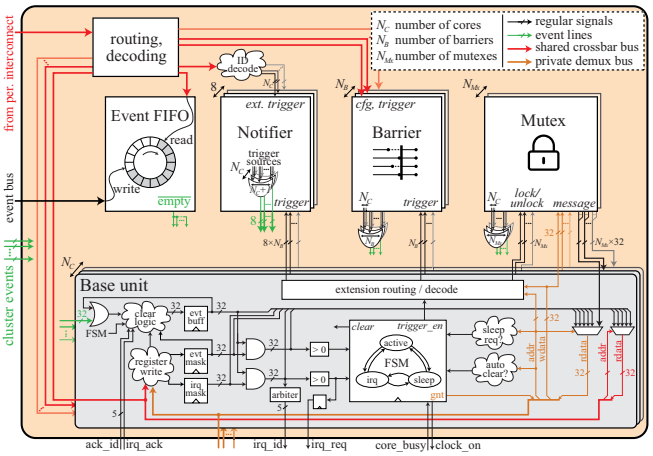

Fig. 2. Simplified architecture of the SCU and extensions, functional overview of the base units. Signals on the bottom connect to the cores, signals on the left to the cluster and higher hierarchies.

to provide generic atomic memory access, used as a baseline for the experiments in this work. A peripheral interconnect system with relaxed throughput requirements enables the cores to configure and control the additional PEs as well as peripherals such as e.g., timers or also the SCU.

B. SCU Concept & Base Unit

A simplified architecture diagram of the SCU is depicted in Fig. 2; detailed insight is given into the architecture of the SCU base unit, instantiated once per RISC-V core and responsible for the fundamental functionality of the SCU, i.e., event and wait state management. The implemented concept is based on 32 level-sensitive *event lines* per core that are connected to associated *event sources* of generic nature. Per core, each set event line is stored into an *event buffer* which in turn is maskable through *event masks* and *interrupt masks* with the resulting signals being used as fundamental trigger points for the tasks of the SCU. Basic interrupt support is also provided to handle exceptions and other irregular events. The control flow within each base unit is orchestrated by a FSM with the three states *active, sleep* and *interrupt-handling* which closely reflects the state of the associated core but employs SCU-internal information to decide about state transitions: Few internally computed Boolean signals are used as inputs, examples include whether the core requested a wait state or if the masked versions of the event buffer have any bit set.

C. SCU Extensions

While some of the 32 manageable event sources per base unit are located outside of the SCU (e.g., specialized PEs or cluster-internal peripherals), the bigger share of the events is responsible for core-to-core signaling and thus generated within the SCU by a set of *SCU extensions*. All extensions generate a (usually) core-specific event that allows the involved cores to continue execution. All extensions have trigger and configuration signals connected to each base unit; the associated functionality is reachable through memory-mapped address regions. The set of all trigger signals gets processed and results in the aforementioned per-core event source; for extension types where simultaneous usage of more than one instance by a core has no sensible usecase, all event lines targeting a specific core get logically ORed to a single line over all instances. The most relevant extensions for the targeted parallel programming frameworks are the *barrier* and *mutex* types as they realize the functionality of the fundamental programming elements *parallel sections* and *critical sections* for parallel programming models such as e.g., OpenMP [21], which we have implemented on top of a bare-metal runtime. The number of barrier units can be anything between a single and 16 to support every team-building variant even for the maximum number of supported cores. Whenever a set of *target cores* may only continue execution after a (possibly different) set of *triggering cores* has reached a defined point in the program, this extension allows to reduce the overhead spent for such orchestration. The possibly different or even disjoint sets of target and triggering cores are reflected in independent registers with the target cores getting sent an event once all triggering cores have arrived at the barrier and entered restful wait states.

