Energy-Efficient Hardware-Accelerated Synchronization for Shared-L1-Memory Multiprocessor Clusters

Florian Glaser, *Student Member, IEEE*, Giuseppe Tagliavini, *Member, IEEE*, Davide Rossi, *Member, IEEE*, Germain Haugou, Qiuting Huang, *Fellow, IEEE*, and Luca Benini, *Fellow, IEEE*

Abstract—The steeply growing performance demands for highly power- and energy-constrained processing systems such as end-nodes of the Internet-of-Things (IoT) have led to parallel near-threshold computing (NTC), joining the energy-efficiency benefits of low-voltage operation with the performance typical of parallel systems. Shared-L1-memory multiprocessor clusters are a promising architecture, delivering performance in the order of GOPS and over 100 GOPS/W of energy-efficiency. However, this level of computational efficiency can only be reached by maximizing the effective utilization of the processing elements (PEs) available in the clusters. Along with this effort, the optimization of PE-to-PE synchronization and communication is a critical factor for performance. In this article, we describe a light-weight hardware-accelerated synchronization and communication unit (SCU) for tightly-coupled clusters of processors. We detail the architecture, which enables fine-grain per-PE power management, and its integration into an eight-core cluster of RISC-V processors. To validate the effectiveness of the proposed solution, we implemented the eight-core cluster in advanced 22 nm FDX technology and evaluated performance and energy-efficiency with tunable microbenchmarks and a set of real-life applications and kernels. The proposed solution allows synchronization-free regions as small as 42 cycles, over 41× smaller than the baseline implementation based on fast test-and-set access to L1 memory when constraining the microbenchmarks to 10 percent synchronization overhead. When evaluated on the real-life DSP-applications, the proposed SCU improves performance by up to 92 and 23 percent on average and energy efficiency by up to 98 and 39 percent on average.

Index Terms—Energy-efficient embedded parallel computing, fine-grain parallelism, tightly memory-coupled multiprocessors

1 Introduction

After being established as the architectural standard for general-purpose and high-performance computing over a decade ago [1], [2], the paradigm of chip multiprocessor (CMPs) is as well being adopted in the embedded computing domain [3], [4]. One of the key drivers for the exploitation of multi-core systems in the deeply embedded domain is the increasing performance demand in internet-of-things (IoT) end-nodes. This class of devices needs to flexibly handle multiple sensor data streams [5], [6] (e.g., from low-resolution cameras or microphone arrays), and perform complex computations to reduce the bandwidth over energy-intensive wireless data links.

- *Florian Glaser, Germain Haugou, and Qiuting Huang are with the Department of Information Technology and Electrical Engineering (D-ITET), ETH Zürich, Switzerland. E-mail: {glaser, haugoug, huang}@iis.ee.ethz.ch.*
- *Giuseppe Tagliavini and Davide Rossi are with the Department of Electrical, Electronic, and Information Engineering, University of Bologna, Italy. E-mail: {giuseppe.tagliavini, davide.rossi}@unibo.it.*
- *Luca Benini is with the Department of Information Technology and Electrical Engineering (D-ITET), ETH Zürich, Switzerland, and also with the Department of Electrical, Electronic, and Information Engineering, University of Bologna, Italy. E-mail: benini@iis.ee.ethz.ch.*

Manuscript received 3 Apr. 2020; revised 31 Aug. 2020; accepted 27 Sept. 2020. Date of publication 6 Oct. 2020; date of current version 19 Oct. 2020. (Corresponding author: Florian Glaser.) Recommended for acceptance by R. Ge Digital Object Identifier no. 10.1109/TPDS.2020.3028691

As the straightforward replacement of microcontroller cores with more powerful core variants featuring multiple-issue, multiple-data pipeline stages, and high operating frequencies, naturally jeopardizes energy-efficiency [7], researchers are turning to *parallel* near-threshold computing (NTC) [8]. This approach enables improvements of energy-efficiency by up to one order of magnitude [9] by reducing the supply voltage of the underlying CMOS circuits to their optimal energy point (OEP) [10] while recovering the performance degradation at low voltage through the exploitation of parallel execution over multiple software-programmable cores. However, parallel NTC can only achieve the fundamental goal of increasing energy-efficiency with parallel workloads and when the underlying hardware can effectively exploit the parallelism present in applications. Moreover, in sequential portions of applications, parallel hardware resources such as processing element (PEs) and part of the interconnect towards the shared memory consume power without contributing to performance. Hence, during these periods, all idle components must be aggressively power-managed at a fine-grain level.

The described requirements highlight the need for communication, synchronization, and power management support in parallel processing clusters. Communication mechanisms are required for PEs to exchange intermediate results and orchestrate parallel execution. In this work, we focus on *shared-memory* multiprocessors that typically rely on data-parallel computational models. For this class of systems, data

exchange is trivial, reducing the synchronization and communication requirements to pointer exchange and data validity signaling. Still, waiting for and notifying PEs fast and efficiently is essential for any application that contains data dependencies between threads (i.e., applications that cannot be *vectorized*). Consequently, the support for synchronization mechanisms remains mandatory also for this type of parallel processing systems.

The most straightforward way to enable functionally correct implementations of every kind of multi-PE synchronization is to provide atomic access to the shared memory (or a part thereof) and use *spinlocks* on thereby protected shared variables. While this approach is universal, flexible, and requires small hardware overhead, its *busy-waiting* nature jeopardizes both performance and energy efficiency, as every contestant repeatedly accesses the shared variables until all have gotten exclusive access.

To overcome the explained challenges and limitations, we propose a light-weight hardware-supported solution that aims at drastically reducing the synchronization overhead in terms of cycles and – more importantly – energy. This approach makes fine-grain parallelization for the targeted shared-memory NTC processing clusters affordable since all signaling is done by *restfully waiting* on *events*, i.e., halting and resuming execution at synchronization points without changing the software context. The foundation of the proposed solution is the synchronization and communication unit (SCU), a compact, single-cycle-accessible shared peripheral that centrally manages event generation, requiring only a single additional instruction compared to the base instruction set architecture (ISA). The SCU features general-purpose PE-to-PE signaling and an easily extensible set of commonly used synchronization primitives. In this work, we focus on *barriers* and *mutexes*, as they correspond to the parallel and critical section constructs that are fundamental in parallel programming models such as OpenMP [11]. For cases where a completely balanced utilization of all PEs is not possible, the SCU features fine-grain power management (PM) in the form of per-PE clock gating, which allows saving energy during idle periods as short as tens of cycles.

To demonstrate the capabilities of the solution, we integrate the SCU into a tightly-coupled cluster that features eight digital signal processing (DSP)-capable RISC-V cores. The SCU, as well as the multi-core cluster, are silicon-proven in multiple technology nodes [12], [13], [14], and are available as open-source hardware under a permissive license [15]. We carry out all experiments on a register-transfer level (RTL) model and a gate-level, fabrication-ready implementation of the multi-core cluster in a 22 nm process, allowing us to obtain cycle-exact performance numbers as well as to measure energy with an accuracy close to that of physical system realizations. Adopting this methodology, we illustrate both the opportunities and relevance of lowering synchronization overhead for parallel NTC through synthetic benchmarks and a set of DSP kernels that are typical for the targeted system type. The results obtained in our evaluation show that the SCU allows synchronization-free region (SFRs) as small as 42 cycles, which is more than 41× smaller than the implementation based on fast test-and-set (TAS) used as a baseline for comparison when constraining the synthetic benchmarks to 10 percent synchronization overhead. Moreover, when evaluated on the set of application kernels, the proposed SCU improves performance by up to 92 percent and energy efficiency by up to 98 percent. The area cost of the SCU is small (less than 2 percent of the cluster), which also implies that its added static power contribution is completely negligible, even at low operating voltage and frequency, typical of NTC systems, with respect to the power savings it enables in the whole cluster.

The remainder of this paper is organized as follows. Section 2 provides a comprehensive overview of the prior art related to synchronization in embedded multiprocessors. Section 3 discusses the relevance of fine-grain synchronization in the context of the targeted system type. The architecture of the proposed SCU and the multi-core cluster is explained in Section 4, followed by the thereby enabled concept of aggressively reducing the overhead for synchronization primitives in Section 5. The baseline method, as well as the experimental setup and methodology which we used, can be found in Section 6, followed by experiments with the microbenchmarks and real-life DSP-applications. Section 7 summarizes and concludes the work.

2 Related Work

The limitations of straight-forward synchronization support based on atomic memory access have been broadly recognized by the research community. A variety of works, therefore, proposes – similar to our approach – hardware-accelerated solutions [16], [17], [18], [19]. Reviews of the performance and characteristics of software-based solutions for shared-memory multiprocessors can be found in [20], [21], [22], [23].

Multiple works [23], [24], [25], [26] propose to dynamically adjust the speed of individual PEs at runtime to equalize their execution speed instead of power-managing them. However, this approach incurs significant control and circuit complexity due to the required asynchronous clocks and severely increases the latency between PEs and memory. Consequently, it only pays off if the costs for entering and leaving low-power modes are in the order of thousands of cycles, as the authors of [23] assume. Our approach instead aims at using a single synchronous clock, implementing a simple variant of PM that is suitable for very short idle periods, equalizing workload with fine-grain parallelization, and ultimately gain from system-wide frequency and supply scaling.

The prior art in hardware-accelerated synchronization for synchronously clocked embedded systems covers a wide range of targeted system types and architectural concepts. What follows is a review of the relevant references that target microcontroller-type (or at least similar) systems, structured by key aspects.

Synchronization-Free Region Size. With the constraint to gain energy-efficiency and performance through parallelization, the cost of synchronization tasks causes a lower bound for the average length of periods that PEs can work independently from each other, often referred to as the *SFR size*. Consequently, the ability to handle typical synchronization tasks in the order of tens of cycles and below is of major importance for energy-efficient parallelization in the

context of the targeted processing clusters. For most references, the SFR size for which the respective results are reported must be implicitly determined by analyzing the employed benchmarks. Multiple references [27], [28], [29] utilize parallel benchmarking suites such as STAMP [30] or SPLASH-2 [31], which feature SFRa of at least 10.000s of cycles and are, therefore – in addition to large data set sizes – not suitable for the microcontroller-type clusters we target. The lack of fit of the SPLASH-2 suite is also mentioned in [17]. As a suitable alternative, the authors propose the usage of a subset of the Livermore Loops [32], a collection of sequential DSP-kernels that can be parallelized with reasonable effort as the original code is annotated with data-hazard pragmas. The parallel versions of loops 2, 3, and 6 are provided in [17] and used in [16], [17], [18]; we include loops 2 and 6 in our set of applications.[1] Our analysis in Section 6 yields SFR sizes of as small as 104 cycles, which matches the range targeted by our proposed solution.

In addition to the conclusions drawn from the used benchmarks, a closer analysis of the employed bus systems can also help to estimate the smallest supported SFR: For example, the synchronization-operation buffer (SB) proposed in [27], managing all synchronization constructs locally at the shared memory, is reachable for the PEs through a high-bandwidth network-on-chip (NoC). A latency of at least 30 cycles for NoC transactions is stated; for requesting and getting notified about the availability of a lock, at least two transactions are required, pushing the affordable SFR size far above the approximately one hundred cycles and below targeted by us.

