Subutai: Speeding Up Legacy Parallel Applications Through Data Synchronization

Rodrigo Cataldo, Ramon Fernandes, Kevin Martin, Jarbas Silveira, Gustavo Sanchez, Johanna Sepulveda, Cesar Marcon, Jean-Phillipe Diguet

► To cite this version:

Rodrigo Cataldo, Ramon Fernandes, Kevin Martin, Jarbas Silveira, Gustavo Sanchez, et al.. Subutai: Speeding Up Legacy Parallel Applications Through Data Synchronization. IEEE Transactions on Parallel and Distributed Systems, 2021, 32 (5), pp.1102-1116. 10.1109/TPDS.2020.3040066 . hal-03082831

HAL Id: hal-03082831

https://hal.science/hal-03082831v1

Submitted on 2 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Subutai: Speeding up Legacy Parallel Applications through Data Synchronization

Rodrigo Cataldo, Ramon Fernandes, Kevin J. M. Martin, Jarbas Silveira, *Member, IEEE*, Gustavo Sanchez, Johanna Sepúlveda, César Marcon, *Senior Member, IEEE*, Jean-Philippe Diguet, *Senior Member, IEEE*

Author version

This document is the author version of the paper "Subutai: Speeding up Legacy Parallel Applications through Data Synchronization" by *Rodrigo Cataldo, Ramon Fernandes, Kevin J. M. Martin, Jarbas Silveira, Gustavo Sanchez, Johanna Sepúlveda, César Marcon, Jean-Philippe Diguet*, accepted for publication in IEEE TPDS. The IEEE Copyright Notice is
IEEE Transactions on Parallel and Distributed Systems
Digital Object Identifier: 10.1109/TPDS.2020.3040066
The original paper is available in IEEE Xplore:
https://dx.doi.org/10.1109/TPDS.2020.3040066

***Abstract*—The decrease of the performance gain dictated by Moore's Law boosted the development of manycore architectures to replace single-core architectures. These new architectures must employ parallel applications and distribute its workload over a multitude of cores to reach the desired performance. Parallel applications are harder to develop than sequential ones since the developer must guarantee data integrity using synchronization primitives. While multiple novel solutions have been proposed to speed up parallel applications through handling one type of data synchronization primitive, exceptionally few works support multiple types of synchronization primitives and legacy code. This work proposes Subutai, a hardware/software co-design solution for accelerating multiple synchronization primitives without modifying the application source code. By providing a new user library, while retaining an existing synchronization API, legacy and novel applications can benefit from our solution. Our experimental evaluation, which provides a POSIX Threads implementation, demonstrates Subutai speeds up to 2.71× and 4.61× the execution of single- and multiple-application executions, respectively.**

***Index Terms*—Legacy Parallel Applications, PThreads, Network-on-Chip, Distributed Scheduler**

I. Introduction

Since the end of the last century, a significant shift has occurred in the industry, transitioning the processor chips from a single- to a multicore design using a dozen cores. This paradigm has evolved to incorporate hundreds and soon thousands of simple cores, performing a manycore architecture, to continue to deliver higher performance.

Unfortunately, only increasing the number of cores does not imply increasing the performance, as the applications must be parallel-compatible to exploit the hardware parallelism. Where once a single sequential thread could do the execution, now the developer has to partition the workload into multiple execution threads and synchronize their execution [1], dealing with deadlock, livelock, race condition, and non-deterministic events [2]. Decisions regarding both partitioning and synchronization of the workload are crucial to determine the achievable performance of the application on manycore systems since even small sequential portions of execution can have a significant performance impact, as observed in Amdahl's law. Because of this impact, parallelization is primarily done manually, allowing fine-grained performance optimizations.

Synchronization, namely the access and update of the application data, is a vital concern in any parallel application. The typical limitation to novel synchronization solutions is that developers have to refactor the source code. The redesign applies even to already parallel-compatible code, as the Application Programming Interface (API) of different solutions are not the same. The refactoring of source code due to API changes has substantial limitations; we highlight these three: (i) software redevelopment cost, (ii) challenge of parallel code refactoring, and (iii) lost legacy source code.

Software development cost already dominates new System-on-Chip (SoC) designs, as the manycore architecture and its counterpart, the parallel applications, are common elements of such designs [3]. Besides, the Read-Copy-Update (RCU) synchronization primitive used by the Linux kernel, for instance, influences over 16 million Lines of Code (LoC) across 15 kernel subsystems [4]; thus, even experienced developers do not easily achieve a refactoring of it.

Source code modification is always an error-prone task. McConnell estimates that up to 100 bugs can be present per thousand LoC [5]. Refactoring parallel code is even more susceptible than sequential code because often the developers are befuddled with the use of synchronization techniques. For instance, while RCU shows impressive results, it demands a thorough understanding of computer architecture design, presenting the tradeoff of rising performance gains but increasing code and maintainability complexity [2].

Finally, the essential requirement for refactoring a legacy application is the source code availability. However, often the legacy source code is lost, leaving only the binary code. Hence, the developers need to rewrite the entire code, increasing the software development cost. Moreover, given the amount of legacy software, a complete rewrite of the entire code is unlikely to happen [6].