The functionality of each mutex instance is realized with a single address, with read access corresponding to lock attempts and write access to unlocking it. The core who gets ownership of the mutex is, consequently following our modular approach, informed through an event. The notifiers realize a general-purpose, matrix-style single-, multi- and broadcast core-to-core signaling mechanism. Each notifier is connected to an independent event source per core and is triggerable from every core.

An additional, heterogeneous trigger source is provided through a system-wide peripheral bus, allowing higher-level cores to wake up idle clusters. Since complex SoCs can feature more than 32 distinct event sources (e.g., from medium/low-speed on- or off-chip peripherals), the event FIFO is used to serialize one event line, shared with all base units. In this way, an ample amount of non-time critical event sources (up to 256) can be handled. A simple 8 bit event bus with handshake pushes the identifier of incoming events into the FIFO; the associated event remains asserted until the buffer is empty.

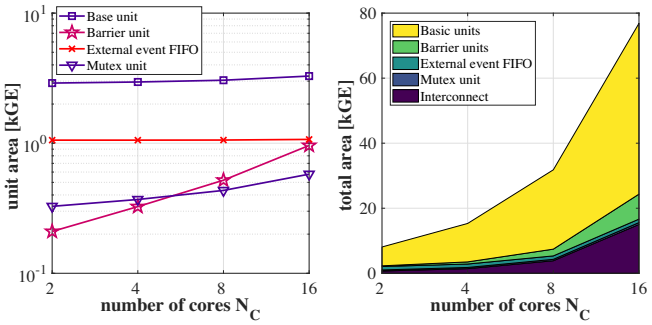

Fig. 3. Area scaling of the basic unit and the available extensions.

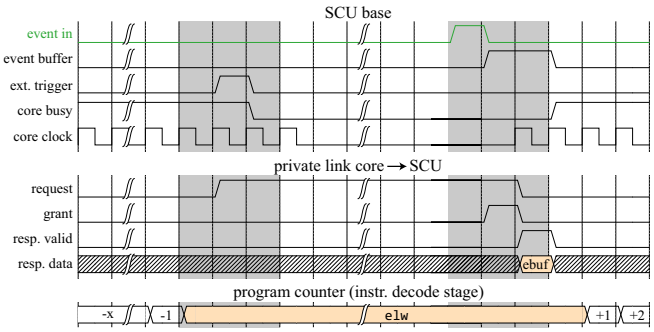

Fig. 4. Interfacing of the SCU with a RISC-V core and corresponding timing. Shaded intervals correspond to transitions from and to sleep state, respectively.

D. SCU Scalability

Fig. 3 shows the area scaling of the SCU when the main parameter, the number of cores N_C, is increased, both for the total area (with the number of barriers $N_B = N_C/2$, number of mutexes $N_{Mx} = 1$) as well as for the individual sub-units; the plots show post-synthesis numbers in a 22 nm process with a timing constraint of 500 MHz. We restrict the maximum number of cores to 16 as the low-latency interconnect systems of the hosting cluster do not scale well beyond this configuration. An analysis of the slopes in the double-logarithmic unit area plot in Fig. 3 yields that the individual components of the SCU scale at worst mildly super-linear (barrier units), most however sub-linearly or even remain constant in size. The resulting overall area favorably scales sub-linearly or linearly for up to eight cores and mildly super-linearly when further increasing N_C to the maximum configuration. In all cases, the base units make up for the lion share while the essential barrier and mutex extensions contribute little extra area; routing for the system-wide peripheral interconnect to the connected base- and barrier units also contributes significantly for the two largest configurations.

E. SCU Integration

Wherever possible, the SCU is connected as a regular, memory mapped slave to the shared peripheral interconnect. In this way however, only a single core can get access in a given cycle, additionally latency is non-deterministic. Concurrent access to the event buffering and –triggering base units is crucial for synchronization performance, as otherwise the performance bottleneck originating from the race for access to shared variables known from atomic memory implementations effectively remains. Consequently, additional private, one-to-one links between each core and its corresponding base unit are employed. As can be seen in Fig. 1, a demux at the data port of each core routes requests to the L1 interconnect, the peripheral interconnect or the private link; in the last case a final routing decision between SCU and DMA is made since the latter similarly requires a fast path for configuration. The private link is purely combinational between core data interface and SCU, enabling single-cycle access. In the example of the barrier extensions, the private links allow any number of cores to concurrently trigger a barrier, rendering the associated synchronization primitive cycle cost constant.