Due to the similarities in the targeted systems, it can be assumed that [33] aims at similarly sized SFRs like ours. Results are reported as number and types of accesses to the synchronization hardware for a wide range of synchronization primitives. The *barrier* and *mutex* primitives that we discuss in detail are estimated to take only two bus transactions or six cycles (measured at the memory bus); we further reduce bus transactions to a single one and latency to four cycles.

Synchronization Hardware Complexity. Keeping overall circuit area and complexity as small as possible is crucial for systems that ought to be employed for parallel NTC, as explained in Section 1.

The cache-alike SB presented in [27] is capable of flexibly managing a large number of synchronization constructs; however, a considerable circuit area is required for each entry to store all required information and the logic that performs single-cycle hit-detection. The need to check every memory access for a matching address causes the activity of the SB to be much higher than the frequency of synchronization points in a given application would require.

The concepts presented in [34] and [29] are based on snoop devices at the memory, or system bus; [29] is of particular interest as it distributes the synchronization-management over one controller per PE that each hosts a locking queue for the respective variables of interest. This concept sounds very appealing from a hardware complexity point of view; however, an important aspect not covered in [29] limits the applicability. In the usual (and desirable) case where the system bus of a CMP can handle multiple transactions at once, each local synchronization controller must have global visibility of the bus and be able to *parallelly* check the maximum number of concurrent transactions against *any* of its monitored locking variables. Building a single such device in a slim way is already challenging; replicating it for every PE further aggravates this issue.

Based on the available knowledge about the amount of information that each barrier filter in [17] needs to store, the complexity of such a module can be estimated to be slightly higher than for our proposed hardware barrier modules. We follow the presented concept and reduce all address-related housekeeping overhead and restrictions by assigning each barrier module a fixed address and providing PE-parallel access.

This work is most comparable to the hardware synchronizer (HWS) proposed in [33] that is as well connected as a shared peripheral, enabling synchronization primitives through appropriate programming of an array of atomic counters and compare registers. While the absolute and relative circuit complexity of the added hardware is comparable to our proposed solution, its power consumption of tens of milliwatt is in the range of our targeted total system consumption and, therefore, prohibitive [13]. We follow the same approach of a memory-mapped shared peripheral with comparable circuit area but further reduce the cost of synchronization primitives, state them in terms of cycles and energy and quantize the achieved system-wide energy savings while maintaining the overall power envelope of a few tens of milliwatts. We reduce the hardware complexity and remove the burden of configuring and mapping the atomic counters to synchronization primitives by providing native and low-cost hardware support for such while maintaining general-purpose signaling.

The concepts presented in [19], [35] principally match ours since shared, dedicated registers are used to represent the state of synchronization primitives at low hardware cost. Nevertheless, many implementation- and integration-related aspects are left undiscussed, and either only locks [19] or only barriers [35] are natively supported. While a hardware lock can be employed to realize a barrier, the cycle cost compared to a hardware barrier is clearly prohibitive for the small SFR sizes that we target. While our solution also features general-purpose atomic PE-to-PE signaling, the most important feature is native hardware support for the most commonly used synchronization primitives as well as concurrent access to them at very small hardware overhead. We extend the aforementioned concept to also satisfy these requirements.

PM and Signaling Mechanism. With the exception of [16], [17], [27], where no explicit statements are made, all references implement or at least suggest idle waiting for PEs that are blocked at synchronization points. The majority of the works with a focus on idle waiting proposes interrupt-based mechanisms [19], [33], [34], [35]. Even basic interrupt and PM support allows avoiding busy-waiting with the help of software synchronization primitives that only require all contestants to become active after updating shared atomic variables. However, the context changes required for interrupt handling naturally incur software overheads; the issues associated with concurrent lock-acquire attempts and the cost and scaling behavior of synchronization primitives remain. As a result,

1. Loop 3 is omitted as it is a fully vectorizable matrix-multiplication, and synchronization is consequently only required between full kernel iterations.

the handling of a single synchronization point can take over a hundred cycles even with less than ten involved PEs as our experiments in Section 6.3 show. Our approach of event-based signaling is supported in [33]; design details on PM and idle-waiting are omitted. We combine event-based signaling with the concept of power-managing idle cores through clock gating as in [18], [28]. Our proposed hardware features lower power and area compared to all three references ([18], [28], [33]), making it suitable for the targeted microcontroller-class systems.

[17], [29] introduce the idea of *stalling* PEs that may not continue by means of absent replies to requests on their data- or instruction ports. As this approach favorably allows handling the check and decision for continuation in one operation as well as PE-externally and centrally, we follow and extend it. We combine the concepts of stalling cores that are blocked at synchronization points and event-based signaling, and tightly couple those with per-core fine-grain clock gating to allow the handling of synchronization primitives in less than ten active PE cycles.

3 Background

3.1 Tightly-Coupled Processors Clusters

This work aims at accelerating synchronization tasks in the context of *tightly memory-coupled multicore processing clusters*, such as Rigel [36], STHORM [37], PULP [8], [13], or even general-purpose GPUs (GP-GPUs). In these architectures, either simple PEs or DSP-capable processors share a multi-banked, word-level interleaved scratchpad memory, seized in the order of tens of kilobytes (typically 32 to 128) [12], [13], [14]. More specifically, we target this architectural template applied to IoT applications, where the primary goal is to improve energy efficiency rather than performance, by joining parallelization with low-voltage operation.

In this context, upper-bounding the core count to 16 and the number of memory banks to 32 (and 128 kByte) in a single cluster allows the usage of rather simple, but fast (in terms of latency) interconnect and bus systems, while still providing a computational performance of several GOPS. Other architectures like GP-GPUs scale the number of PEs to up to 32 with a two-cycle latency shared-memory, prioritizing performance over energy efficiency. In the context of IoT end-nodes, a single-cluster 8-cores configuration demonstrated to provide a favorable trade-off between performance and energy efficiency for several application domains (e.g., [13], [38], [39]). For this reason, although we explore the scalability of the proposed synchronization hardware (SCU) for a range of configurations (i.e., 2 to 16 cores), we set the benchmarking focus of this work to an 8-cores cluster.

3.2 Relevance of Fine-Grain Synchronization Support

The memory constraints of IoT end-nodes limit the size of the working set that can be present in the tightly coupled memory at a given time. To avoid access to outer memory levels and preserve performance, applications with an extensive working set must employ techniques such as *data tiling* coupled with double-buffering direct memory access unit (DMA)-supported mechanisms [40], [41]. The orchestration of tiling introduces additional dimensions to the iteration space of the original algorithm, which map to supplementary inner loops, iterating on smaller bounds (i.e., the tile size). Consequently, synchronization moves to a finer level of granularity than the original algorithm, and the number of synchronization points increases by the number of tiles [42].

Another important aspect that contributes to the importance of fine-grain synchronization support is to allow efficient parallelization of kernels that inherently exhibit small SFRs. We list multiple examples of such kernels in Section 6.4, which can only be efficiently executed in a parallel fashion when the target platform supports the handling of typical synchronization tasks in roughly ten cycles. The adoption of task-level parallelism, combined with software pipelining, can work around these issues; however, this methodology poses a significant limitation in terms of programming flexibility and achievable computation latency. Furthermore, it requires the constant availability of tasks that can be independently scheduled and completely occupy the idle PEs.

4 Architecture

This section starts with an introduction to the high-level design principles and architecture of the hosting multiprocessor cluster before explaining the details of the SCU architecture, its integration into the cluster as well as analyzing its scalability.

Since a significant portion of the overall energy at the OEP is spent through static power consumption, circuit area must be kept small as it is directly linked to the former. As a direct consequence, a beneficial adoption of parallel NTC minimizes the circuit overhead to provide PEs with access to shared resources (such as memories or peripherals) and to communicate and synchronize with each other. Apart from static power, complex synchronization hardware also causes a significant amount of extra active power (relative to our small PEs) that could eventually diminish any energy savings gained from accelerated computations.

Even simple interconnect systems or caches can quickly become comparable to or even exceed the area of the PEs, which is illustrated through the area breakdown in [13]. As a result of this constraint, features such as multi-level data caches with the attached burden of coherency management, memory management unit (MMUs), nested vectorized interrupt support, or network-like communication systems are unaffordable. The absence of these blocks, in turn, prohibits the usage in a control-centric OS-like fashion with virtual memory support but favors the employment of the clusters as programmable many-core accelerator (PMCAs) to execute computation-centric kernels with regular program flow and physical memory addressing [43].

4.1 Multiprocessor Cluster

As a basis for our proposed synchronization solution and adhering to the design principles outlined above, we use the open-source multiprocessor cluster of the PULP project [13], [15], matching the targeted deeply embedded, data cache-less tightly-memory coupled system type. The cluster is depicted in Fig. 1 and designed around a configurable number (typically up to 16) of low-cost in-order RISC-V cores. To greatly accelerate the execution of the targeted DSP-centric processing loads, they feature several extensions to the base

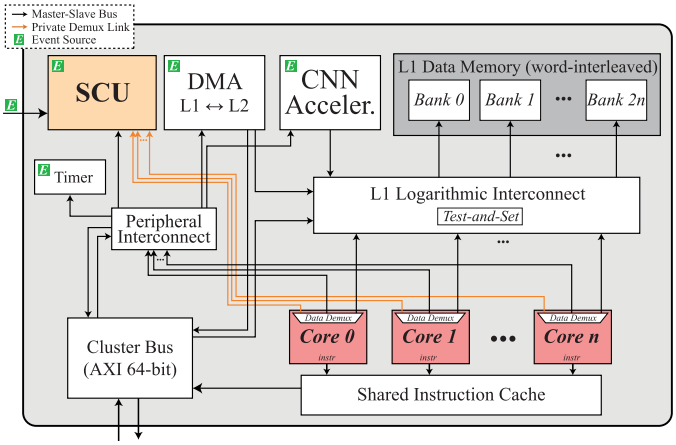

Fig. 1. Multicore cluster, incorporating the proposed synchronization and communication unit (SCU). The private links between cores and SCU are highlighted in orange. The Test-and-Set feature of the L1 logarithmic interconnect is employed in the baseline synchronization implementations against which we compare our proposed solution.