Therefore, **we propose a novel synchronization solution that accelerates parallel applications without modifying the application source code**. Our solution speeds up even applications that do not or cannot share their codes; in this case, as

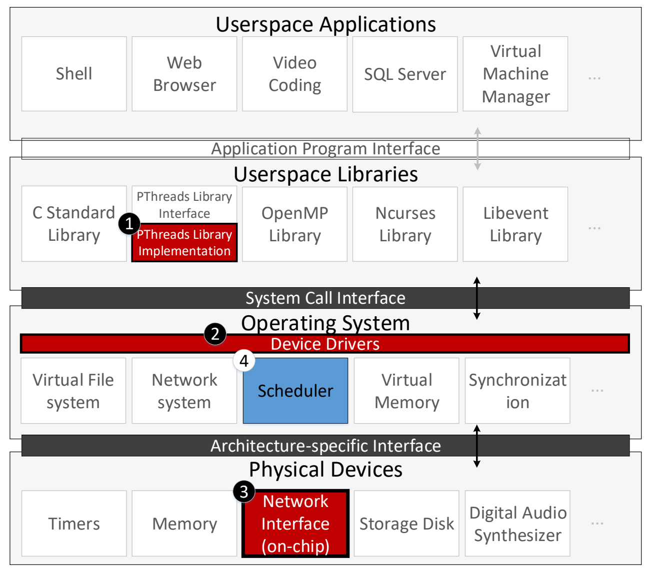


Fig. 1. Subutai components are highlighted in red (1, 2, 3) in the computing stack. Subutai only requires changes in the (1) PThreads implementation, (2) OS NI driver, and (3) on-chip NI. Additionally, (4) a new scheduling policy (in blue) is explored in this work as an optional optimization.

long as the binary is dynamically linked. Otherwise, static or dynamic linked binaries are supported. Our hardware/software solution, called Subutai, tackles the synchronization problem within a low-level Network-on-Chip (NoC) Interface (NI).

Software-wise (Subutai-SW), we implemented the POSIX Threads (PThreads) according to the IEEE Std 1003.1 standard [7][1]. Thus, any application employing the PThreads API (i.e., pthread.h) is compatible with Subutai. The PThreads compatibility restricts a multitude of optimizations since we cannot inject the source code with extra synchronization metadata or change the application communication model. In addition to interfacing with the application, our software must work with new functionalities on the hardware-side; hence, we provide an Operating System (OS) driver responsible for the latter activity.

Hardware-wise (Subutai-HW), we extended an existing on-chip NI to support, in a distributed way, the following synchronization primitives: mutex, barrier, and condition. NI handles new types of packets and requires access to a small (less or equal to 1KiB) memory to record synchronization events and metadata. Fig. 1 depicts the Subutai solution with a general-purpose computing stack, highlighting the components required for its operation.

We demonstrate that our solution speeds up single parallel applications ranging from $1.05\times$ up to $2.71\times$ for 64-thread executions. Moreover, in a competitive scheduling scenario, Subutai speeds up multiple parallel applications ranging from $1.58\times$ up to $4.61\times$. For these results, the hardware requirement for Subutai increases the area of the NI in, approximately, 46%; however, the overhead is insignificant compared to the total chip area (less than 1% for a 400mm^2 chip). The key contributions of this paper are listed next:

1) This work proposes a novel synchronization technique that avoids modifying parallel applications while accelerating their execution. The work supports both legacy and novel applications designed using the PThreads API.
2) We designed all the components of Subutai and provided a detailed analysis of its performance in accelerating standard synchronization primitives. Moreover, we evaluate it with state-of-the-art related work.
3) We conducted experiments using parallel applications provided by PARSEC, a well-known benchmark for this domain. The experiments were analyzed for both single- and multiple parallel executions. Besides, we evaluated scheduling policies for executing parallel applications. Such experiments are essential to evaluate the performance of Subutai on several execution scenarios.

This paper extends a conference version [8] by (i) evaluating Subutai with state-of-the-art related work, (ii) providing new estimations of the Subutai-HW design including memory, (iii) presenting details of the Subutai-SW implementation (userspace library and OS driver), (iv) evaluating an additional application (x264), (v) evaluating a scheduler policy proposal, (vi) evaluating concurrent application execution, and (vii) presenting the synchronization model of the analyzed applications.

II. Related Work

A program can be comprised of many computational units like threads, processes, coroutines, and interrupt handlers. We employ the term thread as a generic word to encompass these computational units. We organize the related work in software-oriented and hardware-oriented/mixed solutions. Table I summarizes the essential characteristics of these solutions and compares our work to the state-of-the-art.

A. *Software-oriented Solutions*

PThreads, Open MultiProcessing (OpenMP), and Intel Threading Blocks Building (TBB) are established solutions that use software to synchronize parallel applications. These solutions provide analogous implementations of a similar set of synchronization primitives, but with different abstraction levels. In contrast, PThreads provides a low-level interface for developers, OpenMP and TBB offer abstract programming models (fork-join and task-based models, respectively) [20].