As the one-to-one correspondence between each core and associated base unit is equivalent to aliasing its address space, no address calculations dependent on the core-ids are required, accelerating and simplifying synchronization primitives. Even though usually not necessary, all base units are accessible by every core as well as cluster-external masters through the peripheral interconnect and a global address space. Registers that would put the corresponding core to sleep when accessed are excepted from this access method.

IV. Single-Instruction Synchronization

When arriving at a synchronization point, systems relying on event-based signaling usually require cores to execute very similar sequences independent of the concrete nature of the synchronization point: First, the synchronization-managing entity has to be informed about the arrival, followed by a special instruction that initiates power managing of the issuing core, i.e., in our approach, clock gating it. Once execution can continue, the triggering event has to be cleared in order to be able to enter following wait states. In case of multiple active event sources, prior to clearing, the event buffer must also be read in order to direct program flow accordingly.

Following our goal of aggressively reducing synchronization overhead, we extensively exploit the dedicated link between each core and corresponding SCU base, the available address space (1 Kibit, per-core aliased) as well as the in-order execution model of the used core to handle the described, frequent sequence with a single instruction. We employ the basic *load-word* (`lw`) instruction with an altered opcode in order for the core controller to distinguish regular and event-related word loads; our added instruction has the mnemonic `elw` (for *event-load-word*). The resulting transactions at the data ports of the cores are identical to those originating from regular loads; *event-load-word* operations addressing the private core-SCU links however have the following desired side-effects:

1) If within a defined set, the employed address causes the *sleep-req* logic in Fig. 2 to flag the request of a wait state to the FSM which in turn blocks the *grant* of the read transaction. Consequently, the core pipeline gets stalled; the core controller flags idleness to the SCU as soon as any potentially ongoing multi-cycle instruction is completed. Finally, the SCU sends the core into a restful wait state by turning off its clock.
2) In addition, a subset of the *sleep-req* address-set causes the

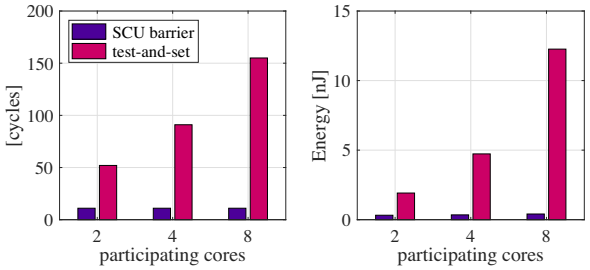

Fig. 5. Barrier primitive cost in terms of cycles and energy.

SCU base to trigger an extension prior to sleeping. Examples include to request ownership of a mutex, triggering of a barrier or sending notifiers. 3) Once an activated event is present in the event buffer, the FSM initiates the core wakeup by re-enabling its clock and simultaneously asserting the so far blocked grant. 4) The response channel, valid one cycle after a granted request, is used to inform the issuing core about the content of its event buffer. For extensions that require message passing, this mechanism can also be leveraged to intrinsically pass up to one word between cores. In the example of the mutex extension, we added the possibility for the unlocking core (which does so through a non-blocking write transaction) to pass a message to the following locking core. 5) Again controllable through the address, all events that are activated and present in the event buffer can be automatically cleared.

The process is illustrated in Fig. 4 with all optional possibilities used (extension triggering on wait-state entering, automatic buffer clear). The required changes in the core to support this powerful mechanism are limited to decoding the `elw` instruction to release the busy-signal which would otherwise remain asserted on a pipeline stall due to the pending load.