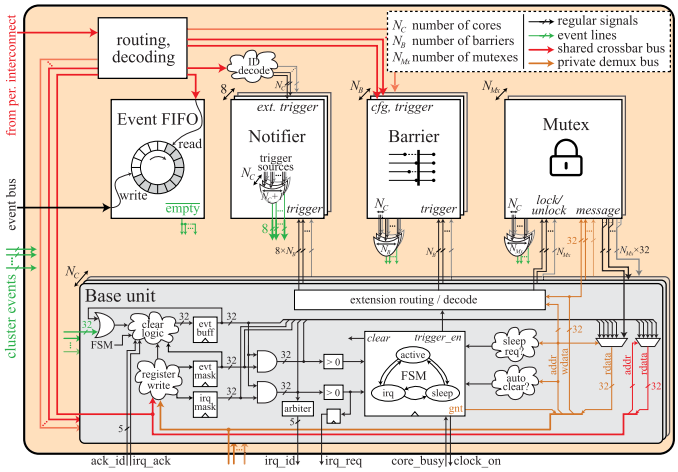

Fig. 2. Simplified overall architecture of the SCU, including the available extensions, and detailed architecture of the base units. Signals on the bottom connect to the cores, signals on the left to the cluster, and higher hierarchies. The private demux busses, drawn in bold orange, are integral to the proposed synchronization mechanism.

instruction set [44] from which a wide range of applications benefit. Specialized PEs such as neural network accelerators can additionally be included in the cluster to cope with more specific tasks requiring very high processing throughput [45].

All PEs share a single-cycle accessible L1 tightly-coupled data memory (TCDM), composed of word-interleaved single-port SRAM macros. A banking factor of two is used to reduce the number of contentions between PEs (i.e., the number of banks is twice the number of PEs). Data transfers between the size-limited TCDM and the L2 memory with larger capacity is facilitated by a tightly-coupled DMA connected to the L1 memory like any other PE. Access routing and arbitration between all PEs and the TCDM banks are handled by a low-latency logarithmic interconnect (LINT), allowing each TCDM bank to be accessed by a PE in every cycle. If multiple PEs access the same bank in the same cycle, sequential access is granted in a round-robin fashion (n simultaneous accesses to the same bank take n clock cycles to complete). All cores fetch their instructions from a hybrid private-shared instruction cache which has – in addition to the DMA – access to the 64-bit AXI cluster bus that connects to the rest of the communication and memory system of a system-on-chip (SoC) that hosts the cluster.

In addition to the TCDM interconnect, the cluster features a peripheral subsystem with dedicated LINT. It allows the specialized PEs to be programmed and controlled through memory-mapped configuration ports, and to connect further peripherals such as timers and the SCU. A master port to the cluster bus allows the RISC-V cores to access all cluster-external address space.

4.1.1 Test-and-Set Atomic Memory Access

Besides routing and arbitrating requests, the LINT provides basic and universal atomic memory access to the whole TCDM address space in the form of TAS. Atomic accesses are signaled by setting an address bit outside of the L1 address space; the LINT checks it upon read-access. The currently stored value is returned to the requesting core (or to the elected one in case of multiple contending requests) and -1 written back to memory in the next cycle before any other core gets its request granted. We use the synchronization primitives based on this feature as a strong baseline (TAS transactions take just three cycles) against which we compare our proposed solution.

4.2 SCU

A beneficial implementation of hardware that accelerates synchronization requires minimal changes to the hosting system (e.g., no profound modifications or extensions to the PE data path) and, wherever possible, reuses existing infrastructure. In this way, both PEs and synchronization hardware can be easily exchanged should one not fit the usage scenario of the hosting cluster. As the central synchronization hardware is best aware of the set of PEs that is waiting at synchronization points, it is the best place to control fine-grain PM on a per-PE basis. Consequently, to perform power-managing on any idle system component, one of the most important parallel NTC principles, also the synchronization hardware itself must be designed in an appropriate way; e.g., it should not waste active power during phases where all PEs are busy, and none is involved in any synchronization action.

4.2.1 SCU Base Unit

Fig. 2 depicts a high-level overview of the SCU architecture, with a deeper focus on the SCU base unit. The base unit is instantiated once per RISC-V core. It provides the fundamental functionality of the SCU, i.e., event and wait-state management, as well as fine-grain PM through direct control of the clock-enable signal of the corresponding core. The design is based on 32 level-sensitive *event lines* (per core) that are connected to associated *event sources*. In a typical usage scenario, a limited number of the event sources are located outside of the SCU (e.g., specialized PEs or cluster-internal peripherals), while the remaining ones are responsible for core-to-core signaling and generated within the SCU by so-called SCU *extensions*.

Event lines are stored into a register called *event buffer*, which is maskable through the *event mask* register. Basic

interrupt support is also provided to handle exceptions and other irregular events; an *interrupt mask* register allows selective enabling and disabling of event lines to trigger hardware interrupts. The central finite state machine (FSM) orchestrates all control flow and includes the three states, *active*, *sleep*, and *interrupt-handling*. The main inputs used to evaluate state transitions are pending events or interrupts, the core busy-status as well as sleep and buffer-clear requests.

4.2.2 SCU Extensions

SCU *extensions* are responsible for core-to-core signaling; generally, they generate core-specific events that allow a subset to continue execution. All extensions have trigger and configuration signals connected to each base unit; as with the base units, their associated functionality is available through memory-mapped addresses. The four available types of extensions are depicted in Fig. 2 and detailed in the following.

Notifier. This extension provides general-purpose, any-to-any matrix-style core-to-core signaling. Each core can trigger one of the eight notifier events for any subset of cores (including itself). For write-triggered events, the write data are used as a target-core mask; for read-triggered events, a dedicated register in each SCU base unit holds the target mask. An all-zero value causes a broadcast notifier to all cores in both cases. This extension is used in the TAS-based variants of the synchronization primitives that we profile and use in Sections 6.3 and 6.4, respectively.

Barrier. Allows a configurable *target* subset of cores to continue execution only after a (possibly different) *worker* subset has reached a specific point in the program. The extension contains a status register that keeps track of each core that has already arrived at the barrier; this is signaled by reading or writing from or to specific addresses (in the address space of the base units which generate trigger signals towards the barrier extensions). Depending on the core that caused the access, the matching bit in the status register is set. Once the status register matches the configured *worker* subset, an event is generated for all cores that are activated in the *target* subset, allowing those to uninterruptedly idle-wait at the barrier until their condition for continuation is met.

Mutex. Represents an object that can only be owned or locked by one core at a time and, therefore, directly supports synchronization primitives that require mutual exclusivity such as, e.g., mutual exclusive code sections. Try-locks are, similar to the barrier extension, signaled by reading from a specific address. The mutex extension keeps track of all pending lock requests and elects one core by sending an event to only that one. The elected core must write to the same address once it releases the mutex, causing the extension to wake up another waiting core (if any). The election among waiting cores can either be done by core-id or in a round-robin fashion. We chose to elect by core-id in favor of lower area and simplicity; starvation is prevented in the programming model by always combining a critical section with a subsequent barrier. Other arbitration schemes can be easily included since a separate module with a canonical interface is responsible for said election.

Event FIFO. The SCU includes the event FIFO extension to react to (relatively slow) cluster-external event sources. It allows handling of up to 256 cluster-external event sources that can be triggered by, e.g., chip-level peripherals or higher-level control cores, as can be found in modern SoCs. The external events are sequentially received over a simple request/grant-based asynchronous 8-bit event bus and stored into the FIFO. As long as there is at least one event present, an event line associated with the FIFO is asserted that is connected to all SCU base units. In a typical use-case, the event line triggers an interrupt handler on one core that then pops the events from the FIFO and processes them.

For the targeted parallel programming models such as, e.g., OpenMP [11], the barrier and mutex extensions are the most important ones as they provide hardware support for the fundamental *parallel sections* and *critical sections* programming primitives. The number of barrier and mutex extension instances, N_B and N_{Mx}, can be independently set at design time to, e.g., support every team-building variant. As every core can only wait at one barrier or try-lock one mutex at a time, the corresponding core-specific events of all instances are combined into a single event per extension type and core.

4.3 SCU Integration

As a part of the peripheral subsystem introduced in Section 4.1, the SCU is connected as an additional shared, memory-mapped peripheral to the corresponding LINT. However, this single-port solution has the major drawbacks of non-deterministic core-to-SCU access latency and sequentialized accesses whenever more than one core wants to access the SCU in a given cycle. Since the limitations (in terms of performance, energy-efficiency, and scalability) of synchronization primitives that are realized with classic atomic memory access mostly result from sequential access to shared variables, parallel access to the SCU base units and extensions responsible for core-to-core signaling is paramount.

Therefore, we use additional, private one-to-one buses between each core and its corresponding SCU base unit, shown in orange in Figs. 1 and 2. A demux at the data port of each core selects between the L1 TCDM and peripheral LINTs and the private SCU link. The one-to-one correspondence between the cores and SCU base units allows to alias their address space, thereby simplifying synchronization primitives by removing core-id dependent address calculations. As the paths through the LINTs, the private core-SCU links are purely combinational and allow for single-cycle access. As we demonstrate in Section 6.3, the fully-parallel access to the SCU can even result in constant cycle cost for specific synchronization primitives, independently from the number of involved cores – a very favorable scaling property compared to classical atomic-memory based approaches.

In order to retain a global address space (for, e.g., debugging purposes), all base units are as well accessible from every core and from outside the cluster through the peripheral LINT. All power-managing functionality of the SCU base units (resulting in a core idle-waiting for an event) is not implemented for this access method as it would disturb the inter-core control flow.

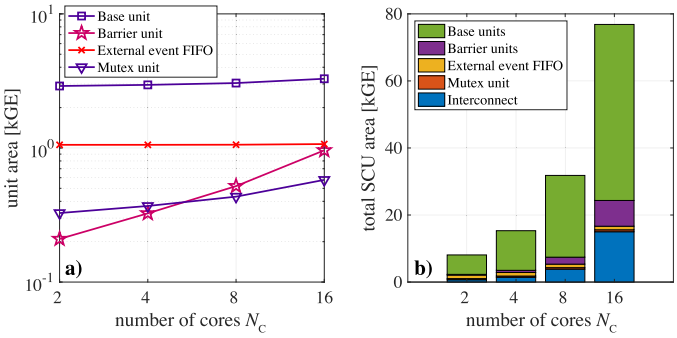

Fig. 3. Scaling of the circuit area (in gate-equivalents (GE)) for the base unit and the available extensions (a) and for the overall SCU (b). The largest share of the overall area is in all configurations attributed to the base units.

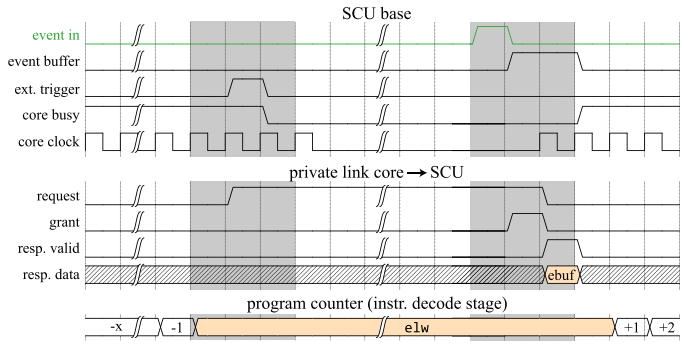

Fig. 4. Interfacing of the SCU with a RISC-V core and corresponding timing. Shaded intervals correspond to transitions to (left) and from (right) sleep state, respectively. The core executes a single instruction during the whole process. Note the absence of the grant (controlled by the SCU) until one cycle after an incoming event.

4.4 SCU Scalability

Fig. 3 shows both the total SCU area as well as the area of the individual sub-units and extensions in relation to the number of cores N_C. For the total area, a typical configuration with the number of barrier extensions $N_B = N_C/2$, and the number of mutex extensions $N_{Mx} = 1$, is shown. The plots show post-synthesis numbers; we used the same 22 nm CMOS process as for the multicore cluster, which hosts the SCU. Design synthesis was done in the slow-slow process corner, at 0.72 V supply voltage, a temperature of 125 °C, and with a 500 MHz timing constraint.[2] We restrict N_C to a maximum of 16, matching the typical scalability limits of the targeted cluster-based architecture. An analysis of the slopes in the double-logarithmic sub-unit area plot in Fig. 3a) reveals a mildly super-linear scaling for the barrier extensions and sub-linear or constant scaling for the others. The overall SCU area favorably scales sub-linearly up to the typically used configuration of $N_C = 8$ and mildly super-linearly if N_C is further increased to the maximum configuration. The area contributions of the SCU base units and barrier extensions dominate in all configurations. However, the share of the SCU-internal interconnect logic to correctly route all $N_C + 1$ slave ports to the respectively connected sub-units becomes as well significant for the two largest configurations.

5 Single-Instruction Synchronization

With our goal of aggressively reducing synchronization overhead in mind, we propose a scheme that allows handling common synchronization tasks with the execution of a single instruction in each involved RISC-V core. To achieve this, we extensively leverage the dedicated link between each core and the corresponding SCU base unit, the associated aliased address space of 1 Kibit, and the possibility to stall a core by not granting read accesses made over the private links.

A fundamental aspect of our proposed solution is that whether a core can continue at a synchronization point is always signaled through events that are generated inside the SCU by one of the extensions. Each involved core idle-waits for the appropriate event to occur; the corresponding event line has to be activated in the event mask. Waiting is universally initiated by executing the `elw` instruction that we added to the extensible RISC-V ISA. The mnemonic stands for *event-load-word*; the instruction is identical to the regular *load-word* (`lw`) of the base ISA with the exception of an altered opcode such that the core controller can distinguish them. Whenever a core executes `elw` with an address that requests waiting for an event, the SCU will block the resulting transaction on the private link by not asserting the grant signal (given that no events are currently registered in the event buffer). This process is depicted in the left shaded part of Fig. 4, which shows the details of a private core-SCU link and the most important signals of the corresponding SCU base unit. Due to the in-order nature of the employed cores, the stall at the data port propagates through the core pipeline. The `elw` opcode causes the core controller to release the busy signal once any prior multi-cycle instructions have been executed; consecutively, the SCU power-manages the requesting core by lowering its clock-enable signal. Depending on the address of the discussed read transaction, an extension in the SCU gets simultaneously triggered (e.g., try-lock a mutex, set the status bit in a barrier, send a notifier event). The extension triggering is controlled by the FSM in the SCU base unit to ensure that per `elw`-transaction triggering is only done once.

The right shaded part of Fig. 4 shows the process of a core waking up and continuing execution, initiated by an incoming event. The event is present in the event buffer in the consecutive cycle, causing the SCU both to re-enable the core clock and assert the grant for the still-pending read request. Another cycle later, the response channel of the private link is used to deliver additional information to the requesting core: Often, the content of the event buffer is sent such that in the case of multiple activated event lines, the core can immediately evaluate the reason for the return from sleep. More interestingly, however, the response channel can also be used to pass extension-specific data. In the example of the mutex extension, it allows the unlocking core, done with a write transaction, to intrinsically pass a 32-bit message to the core that locks the mutex next. Once the response data is consumed, the event buffer can – again controlled through the address of the `elw` – automatically be cleared, freeing cores from yet another common task, especially for the usual case of waiting for a single event line only.

2. Even though our multicore clusters are usually constrained to slower clock frequencies (see Section 6.2), we chose this constraint to also verify the suitability of the SCU for systems that target slightly higher clock speeds.

Fig. 4 shows the process of entering and leaving a wait state with an address that results in both triggering an extension and automatically clearing the buffer. This example highlights the small amount of only six cycles of active core clock for handling a synchronization point (excluding the possibly required address calculation for `elw`). For cases where an event occurs before or during a wait request (e.g., when the last core arrives at a barrier), the grant is immediately given, and no power-managing is done to not waste any cycles. The required changes in the core to support the described, 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 at the data port.

5.1 Fused Interrupt Handling

The targeted type of clusters is primarily meant for executing kernels with a regular program flow, the synchronization of which can be purely handled with events and idle waiting. Still, interrupts are often required to, e.g., handle data exceptions or react to other spontaneous, irregular, but important events that require an immediate change of program flow. A dedicated FSM state and an additional mask register in each SCU base unit are employed to fulfill said requirement; the event buffer is shared between both masks for increased area and energy efficiency. The few cases where a core needs to be sensitive to the same event source both as an interrupt and event trigger can be handled with a combination of an interrupt handler and a self-triggering notifier event. Two dedicated request/identifier pairs connect each core and the corresponding SCU base unit for both signaling and clearing interrupts, respectively. The SCU arbitrates one of the pending interrupts to the core, which, in turn, acknowledges the processing of the arbitrated interrupt upon entering the respective handler. Similar to the auto-clearing capability when waking up through events, the bit corresponding to the called interrupt handler is cleared in the event buffer to reduce management overhead in the handlers.

Should an activated event line trigger during an interrupt handler, regular program flow is immediately continued after its termination. In the usual case, the FSM transits to sleep again and awaits further incoming events and interrupts. After termination of the interrupt handler, the `elw` instruction responsible for the original wait state is re-executed, allowing the SCU to detect said termination and power-manage the core again. In such cases, the FSM takes care of inhibiting erroneous extension re-triggering upon the repeated sleep request after interrupt handling.

6 Experimental Results

To demonstrate the effectiveness of the proposed event-based, hardware-supported synchronization concept, we present two types of experimental results in this section. We first show the theoretically achievable improvements through synthetic benchmarks where all experiment parameters can be controlled. We successively analyze the performance and energy efficiency improvements that are observed when executing a range of applications that the targeted class of deeply embedded CMPs is typically used for. As the leading principle and motivation for this work is to reduce the energy that the cluster consumes for a given workload or task, we report not only the total cluster energy but also power and execution time in all cases to provide insight into how the energy reduction is achieved. We additionally provide power breakdowns into the main contributors to highlight the importance of fine-grain power management in the form of clock gating. Finally, an analysis of both the amount of total and active cycles spent on synchronization shows how the proposed solution drastically reduces synchronization-related overhead.

6.1 Baseline

As a baseline, we use purely software-based implementations of synchronization primitives that employ spin-locks on TAS-protected variables in the L1 TCDM with the help of the TAS-feature of the logarithmic interconnect. This concept is inherently disadvantageous not only from an energy-efficiency viewpoint due to the wasted energy for every failing lock-acquire attempt but also from a performance point-of-view as the concurrent attempts can put high loads on the TCDM interconnect and cause contentions on the shared memory system. Additionally, as every contestant has to acquire the lock sequentially, the cost of synchronization primitives in terms of cycles is even in the best case lower bounded by the product of memory access latency and the number of involved PEs, therefore growing with the number of contestants. Consequently, most modern multiprocessor systems feature hardware support for idle waiting, thereby avoiding the active and continuous polling of synchronization variables. This policy removes a very significant amount of core activity and memory accesses and, therefore, wasted energy and memory bandwidth.

To take account of this, we include a second TAS-based solution in our comparisons where cores that do not succeed in acquiring a synchronization variable (e.g., to update the barrier status stored in the variable) are put to sleep with the help of the SCU (by idle waiting on an event as described in Section 5). Whenever the current owner updates or releases the variable, it also uses an SCU notifier broadcast event to wake up all the remaining, sleeping cores, which will then again try to acquire the synchronization variable. While the described synchronization mechanism can also be realized with similar solutions for idle waiting and notifying cores that may be found in comparable systems, it – in our case – already benefits from the low latencies for notifiers and idle state handling that are enabled through the SCU. In the following, the purely spin-lock based implementations of synchronization primitives will be referred to as *SW* and the idle-waiting extended versions as *TAS*.

6.2 Experimental Setup and Methodology

All experiments were carried out on an eight-core implementation of the multicore cluster of Fig. 1. It features 64 kByte L1 TCDM and eight kByte of shared instruction cache; the SCU contains four barrier and one mutex extensions. For all cycle-based results, an RTL description of the cluster and cycle-exact simulations were used. A range of observation tools both in the RTL description of the RISC-V

cores as well as in the testbench is employed to obtain more detailed insight than total execution time only:

- Per-core performance counters record the number of executed instructions, active cycles, stall cycles at both data and instruction ports, and the like. Due to area considerations, only one general-purpose counter per core is kept in the synthesized version.
- Non-synthesizable, per-core instruction tracers that allow detailed analysis of the executed applications and benchmarks.
- Overall execution time is measured with the help of a cluster-global timer that is part of the cluster peripherals and kept during synthesis.

The timer is activated only during periods where the actual benchmark (synthetic or application) is executed to exclude, e.g., initialization and boot periods. The enable signal of the timer is monitored in the testbench, and the timestamps of rising and falling edges recorded to a text file, allowing the identification of the relevant portions of the instruction traces.

6.2.1 Physical Implementation

The majority of the previous works employs behavioral models of the individual system components (PEs, interconnect, memories, synchronization hardware) written in higher-level languages and instruction- or transaction-level simulators such as, e.g., GRAPES in [27], MPARM in [28], [34], or M5 in [16], [29]. While this approach enables the simulation of complex and large-scale architectures in reasonable times, it has the drawback of reduced accuracy for the figures of interest, performance (or execution time) and power, when compared to cycle-exact simulations based on synthesizable modules captured in a hardware description language (HDL) or gate-level implementations. The loss in accuracy may be acceptable for evaluating the performance of synchronization solutions for task-level parallelism or the support of transactional memory (TM) [28]. For our goal of enabling fine-grain parallelism with few tens or hundreds of cycles between synchronization points, however, cycle-level accuracy is required to reliably evaluate the effects of different solutions with an increasing degree of hardware support (from atomic memory access to full synchronization primitives). For example, slight differences in the arrival instants of PEs at synchronization points due to cache misses or small workload imbalances can have a large impact on the subsequently caused contention during lock acquire trials.

We consequently use a cycle-exact RTL implementation of the whole cluster to measure performance and a post-layout, fabrication-ready[3] physical implementation in a current 22 nm CMOS process as a basis for the most important figure of merit, the total system energy with and without our proposed solution for fine-grain parallelism. An important rationale for the post-layout implementation stage is that it considers the clock distribution network, which typically consumes a significant share of the overall dynamic power of synchronous digital circuits, yet often gets neglected in energy analyses. Furthermore, the efficacy of fine-grain PM in the form of silencing parts of the clock network, a central part of our concept, can only be shown in this way.