V. Experimental Results

To demonstrate the efficiency of the proposed SCU-based solution, we provide an overhead analysis for the two most commonly used synchronization primitives, namely barriers used in parallel, and mutexes, used in critical sections [21]. As our primary goal of energy reduction for fine-grain parallelized tasks is mainly achieved by reducing the execution time of synchronization primitives, we provide a comparison of our solution with a software-based test-and-set approach both in terms of cycles as well as in terms of the resulting energy.

A. Hardware Platform

Energy results were obtained from a placed and routed implementation of the multicore cluster in Fig. 1 in a 22 nm CMOS process. It features eight RISC-V cores, 64 kByte L1 memory and 8 kByte of instruction cache. The implemented module measures 0.8 mm×1.4 mm with pre-placed SRAM macros for the L1 memory and a 250 MHz timing constraint[1]; the SCU with four barriers and one mutex extension account for less than 2 % of the cluster area. Energy was computed as the product of used cycles and average power consumption, obtained through simulations of the resulting gate-level netlist (typical process corner, 0.8 V supply, 25 °C).

[1]Tools used: Synopsys Design Compiler 2016-12, Cadence Innovus 17.11

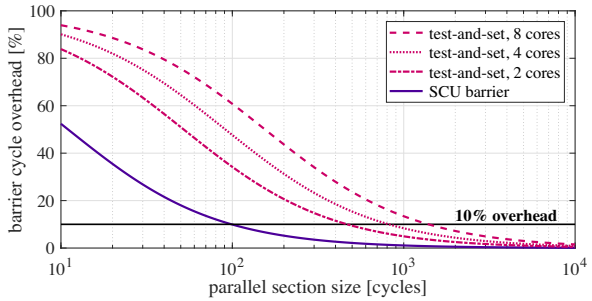

Fig. 6. Relative overhead vs. parallel section size with respect to cycles of test-and-set software-based and SCU hardware-based barrier implementations.

B. Synthetic Benchmarks

To accurately quantify the synchronization overhead, we execute a parallel section of controllable size on a varying number of cores. At the end of the parallel section, the involved cores need to synchronize execution which is done both through a test-and-set (TAS) based software barrier as well as with the SCU barrier extension. For fair comparison, the TAS-based version does not let the early arriving cores spin at the barrier but also uses the idle-wait mechanism described in Sec. IV. To quantify the savings for mutual exclusive code sections, we add a critical section at the end of the parallel section, implemented either with TAS-based atomic memory access or the SCU mutex extension. The placement of the critical section reflects typical kernel usecases where a shared variable needs to be updated by all worker cores at the end of a parallel section, taking only very few (up to ten) cycles. The benchmarks were compiled with an extended version of the riscv-gcc 7.1.1 toolchain that supports the `elw` instruction.

Fig. 5 shows the cost of a barrier in cycles for both implementations. As already mentioned in Sec. IV, the possibility of concurrent access to the SCU for all cores allows to keep the cycle overhead constant, no matter how many involved cores; the low SCU access latency and instruction count for a barrier results in only eleven cycles for the execution of the primitive. Since the length of the parallel section has no influence on the execution time of following synchronization primitives for both implementations, the relative cycle overhead can be calculated for arbitrary parallel section sizes, as is shown in Fig. 6 for parallel and Fig. 7 for critical sections. For parallel sections of up to 100 cycles, the relative overhead is reduced by at least 24 %; when allowing for a 10 % synchronization cycle overhead, our solution allows for 4.7 to 14 × smaller minimum parallel section sizes. The difference between the two implementations gets even larger when energy is considered; Fig. 5 shows that the SCU supported variant consumes between 4 × (two active cores) and 30 × (all eight cores) less energy, in the latter case allowing for parallel sections as small as 72 instead of 2154 cycles with only 10 % of the energy spent for synchronization. While a large part of the improvement stems from the reduced cycle count, the TAS-based implementation also suffers from energy-intensive L1 accesses.