The RTL model of the cluster was synthesized and a placed and routed physical implementation created; both steps were done with a 350 MHz timing constraint and in the slow-slow process corner at 0.72 V supply voltage and a temperature of 125 °C.[4] The resulting module measures 1.4 mm × 0.8 mm with pre-placed SRAM macros for the L1 TCDM; the SCU accounts for less than 2 percent of the total circuit area. The synthetic benchmarks and the applications were run on the resulting gate-level netlist and VCD activity files recorded during the benchmarking periods for every electrical net in the cluster. Again, the enable signal of the cluster-global timer is used to start and stop the activity recording. The subsequent hierarchical power analysis was done in the typical-typical process corner at 0.8 V supply voltage and 25 °C, allowing us to report not only the total power and energy but also the respective breakdowns to analyze the contributions of the individual cluster building blocks. The reported power and energy results correspond to a cluster operating frequency of 350 MHz.

6.3 Synthetic Benchmarks

We start our analysis by quantifying the cost in terms of cycles and energy for executing barriers and critical sections, the two synchronization primitives that are most commonly used in the targeted parallel programming models. We compare the hardware variants featured in the SCU with the purely spin-lock based as well as with the idle-wait extended baseline variants as described in Section 6.1. To highlight the favorable scaling behavior (with respect to the number of participating cores) of the SCU, we provide the quantification for two, four, and eight cores even though the cluster is mainly designed for execution on all eight cores. We let the involved cores execute a loop eight times that contains the respective primitive 32 times; the first loop iteration is used for cache-warming and excluded from benchmarking. The resulting cycle costs are obtained by dividing the total cluster-wide cycle count by the product of cache-hot loop iterations and primitive executions per loop iteration. A typical use-case for critical sections in the type of targeted systems is the placement at the end of an SFR to perform small control tasks like updating a shared variable by all worker cores. Consequently, the critical section is usually very short (up to ten cycles only), a circumstance that we consider in our experiments.

We compiled the synthetic benchmarks with an extended version of the riscv-gcc 7.1.1 toolchain that supports the `elw` instruction, using the `-O3` flag. To compute the absolute cost figures reported in Table 1 and the relative energy overhead, we also measure the power during the execution of 512 `nop` instructions on a varying number of cores. While the choice of the `nop` instruction may intuitively not be a suitable representation of actual processing loads, the resulting relation between

3. A SoC that contains the cluster has been fabricated and functionally verified. The same applies to multiple application-specific integrated circuit (ASICs) that contain very similar clusters in various technology nodes [12], [13], [14].

4. Timing was verified with all permutations of the slow-slow/fast-fast process corners, 0.72 V/0.88 V supply voltage, and temperatures of -40 °C/125 °C.

TABLE 1
Cost of Synchronization Primitives in Terms of Cycles and Energy

N_C (core count)		cycles			energy [nJ]		
		2	4	8	2	4	8
Barrier	SCU	6	6	6	0.1	0.1	0.1
	TAS	52	91	176	0.8	1.7	4.3
	SW	47	87	176	0.8	1.8	4.7
5-cycle crit. sect.	SCU	12	23	44	0.2	0.3	0.6
	TAS	25	39	69	0.4	0.7	1.6
	SW	12	25	72	0.2	0.5	1.6
10-cycle crit. sect.	SCU	13	24	50	0.2	0.3	0.7
	TAS	26	50	89	0.4	0.9	2.1
	SW	13	26	55	0.2	0.6	1.5

The reported figures reflect the costs for the whole cluster, i.e., the number of cycles and energy used for all cores to execute a primitive.

SFR size and overhead still is a very reasonable estimate for the behavior that results when executing actual applications, as our analysis in Section 6.4 shows.

6.3.1 Barriers

When considering the pure primitive cost, the SCU barrier requires between 7.8× (2 cores, SW) and 29× (8 cores, SW and TAS) fewer cycles, as can be seen in Table 1. The gap widens when considering energy where the reduction ranges between 10× (2 cores, SW and TAS) and 38× (8 cores, TAS) or 41× (8 cores, SW), respectively. With higher core counts, the TAS version shows slightly lower energy compared to the SW version thanks to the idle-wait behavior. For the SCU variant, not surprisingly, the fully parallel access to the barrier extension makes the cycle cost independent from the number of cores and incurs minimal additional energy when increasing the number of participants. As a result, the SCU supported barrier is especially favorable when a task is parallelized on all eight cores, which is the intended way of using the cluster.

Figs. 5a and 5d illustrate the raw barrier cost in relation to a preceding SFR of varying size by showing the relative overhead for executing the barrier in terms of cycles and energy. While significant overhead reductions can be observed with SFRs of up to around 1000 cycles and eight active cores, the graphs reveal another even more important characteristic of the SCU barrier: With a typical constraint of allowing up to 10 percent of synchronization overhead, the SCU drastically reduces the smallest allowable SFR. The cycle-related relative minimum SFR reductions are (mathematically) identical to those for the raw primitive-cost; the energy-related reductions show only insignificant differences compared to the corresponding raw cost ratios. Besides the relative overhead reductions, the absolute size of the smallest allowable SFR is important, which is with the SCU barrier for both cycle and energy overhead and all core counts below 100 cycles and therefore matches the in Section 2 stated requirement for fine-grain synchronization. This is in stark contrast to the overheads resulting from the TAS and SW variants, where both cycle and energy-based SFR sizes must be at least multiple hundreds of cycles when considering two or four participating cores. The energy-related minimum SFR with all eight cores participating, representing the most important case, is with 1622 cycles (TAS) and 1771 cycles (SW), two orders of magnitude higher than the corresponding SFR size when employing the SCU barrier (42 cycles). This outcome poses a strong limitation on the range of applications that can be efficiently parallelized on the targeted architecture.

6.3.2 Critical Sections

Compared to barriers, the critical or mutual exclusive section synchronization primitive can be more easily implemented with basic atomic memory access. The ability to enter the critical section can be managed with a single TAS-protected variable that needs to be tested upon entering and written with the test value by the owning core upon exiting. For the TAS-variant of this primitive, we link each access to the synchronization variable to the usage of a notifier event to avoid constant testing of the variable by all cores that are waiting to enter the critical section: Any core that fails to enter will idle-wait for the related event. The core that currently executes the critical section triggers the event upon exiting, causing all queued cores to quickly wake up and retest the TAS-variable, with all but the elected one immediately going back to sleep afterwards.

In the SCU-based implementation, we execute `elw` with an address mapped to the mutex extension, which elects one core for which continuation is enabled through the generation of a core-specific event. All others idle-wait at the mutex load until they are elected. Similar to the variants based on TAS-variables, a write to the mutex by the previously elected core upon leaving the critical section unlocks the mutex and triggers the election of the next core to enter the section alongside the appropriate event generation. The distinction between locking and unlocking the mutex is done with the access type (read/write) and allows to use the same address for both operations, further reducing the software overhead for the synchronization primitive.

As with barriers, we provide both raw-primitive cost (Table 1) as well as relative overheads in terms of performance and energy (Fig. 5), each, for two different critical section sizes. The latter is necessary since the wait behavior of cores that yet have to enter the critical section greatly differs between the implementations: For the SW variant, waiting cores do not only test the synchronization variable upon another one exiting the critical section but constantly. Consequently, the duration of the critical section has an impact on so-caused parasitic energy. We calculate the cost figures reported in Table 1 as the difference between ideal cycle count and energy and the measured ones. The ideal number of cycles is $T_{ideal} = N_C T_{crit}$ and the ideal energy $E_{ideal} = T_{ideal} P_{comp,1}$ with T_{crit} denoting the length of the critical section, $P_{comp,1}$ the single-core cluster power, and N_C the number of cores that need to execute the critical section.

In relation to the barrier results, the differences between the SCU and the TAS-based variants are considerably smaller: As can be seen in the right half of Fig. 5, for two cores, the minimum SFR size for 10 percent overhead is at most reduced by 2.5× from 232 cycles to 91 cycles when comparing the energy overhead of the TAS and SCU variants. The differences in the

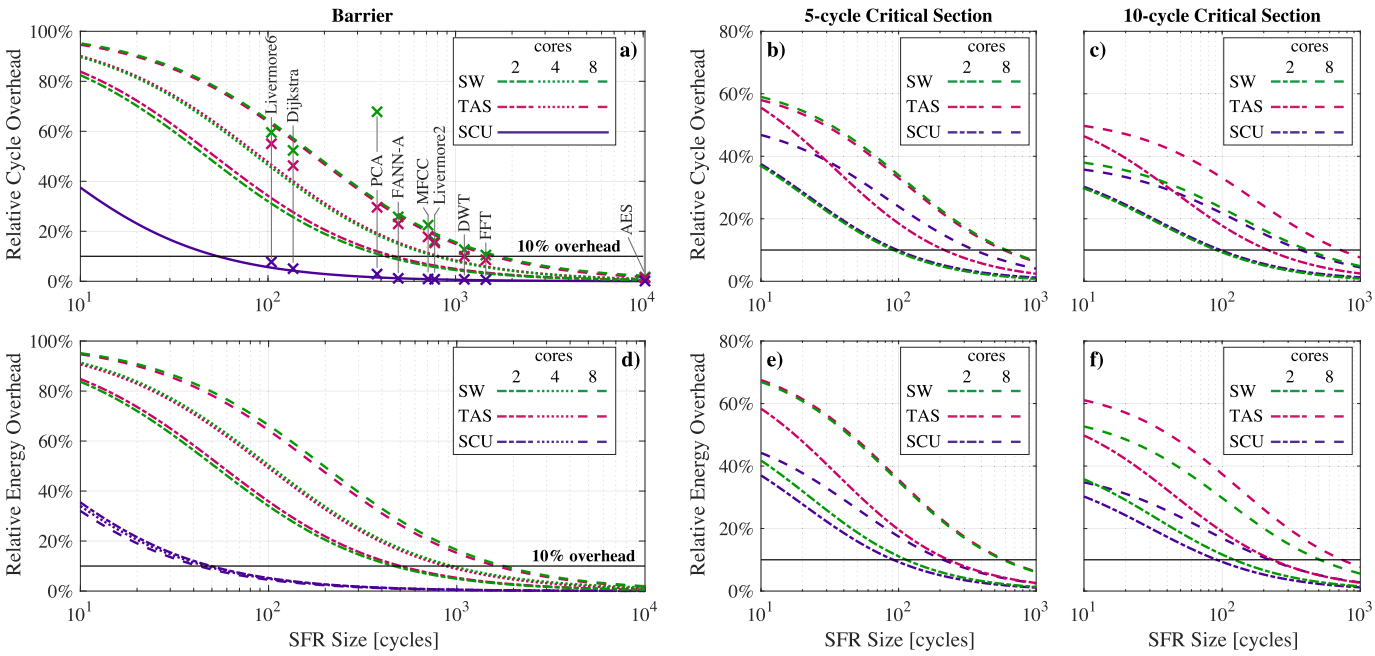

Fig. 5. Relative overhead in terms of cycles (a)-c), top) and energy (d)-f), bottom) versus SFR size for the three realizations of barriers (a), d), left) and critical sections with two lengths (b), c), e), f), right). The markers in a) indicate the relative share of active synchronization cycles for the range of DSP applications discussed in Section 6.4. For critical sections, the graph lines corresponding to four cores have been omitted to improve graph readability. The scaling behavior in terms of overhead versus core count is strictly monotonic (see raw costs in Table 1).

relative cycle overhead are smaller or even non-existent. The picture changes, however, when considering eight participating cores: While the cycle-related differences remain small, the energy-related gap widens. The smallest SFR for 10 percent relative energy overhead is reduced by at least 2.3× (10-cycle critical section, SW to SCU) and up to 3.3× (10-cycle critical section, TAS to SCU). Still, compared to the barrier, the savings achievable with the SCU mutex extension are one order of magnitude lower. The reason for this behavior is twofold: First, a mutex is a much simpler synchronization primitive than a barrier, and second, a TAS-protected variable inherently allows for very efficient implementations. Still, the avoidance of any TCDM accesses when using the SCU results in consistently lower power and energy for all core counts and critical section lengths.

Counterintuitively, the TAS-variant performs both in terms of cycles and energy worse than the straight-forward SW version for all core counts and critical section lengths. This circumstance can be explained by analyzing the software footprint of each implementation variant. With fully inlined functions for entering and leaving critical sections, leaving always requires the execution of a single instruction for the SW and SCU variants and two for the TAS variant. For entering, however, only the SCU variant guarantees a single instruction for all cores. The naive SW variant requires two instructions per locking attempt; the TAS variant can only match this count if the first attempt is successful. For all additional ones, five instructions need to be executed to handle the idle-wait functionality. Conclusively, the TAS variant can reduce the number of lock attempts; however, each attempt is more expensive. For cases where re-election occurs after roughly ten cycles, this can lead to an overall increase in both cycles and energy used for the primitive that outweigh the energy saved with cores that sleep for very short instances only. Hence, the critical section lengths used in our experiments are too short for the TAS variant to show a benefit over the SW one; without the SCU, programmers have to choose the optimal implementation in dependency of T_{crit}. Additionally, the usage of `nop` instructions during the critical section hides a disadvantage of the SW implementation that would show with real applications: The repeated synchronization variable polling by all cores that yet have to enter the critical section puts a significant load on both the TCDM and the associated interconnect that would slow down the execution of any critical section which contains TCDM accesses.

6.4 DSP Applications

After exploring the theoretically achievable improvements with dummy code between synchronization points, we ran actual DSP-centric applications on the multicore cluster, each with the three different implementations of synchronization primitives. The applications are, e.g., in turn, applied in real-world use-cases such as [38], [39]. Compilation was done in the same way as with the synthetic benchmarks from Section 6.3; additionally, the combination of the GCC-flags `-flto` and `-fno-tree-loop-distribute-patterns` that yields best performance (for each application individually) has been determined and applied. To obtain accurate results, each application was run seven times on the RTL model, where the two first iterations are used to warm the instruction cache and are not counted towards any results. All cycle-based results are calculated from the averaged outputs of the observation tools over the last five iterations. For calculating power results, the applications were run in the same manner on the post-layout model with signal activity being recorded during a cache-hot iteration. As with the synthetic experiments, all results reflect running the cluster at 350 MHz.

A short description of each application and its synchronization behavior is as follows: **DWT:** 512-element 1D Haar real-valued 32-bit fixed-point discrete wavelet transform

TABLE 2
Main Properties and Results for the Range of Benchmarked DSP Applications

Name	Domain	Barrier		SFR Size	Energy	Execution Cycles		Synchronization Cycles		IPC
		count	*type*	*[cycles]*	*[µJ]*	*total*	*active (stddev)*	*total*	*active*	
DWT	Signal processing	10	SCU	1.1k	0.7	11.3k	10.8k (155)	0.6k (5.2%)	84 (0.8%)	5.01
			TAS	1.1k	0.8	12.9k	12.7k (63)	1.5k (11.7%)	1.3k (10.0%)	4.65
			SW	1.1k	0.8	12.9k	12.9k (0)	1.6k (12.6%)	1.6k (12.7%)	4.56
Dijkstra	Graph search	238	SCU	122	2.0	33.7k	30.6k (2.9k)	4.6k (13.7%)	1.53k (5.0%)	4.72
			TAS	156	4.0	71.3k	69.1k (0.7k)	34.1k (47.9%)	32.0k (46.3%)	4.48
			SW	130	4.0	64.9k	64.9k (0)	34.0k (52.3%)	34.0k (52.3%)	4.09
AES	Cryptography	4	SCU	10.2k	2.8	41.2k	40.9k (188)	339 (0.8%)	34 (0.1%)	5.84
			TAS	10.2k	2.8	41.6k	41.5k (123)	732 (1.8%)	547 (1.3%)	5.82
			SW	10.2k	2.9	41.6k	41.6k (0)	719 (1.7%)	719 (1.7%)	5.80
Livermore6	Linear recurrence	127	SCU	104	1.1	24.5k	14.0k (6.8k)	11.3k (46.1%)	760 (7.7%)	6.00
			TAS	104	1.7	32.3k	28.1k (3.4k)	19.1k (59.0%)	14.9k (55.0%)	5.25
			SW	105	2.1	32.8k	32.8k (0)	19.6k (59.5%)	19.6k (59.5%)	4.74
Livermore2	Gradient descent	12	SCU	744	0.6	9.2k	9.0k (46)	0.3k (2.8%)	71 (0.8%)	6.67
			TAS	789	0.7	11.3k	11.2k (17)	1.8k (16.1%)	1.7k (15.4%)	5.94
			SW	788	0.8	11.3k	11.3k (0)	1.8k (16.1%)	1.8k (16.1%)	5.84
FFT	Frequency analysis	4	SCU	1.5k	0.5	6.1k	6.0k (73)	203 (3.3%)	39 (0.7%)	5.53
			TAS	1.4k	0.5	6.4k	6.3k (23)	606 (9.5%)	540 (8.6%)	5.40
			SW	1.4k	0.5	6.4k	6.4k (0)	670 (10.5%)	670 (10.5%)	5.34
FANN-A	Machine learning	160	SCU	519	6.9	92.4k	84.0k (2.2k)	9.3k (10.1%)	982 (1.2%)	6.72
			TAS	483	7.7	103.0k	100.3k (0.9k)	25.7k (25.0%)	23.0k (23.0%)	6.48
			SW	482	7.9	103.8k	103.8k (0)	26.7k (25.8%)	26.7k (25.8%)	6.24
MFCC	Audio processing	693	SCU	718	36.1	0.53M	0.50M (14.8k)	33.1k (6.2%)	4.64k (0.9%)	6.84
			TAS	714	41.5	0.64M	0.60M (10.5k)	142.3k (22.3%)	106.7k (17.8%)	6.28
			SW	709	43.5	0.63M	0.63M (0)	142.3k (22.4%)	142.3k (22.4%)	6.05
PCA	Data analysis	2305	SCU	375	75.0	2.48M	0.88M (0.6M)	1.62M (65.2%)	20.55k (2.9%)	4.47
			TAS	388	89.6	2.66M	1.20M (0.6M)	1.76M (66.3%)	0.30M (29.6%)	4.08
			SW	381	148.3	2.73M	2.73M (0)	1.85M (67.8%)	1.85M (67.8%)	3.45

Active cycles reflect core-active cycles, i.e., cycles where the core clock is active, averaged over all eight cores. The share of active synchronization cycles is based on the total amount of active execution cycles.

(DWT); one barrier after the initial variable and pointer setup phase and after each DWT step. **Dijkstra:** Dijkstra's minimum distance algorithm for a graph with 121 nodes; for each node, the minimum distance to node zero is calculated. Two barriers per node that ensure each core is done with its part of the graph before deciding on the minimum distance for each node. **AES:** One round of encryption and one round of decryption of 1 kByte of data using the Advanced Encryption Standard (AES) in counter mode. Barriers are only used before and after the two phases of the algorithm, as it can be fully vectorized. **Livermore6:** General linear recurrence equation from the Livermore Loops [32]; the transformed, parallelizable version of the algorithm proposed in [17] was used with a 128-element single-precision input vector. A barrier must be passed on each iteration of the outer loop as there are data dependencies between the iterations. **Livermore2:** Excerpt from an incomplete Cholesky-Conjugate gradient descent that processes an 8 kByte single-precision input vector. The algorithm reduces the part of the vector that is processed in each iteration by a factor of two and, therefore, only requires 12 outer loop iterations after each of which a barrier is required. **FFT:** 512-point complex-valued single-precision radix-8 fast Fourier transform (FFT) with precomputed twiddle factors. Barriers are only required between each radix-8 butterfly step (two with the input size at hand) and at the end of the algorithm to arrange the output values in the correct order. **FANN-A:** Hand gesture recognition from [46], based on a 32-bit fixed-point fully-connected fast artificial neural network (FANN) with five layers, 691 neurons, and over 0.4 MByte of weights. Barriers are required both after processing each layer (outer loop) as well as after each fully-parallel inner loop iteration in which each core calculates a neuron value. The barrier at the inner loop is required to manage the loading of the currently required weight values into the TCDM by the DMA in the background as the TCDM is far too small to fit all weight values at once. **MFCC:** Calculation of the Mel-frequency cepstrum (MFC) (inverse FFT of the logarithm of the power spectrum) of a 20.000-element 16-bit fixed-point vector. An outer loop runs over frames of four bytes with a barrier after each iteration. For each frame, nine processing steps with a barrier in between each are carried out. With the exception of the forward FFT to compute the power spectrum, all processing steps are fully vectorized and free from synchronization points. **PCA:** 32-bit fixed-point principal component analysis (PCA) based on Householder rotations on a dataset composed of 23 channels and 256 observations; the algorithm is distributed over five processing steps (data normalization, Householder reduction to bidiagonal form, accumulation of the right-hand transformation, diagonalization, final computation of principal components) with a barrier in between each. Four of the processing steps contain numerous barriers due to data dependencies and short sequential sections for combining intermediate results from preceding parallel sections; the diagonalization part of the algorithm is largely sequential.

The applications were selected with a focus on covering both a wide range of domains and the relevant parameter space: As Table 2 shows, barrier count, the number of

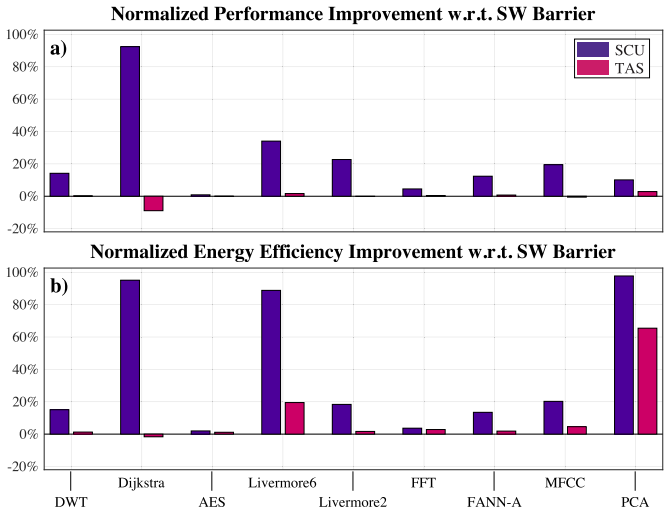

Fig. 6. Normalized performance (a) and energy improvements (b) for the range of DSP-applications and the SCU and TAS barrier implementations relative to the SW baseline.

execution cycles as well as total energy all range over four orders of magnitude. The range of average SFR sizes is roughly lower bound at around 100 cycles (Dijkstra), a size for which Figs. 5a and 5d show that synchronization overheads achieved with the SCU barrier are still well below the acceptable margin of 10 percent. At the upper end of the spectrum, applications with SFR sizes of one thousand cycles and more are as well included (AES, FFT), representing the range of SFR sizes where Figs. 5a and 5d indicate only small overhead reductions when comparing the SCU barrier to the TAS and SW baselines.