The behavior for mutual exclusive sections is evaluated by measuring the actual amount of cycles required for all cores

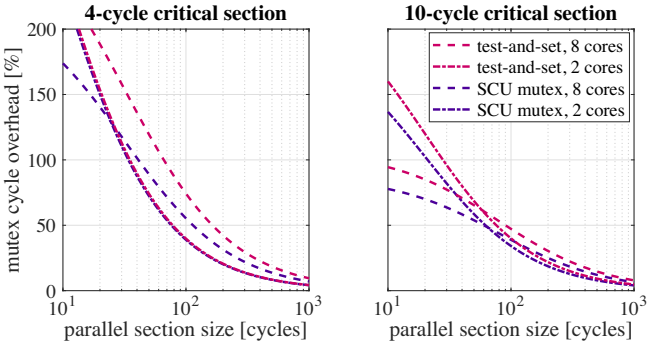

Fig. 7. Relative cycle overhead for critical sections vs. parallel section length for TAS-based and SCU mutex supported implementations. The lines corresponding to four cores have been omitted to improve graph readability.

to execute the parallel plus critical section. Ideally, this should take $T_{\text{ideal}} = T_{\text{para}} + T_{\text{crit}} N_C$ cycles with T_{para} and T_{crit} being the length of the parallel and critical section respectively, and N_C the number of involved cores. The relative cycle overhead is computed as $(T_{\text{total}} - T_{\text{ideal}})/T_{\text{ideal}}$ and shown in Fig. 7 for two different T_{crit}. Compared to the cycle reduction for parallel sections, the savings achieved with the SCU mutex are much smaller; in the extreme case of $T_{\text{crit}} = 4$ and $N_C = 2$, no difference is perceptible. However, for $T_{\text{crit}} = 4$, eight cores and very small parallel sections, causing the relative overhead to exceed 100 % with both implementations, the dedicated SCU architecture can reduce the overhead for e.g., $T_{\text{para}} = 20$ from 188 % to 140 %. The main reason for the limited performance gains through the SCU based implementation is that a TAS-variable inherently allows for cycle-efficient mutex implementation while for the barrier case a counter must be made exclusively accessible.

The picture is different however for the energy spent for small critical sections: As can be seen in Fig. 8, the energy consumed by the TAS-based solution is between five and 62 % higher ($T_{\text{crit}} = 10$, $N_C = 8$), in the latter case translating into a minimum parallel section size of 992 instead of 1603 cycles for 10 % energy overhead, an improvement of 38 %. Similar to the barrier case, the avoidance of expensive L1 memory accesses makes the SCU superior with respect to energy efficiency, even with significantly smaller cycle count reduction.

VI. Conclusion

We showed that straight-forward implementations of synchronization mechanisms can significantly compromise energy efficiency of embedded multicore systems and prohibit fine-grain parallelization [4]. We proposed a hardware-supported solution that aggressively reduces the overhead used for synchronization primitives to enable parallel sections in the order of 10s of cycles. Careful and tailored design of the presented SCU improves energy efficiency by reducing instruction and cycle count through low-latency and parallel access as well as by doing away with contending shared variable access. An extended version of this work is currently in preparation, including a detailed analysis of the impact of the proposed synchronization solution on the performance and energy of parallel kernels and benchmarks.

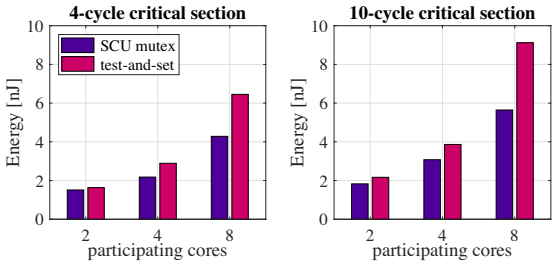

Fig. 8. Energy spent for the execution of critical sections.