Both discussed Livermore Loops were mainly chosen for benchmarking to allow for quantitative comparison to literature. We could identify no case where systems from literature performed better when comparing against the SCU-type primitives and only two cases when comparing against the TAS- or SW-type primitives (they resulted in uniformly very similar cycle counts for both Livermore Loops). Additionally, the systems in question feature at least a doubled core count compared to our cluster: For Livermore6 with data size 256, the 16-core CMP from [17] performs 6 percent better; for Livermore2 with a 2048-element vector, the 128-core system of [16] achieves 14 percent lower cycle count. For Livermore2 and all other vector sizes used in [16], [17], we achieved performance improvements between 7 and 38 percent with the TAS and SW barrier variants and between 26 percent and 6.2× with the SCU barrier. For Livermore6, the improvements in comparison to [17] range between 17 and 55 percent with the TAS and SW barriers and between 13 percent and 4.9× when using SCU barriers.

In relation to the 7-core system used in [18], performance for Livermore2 was improved by over 5× with TAS and SW barriers and by over 8.9× with the SCU barrier. For Livermore6, we observe an improvement of 2.7× with the TAS and SW type primitives and 3× when using the SCU. For both benchmarks, [18] uses a 1024-element vector.

6.4.1 Calculation of Synchronization Overhead

As the main goal of this work is to boost energy efficiency by drastically reducing the synchronization-related overhead, we provide for each application and synchronization primitive implementation both the number of total and active cycles that cores use to execute synchronization primitives. The cycle counts have been determined with a profiling script that parses the trace files of each core and application. For SCU-type synchronization primitives, the detection is done by searching for the `elw` instruction with matching physical address. Any preceding instructions that are used to calculate the address are as well counted towards the synchronization cycles. In the case of the TAS and SW variants, two detection methods have to be used: By analyzing the disassembly of each application, the address range(s) of synchronization functions are extracted. If this step succeeds, the traces are scanned for time periods where a core executes instructions within a relevant address range. However, this method fails for many applications due to the fact that the compiler inlines synchronization functions. Consequently, the inlined functions must be detected by matching the disassembly against patterns that unambiguously identify synchronization primitives. This method requires much more careful analysis as the functions can be spread across multiple non-contiguous address ranges with linking jump or branch instructions. Furthermore, multiple entry- and exit points to and from the primitives may exist. The output of the described analysis methods is, in any case, a list of synchronization periods where each entry contains a begin and end cycle number. Combining these timestamps with the benchmarking intervals allows us to calculate both the total and active number of synchronization cycles for each core and benchmark iteration, the average of which is shown in Table 2.

It is important to note that the *total* number of synchronization cycles naturally includes core wait periods that are mostly caused by workload imbalance. Therefore, the number of *active* synchronization cycles is a much better measure for the actual synchronization overhead and substantially lower than the former count for the idle-wait featuring SCU and TAS variants and considering applications that exhibit significant workload imbalance (Livermore6, PCA).

6.4.2 Discussion of Results

Table 2 lists the most important properties of each application alongside the cycle-based and energy results. In order to provide full insight into the components contributing to energy, Fig. 7 shows both the total power and the corresponding breakdown into the shares associated with the main cluster components. Even at the nominal voltage of 0.80 V, at which we performed the experiments, the share of static power already ranges between 8 and 20 percent. Operating the cluster at lower voltages to increase energy efficiency (e.g., at 0.65 V) further increases the share, highlighting the importance of area-efficient circuit design in the context of parallel NTC systems. Finally, Fig. 6 highlights the normalized improvements in terms of cycles and energy that we were able to achieve with each application when employing the SCU- and TAS-based synchronization primitives in relation to the SW baseline.

Our solution increases the energy-efficiency of the cluster in two ways, where both essentially stem from reducing the execution time for a given task or application: First, the energy spent gets reduced roughly proportionally to execution time

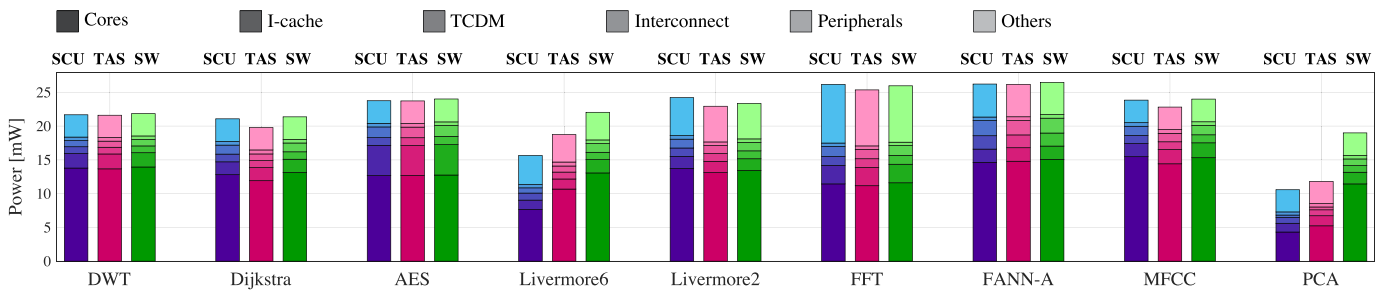

Fig. 7. Total cluster power and breakdown into the main contributors (different shadings) for the range of DSP applications. The SCU is contained in the Peripherals group.

as long as the power of the SCU-accelerated task variant is similar to that of the baseline variants. Second, the reduction in execution time allows lowering the operating point (voltage and frequency) of the cluster, moving it closer to the OEP for a given throughput or latency target.

Over the range of the benchmarked applications, the SCU achieves relative performance improvements between 1 and 92 percent, with an average of 23 percent. While the lower and upper bounds for relative energy improvement are very similar, amounting to 2 and 98 percent, respectively, power reductions with the SCU result in a higher average gain of 39 percent.

The instructions-per-cycle (IPC) shown in Table 2 is calculated as the ratio between the amount of executed instructions and active cycles, individually for each core, and then summed up over all eight cores. Since the SCU adds both a minimal amount of instructions (between one and three) and active cycles (between six and eight) per barrier, the IPCs achieved with the SCU barriers closely reflect those of the applications themselves and how well they parallelize on the employed cluster. As can be seen in Table 2, using TAS or SW barriers degrades the IPC in all cases, even though one might think test-and-set related instructions have a high IPC of close to one. The unavoidable contentions that result from simultaneous accesses to the TAS-protected locking variable, however, degrade the IPC.

When relating the average SFR size from Table 2 with the normalized improvements from Fig. 6, one can see that the SFR size is a strong indicator of whether the type of synchronization implementation influences overall performance and energy or not. Consequently, the most significant improvements are achieved with applications that exhibit SFRs of few hundreds of cycles (Dijkstra, Livermore6, PCA) and the lowest with SFR sizes of around one thousand or several thousands of cycles (AES, FFT).

6.4.3 Overhead Modeling

The relative overhead of active synchronization cycles (averaged over all cores) in relation to the SFR size for each application and barrier variant is marked in Fig. 5a. The number of *active* synchronization cycles is used since, in the synthetic benchmarks, all cores arrive at almost the same time at a barrier. It can be noticed that the results of the synthetic benchmarks in Section 6.3 can provide a reasonably accurate estimate of the synchronization overhead to be expected, given the SFR size. At the same time, in real applications, core-to-core workload imbalances cause a higher variation of the arrival instances. For the SCU barrier, the overhead estimated by the synthetic benchmarks closely matches the actual application-related one for all applications. For the TAS and SW barriers, however, the synthetic experiments mostly provide a conservative overhead estimation. This effect can be explained with the already mentioned workload imbalances; the spread-out arrival instances also reduce core-concurrent access to the TAS-variable that protects the barrier status from hazardous modification. As a consequence, fewer cycles are wasted due to contention while accessing the said variable. The fully-parallel access to the SCU barrier extension, on the other hand, causes the barrier overhead to be completely independent of the distribution of the arrival instances, leading to a greatly improved (cycle) overhead modeling.

6.4.4 Implications on Power Consumption

An important observation is that the SCU does *not* reduce power for most applications: It does not affect it significantly (see DWT, AES, FANN-A) or slightly increases power compared to the TAS variant, which also features idle-waiting (see Dijkstra, Livermore2, FFT, MFCC). As Fig. 7 shows, the increase in the latter case is due to higher power consumption in the cores. This effect is a consequence of the reduction of synchronization cycles and the therefore relatively higher share of (usually) energy-intensive processing cycles.

However, there are two exceptions to this behavior (Livermore2 and PCA), where total power is reduced by 15 and 38 percent when using the TAS barrier, or, respectively, 29 and 44 percent with the SCU variant. In these cases, application-inherent workload imbalances, indicated through the standard deviation of active execution cycles (over cores) in Table 2, result in large differences for the times that individual cores wait at barriers. Avoiding active spinning on synchronization variables during the resulting prolonged wait periods with both the TAS and SCU barriers reduces the power of the involved components (cores, interconnect, TCDM) and – since they consume the majority of overall power – also of the whole cluster very significantly. This circumstance also shows when comparing the normalized cycle and energy efficiency improvements in Fig. 6, where the gains in energy efficiency are very similar to those for performance except for the two applications in question. In those cases, the discussed power reduction results in much greater improvements for energy efficiency.

This work focuses on the optimization of the core-to-core communication and synchronization in ultra-low-power clusters of processors in the IoT domain, leveraging parallelism to improve energy-efficiency of computations rather than performance only. In different contexts, such as high-

end devices, having power/performance scalable systems, able to scale-up to 100s or 1000s of cores, is a desirable feature. However, state-of-the-art parallel computing systems, such as GP-GPUs, feature a clear trade-off between performance and efficiency, both from the point of view of the parallelism available in embedded applications as well as from a physical implementation perspective (as discussed in Section 3.1). In the context of PULP-based systems, where energy efficiency cannot be traded off against performance, scalability is still an open problem, and we plan to explore this scenario as future work.

7 Conclusion

We proposed a light-weight hardware-supported synchronization solution for embedded CMPs that aggressively reduces synchronization overhead, both in terms of execution time and energy, the latter being a crucial metric for most embedded systems. Both the proposed SCU and the hosting PULP cluster are silicon-proven and available as open-source hardware [15] under a very permissive license. In addition to showing energy cost reductions for synchronization primitives of up to 41× and resulting minimum SFR sizes of as little as tens of cycles, we demonstrated the importance of energy-efficient synchronization on a range of typical applications that covers four orders of magnitude of execution time. The proposed solution improves both performance and energy efficiency in all cases and has a beneficial impact of up to 92 percent for performance and 98 percent for energy efficiency for applications with SFR sizes of around one hundred cycles. In the future, we plan to explore hierarchical architectures composed of multiple tightly-coupled clusters, with the target of scaling up the performance of PULP systems with no compromises on energy efficiency.

Acknowledgments

This work was supported in part by the EU Horizon 2020 Projects OPRECOMP under Grant 732631, in part by Eurostars EUREKA EXCITING under Grant 10691, and in part by WiPLASH under Grant 863337.

References

[1] D. Genbrugge and L. Eeckhout, "Chip multiprocessor design space exploration through statistical simulation," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1668–1681, Dec. 2009.
[2] D. Geer, "Chip makers turn to multicore processors," *IEEE Comput.*, vol. 38, no. 5, pp. 11–13, May 2005.
[3] D. Bertozzi *et al.*, "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 2, pp. 113–129, Feb. 2005.
[4] T. M. Conte and M. Levy, "Embedded multicore processors and systems," *IEEE Micro*, vol. 29, no. 3, pp. 7–9, May/Jun. 2009.
[5] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: The Internet of Things architecture, possible applications and key challenges," in *Proc. Int. Conf. Frontiers Inf. Technol.*, 2012, pp. 257–260.
[6] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013.
[7] O. Azizi, A. Mahesri, B. Lee, S. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," *ACM SIGARCH Comput. Archit. News*, vol. 38, pp. 26–36, 2010.
[8] D. Rossi *et al.*, "Energy-efficient near-threshold parallel computing: The PULPv2 cluster," *IEEE Micro*, vol. 37, no. 5, pp. 20–31, Sep./Oct. 2017.
[9] 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. IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.
[10] S. Salamin, H. Amrouch, and J. Henkel, "Selecting the optimal energy point in near-threshold computing," *Des. Autom. Test Europe Conf. Exhibit.*, 2019, pp. 1670–1675.
[11] OpenMP Architecture Review Board, "The OpenMP API specification for parallel programming," 1997. [Online]. Available: https://www.openmp.org
[12] E. Flamand *et al.*, "GAP-8: A RISC-V SoC for AI at the edge of the IoT," in *Proc. IEEE Int. Conf. Appl.-Specific Syst. Architectures Processors*, 2018, pp. 1–4.
[13] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, "Mr. Wolf: An energy-precision scalable parallel ultra low power SoC for IoT edge processing," *IEEE J. Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, Jul. 2019.
[14] P. Schönle *et al.*, "A multi-sensor and parallel processing SoC for miniaturized medical instrumentation," *IEEE J. Solid-State Circuits*, vol. 53, no. 7, pp. 2076–2087, Jul. 2018.
[15] PULP Platform, 2013. [Online]. Available: http://www.pulp-platform.org
[16] J. Sartori and R. Kumar, "Low-overhead, high-speed multi-core barrier synchronization," in *Proc. Int. Conf. High-Perform. Embedded Architectures Compilers*, 2010, pp. 18–34.
[17] J. Sampson, R. Gonzalez, J. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2006, pp. 235–246.
[18] H. Xiao, T. Isshiki, D. Li, H. Kunieda, Y. Nakase, and S. Kimura, "Optimized communication and synchronization for embedded multiprocessors using ASIP methodology," *Inf. Media Technol.*, vol. 7, no. 4, pp. 1331–1345, Jan. 2012.
[19] B. E. Saglam and V. J. Mooney, "System-on-a-chip processor synchronization support in hardware," *Des. Autom. Test Eur. Conf. Exhibit.*, 2001, pp. 633–639.
[20] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, Jan. 1990.
[21] A. Kagi, D. Burger, and J. R. Goodman, "Efficient synchronization: Let them eat QOLB," in *Proc. Int. Symp. Comput. Archit.*, 1997, pp. 170–180.
[22] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.
[23] C. Ferri, R. I. Bahar, M. Loghi, and M. Poncino, "Energy-optimal synchronization primitives for single-chip multi-processors," in *Proc. ACM Great Lakes Symp. VLSI*, 2009, pp. 141–144.
[24] T. Tsai, L. Fan, Y. Chen, and T. Yao, "Triple speed: Energy-aware real-time task synchronization in homogeneous multi-core systems," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1297–1309, Apr. 2016.
[25] M. Loghi, M. Poncino, and L. Benini, "Synchronization-driven dynamic speed scaling for MPSoCs," in *Proc. Int. Symp. Low Power Electron. Des.*, 2006, pp. 346–349.
[26] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, "Exploiting barriers to optimize power consumption of CMPs," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2005. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1419819
[27] 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.
[28] S. H. Kim *et al.*, "*C-Lock*: Energy efficient synchronization for embedded multicore systems," *IEEE Trans. Comput.*, vol. 63, no. 8, pp. 1962–1974, Aug. 2014.
[29] 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.
[30] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2008, pp. 35–46.
[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.
[32] J. T. Feo, "An analysis of the computational and parallel complexity of the Livermore Loops," *Parallel Comput.*, vol. 7, no. 2, pp. 163–185, Jun. 1988.

[33] F. Thabet, Y. Lhuillier, C. Andriamisaina, J.-M. Philippe, and R. David, "An efficient and flexible hardware support for accelerating synchronization operations on the STHORM many-core architecture," in *Proc. Des. Autom. Test Eur. Conf. Exhibit.*, 2013, pp. 531–534.
[34] C. Ferri, A. Viescas, T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy efficient synchronization techniques for embedded architectures," in *Proc. ACM Great Lakes Symp. VLSI*, 2008, pp. 435–440.
[35] C. J. Beckmann and C. D. Polychronopoulos, "Fast barrier synchronization hardware," in *Proc. ACM Int. Conf. Supercomputing*, 1990, pp. 180–189.
[36] J. H. Kelm *et al.*, "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," in *Proc. Int. Symp. Comput. Archit.*, 2009, pp. 140–151.
[37] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Proc. Des. Autom. Test Eur. Conf. Exhibit.*, 2012, pp. 983–987.
[38] D. Palossi *et al.*, "A 64-mW DNN-based visual navigation engine for autonomous nano-drones," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8357–8371, Oct. 2019.
[39] V. Kartsch *et al.*, "BioWolf: A sub-10-mW 8-channel advanced brain–computer interface platform with a nine-core processor and BLE connectivity," *IEEE Trans. Biomed. Circuits Syst.*, vol. 13, no. 5, pp. 893–906, Oct. 2019.
[40] G. Tagliavini, G. Haugou, and L. Benini, "Optimizing memory bandwidth in OpenVX graph execution on embedded many-core accelerators," in *Proc. Conf. Des. Architectures Signal Image Process.*, 2014, pp. 1–8.
[41] A. Burrello, F. Conti, A. Garofalo, D. Rossi, and L. Benini, "Work-in-progress: DORY: Lightweight memory hierarchy management for deep NN inference on IoT endnodes," in *Proc. Int. Conf. Hardware/Softw. Codesign Syst. Synthesis*, 2019, pp. 1–2.
[42] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "Enabling OpenVX support in mW-scale parallel accelerators," in *Proc. IEEE Int. Conf. Compilers Architectures Sythesis Embedded Syst.*, 2016, pp. 1–10.
[43] P. Vogel, A. Marongiu, and L. Benini, "Lightweight virtual memory support for zero-copy sharing of pointer-rich data structures in heterogeneous embedded SoCs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1947–1959, Jul. 2017.
[44] 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.
[45] 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, Sep. 2017.
[46] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "FANN-on-MCU: An open-source toolkit for energy-efficient neural network inference at the edge of the Internet of Things," 2019. [Online]. Available: https://arxiv.org/abs/1911.03314

Florian Glaser (Student Member, IEEE) received the MSc degree in electrical engineering from ETH Zurich, Switzerland, in 2015, where he is currently working toward the PhD degree at the Integrated Systems Laboratory. His current research interests include low-power integrated circuits with a special focus on energy-efficient synchronization of multi-core clusters and mixed-signal systems-on-chip for miniaturized biomedical instrumentation.

Giuseppe Tagliavini (Member, IEEE) received the PhD degree in electronic engineering from the University of Bologna, Bologna, Italy, in 2017. He is currently an assistant professor with the Department of Computer Science and Engineering (DISI), University of Bologna. He has co-authored more than 30 papers in international conferences and journals. His research interests include parallel programming models for embedded systems, runtime optimization for multicore and many-core accelerators, and design of software stacks for emerging computing architectures.

Davide Rossi (Member, IEEE) received the PhD degree from the University of Bologna, Bologna, Italy, in 2012. He has been a post-doctoral researcher with the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi," University of Bologna, since 2015, where he is currently an assistant professor. His research interest includes energy-efficient digital architectures. In this field, he has published more than 100 papers in international peer-reviewed conferences and journals. He is recipient of Donald O. Pederson Best Paper Award 2018, 2020 IEEE TCAS Darlington Best Paper Award, and 2020 IEEE TVLSI Prize Paper Award.

Germain Haugou received the engineering degree in telecommunication from the University of Grenoble, in 2004. He was with ST Microelectronics as a research engineer, for ten years. He is currently with ETH Zurich, Switzerland, as a research assistant. His research interests include virtual platforms, run-time systems, compilers, and programming models for many-core embedded architectures.

Qiuting Huang (Fellow, IEEE) received the PhD degree in applied sciences from the Katholieke Universiteit Leuven, Leuven, Belgium, in 1987. Between 1987 and 1992, he was a lecturer with the University of East Anglia, Norwich, United Kingdom. Since January 1993, he has been with the Integrated Systems Laboratory, ETH Zurich, Switzerland, where he is professor of Electronics. In 2007, he was also appointed as a part-time Cheung Kong Seminar professor by the Chinese Ministry of Education and the Cheung Kong Foundation and has been affiliated with the South East University, Nanjing, China. His research interests include span RF, analog, mixed analog-digital as well as digital application specific integrated circuits and systems, with an emphasis on wireless communications and biomedical applications in recent years. His currently serves as vice chair of the steering committee, as well as a sub committee chair of the technical program committee of the European Solid-State Circuits Conference (ESSCIRC). He also served on the technical program and executive committees of the International Solid-State Circuits Conference (ISSCC) between 2000 and 2010.

Luca Benini (Fellow, IEEE) received the PhD degree from Stanford University and currently holds the chair of the Digital Circuits and Systems Group with the Integrated Systems Laboratory, ETH Zurich and is full professor with the Universita di Bologna. He served as a chief architect in STMicroelectronics France. His research interests include energy-efficient parallel computing systems, smart sensing micro-systems, and machine learning hardware. He has published more than 1000 peer-reviewed papers and five books. He is a fellow of the ACM and a member of the Academia Europaea. He is the recipient of the 2016 IEEE CAS Mac Van Valkenburg award and of the 2019 IEEE TCAD Donald O. Pederson Best Paper Award.