References

[1] D. Geer, "Chip makers turn to multicore processors," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, Sept. 2013.

[3] A. Y. Dogan et al., "Power/performance exploration of single-core and multi-core processor approaches for biomedical signal processing," *Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 102–111, Sept. 2011.

[4] D. Rossi et al., "Energy-efficient near-threshold parallel computing: The PULPv2 cluster," *IEEE Micro*, vol. 37, no. 5, pp. 20–31, Sept. 2017.

[5] R. G. Dreslinski, B. Zhai, T. Mudge, D. Blaauw, and D. Sylvester, "An energy efficient parallel architecture using near threshold operation," *Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pp. 175–188, Sept. 2007.

[6] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming Moore's Law through energy efficient integrated circuits," *Proc. of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[7] A. Marongiu and L. Benini, "An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 222–236, Feb 2012.

[8] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Trans. on Computers*, vol. 41, no. 7, pp. 33–38, July 2008.

[9] O. Golubeva, M. Loghi, and M. Poncino, "On the energy efficiency of synchronization primitives for shared-memory single-chip multiprocessors," *Proc. of ACM Great Lakes Symp. on VLSI (GLSVLSI)*, pp. 489–492, March 2007.

[10] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Efficient synchronization for embedded on-chip multiprocessors," *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.*, vol. 14, no. 10, pp. 1049–1062, Oct. 2006.

[11] C. Yu and P. Petrov, "Low-cost and energy-efficient distributed synchronization for embedded multiprocessors," *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.*, vol. 18, no. 8, pp. 1257–1261, Aug. 2010.

[12] ARM Holdings, *ARM Cortex-M7 Processor Technical Reference Manual, Chapter 7. Nested Vectored Interrupt Controller*, https://developer.arm.com/docs/ddi0489/latest/nested-vectored-interrupt-controller, July 2015.

[13] H. Xiao, T. Isshiki, D. Li, H. Kunieda, Y. Nakase, and S. Kimura, "Optimized communication and synchronization for embedded multiprocessors using ASIP methodology," *Information and Media Technologies*, vol. 7, no. 4, pp. 1331–1345, Jan. 2012.

[14] S. H. K. et al., "*C-Lock*: Energy efficient synchronization for embedded multicore systems," *IEEE Trans. on Computers*, vol. 63, no. 8, pp. 1962–1974, Aug. 2014.

[15] C. Ferri, R. I. Bahar, M. Loghi, and M. Poncino, "Energy-optimal synchronization primitives for single-chip multi-processors," *Proc. of ACM Great Lakes Symp. on VLSI (GLSVLSI)*, pp. 141–144, 2009.

[16] M. Loghi, M. Poncino, and L. Benini, "Synchronization-driven dynamic speed scaling for MPSoCs," *Proc. of Int. Symp. on Low Power Electronics and Design (ISLPED)*, pp. 346–349, Oct. 2006.

[17] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, "Exploiting barriers to optimize power consumption of CMPs," *Proc. of Int. Parallel and Distributed Processing Symp. (IPDPS)*, April 2005.

[18] M. Gautschi et al., "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct 2017.

[19] F. Conti et al., "An IoT endpoint System-on-Chip for secure and energy-efficient near-sensor analytics," *IEEE Trans. Circuits Syst. – I: Reg. Papers*, vol. 64, no. 9, pp. 2481–2494, Sept 2017.

[20] A. Pullini, D. Rossi, I. Loi, A. D. Mauro, and L. Benini, "Mr. Wolf: a 1 GFLOP/s energy-proportional parallel ultra low power SoC for IoT edge processing," *Proc. IEEE European Solid-State Circuits Conf. (ESSCIRC)*, Sept. 2018.

[21] OpenMP Architecture Review Board, *The OpenMP API specification for parallel programming*, https://www.openmp.org.