DeLozier et al. [1] propose SOFRITAS, a software-only robust memory consistency model that can detect and prevent atomic violations on parallel applications at the cost of execution overhead (roughly 59%). Unfortunately, the applications must be annotated with a novel API when using library calls.

Boehm [10] and France-Pillois et al. [11] provide optimizations on the implementations of the PThreads and OpenMP libraries, respectively. The first work suggests relaxing the reordering rules for load and store operations, while the last work identifies an expansive function that was uselessly being called during the barrier waking process.

[1]Includes mutex, barrier, and conditions. Besides, we provide the PThreads software implementation for supporting the options provided by the `attribute` parameter.

TABLE I
RELATED WORK SUMMARY.

Solution	Orientation	Requirements	Legacy code compatible*	Uses PThreads	Target data synchronization	Experimental results
PThreads	Software	Latency	No	Yes	Barr., cond., mutex	Real applications
OpenMP	Software	Latency, app. model	No	Yes (libgomp)	Atomic, barr., mutex	Real applications
TBB	Software	Latency, app. model	No	Yes (Linux)	Atomic, cond., mutex	Real applications
RCU [9]	Software	Latency	No	May use	Mutex	Linux kernel
Boehm [10]	Software	Latency	Maybe	Yes	Mutex	Synthetic
F.-P. et al. [11]	Software	Latency	Yes	Indirectly[b]	Barrier	IS and synthetic
SOFRITAS [1]	Software	Code correctness	Limited	Yes	Barr., cond., mutex	PARSEC, ...
Sivaram et al. [12]	Mixed	Fault-tolerance	No[a]	No	Barrier	Synthetic
Abellán et al. [13]	Mixed	Latency and area	No[a]	Indirectly[b]	Barrier	Synthetic
Stoif et al. [14]	Mixed	Latency	No[a]	No	Barrier, mutex	FPGA, synthetic
MCAS [15]	Mixed	Latency and area	No	No	Atomic	Synthetic
CASPAR [16]	Hardware	Latency	Yes	No	Atomic	FFT, IS, ...
HTM [17] [18]	Mixed	Latency	Maybe	May use	Mutex, spin lock	Indirectly[c]
Not. Mem. [19]	Hardware	Latency, app. model	Yes	May use	Spin lock	MPEG-4 decoder
Subutai	**Mixed**	**Latency and area**	**Yes**	**Yes**	**Barr., cond., mutex**	**PARSEC**

Barr. = Barrier; cond. = Condition; app. = Application; Not. = Notifying; Mem. = Memories; F.-P. = France-Pillois;
* This term is defined in Section II-C;
[a] Not addressed in the work;
[b] The work employs OpenMP, and it employs PThreads internally;
[c] HTM can be used on the PThreads implementation.

Attiya et al. [21] formally proved that deterministic structures, as employed by the previously discussed libraries, cannot eliminate the use of expensive synchronization. Therefore, non-deterministic solutions focusing on relaxing the constraints that force the use of such expansive synchronization have been proposed to tackle this problem. Kirsch et al. [22] propose k-FIFO, which is a lock-free queue that removes up to $k - 1$ out-of-order elements from the queue. Desnoyers et al. [9] describe a synchronization technique based on the publish-subscribe mechanism called RCU. Parallel applications that rely on RCU have to deal with stale data. The bottleneck of these solutions is that the application code adaptation is passed on to the developer.

B. Hardware-oriented/Mixed Solutions

Sivaram et al. [12] propose a fault-tolerant hardware-based barrier synchronization. Their design uses a tree structure to sum intermediate values, decreasing the number of packets injected into the network. Their work is complementary to our solution. Abellán et al. [13] explore three HW barrier architectures and integrate them on the OpenMP programming model. Unfortunately, they evaluated only synthetic applications. Stoif et al. [14] implement an arbiter on FPGA that guarantees mutual exclusion to a portion of the shared memory area and an HW-based synchronization barrier that speeds up the application execution; however, their work does not implement full barriers and conditions, and it is limited to simple test cases instead of real applications.

CASPAR [16] improves the performance of CAS operations by breaking the serialization of multiple CAS calls and executing them in parallel. Patel et al. [15] propose a special HW instruction, called MCAS, to change multiple memory positions atomically, optimizing the synchronization process. Hardware Transactional Memory (HTM)[2] provides an abstraction for executing blocks of code atomically. HTM guarantees correctness by aborting transactions that conflict with others [17].

Finally, Martin et al. [19] propose the Notifying Memories concept to reduce communication latencies introduced in the NoC by pruning useless memory accesses. This concept uses spinlocks and is applied to dataflow applications only. Our work is also based on extending the NI architecture, but targeting shared-memory systems.

C. Comparison with the state-of-the-art work

A direct comparison of works in the data synchronization field is unfeasible as they do not employ a common test scenario that standardizes the experimental evaluation. Especially hardware and mixed solutions employ a varied set of target applications. Table I shows that there is no intersection of applications in the experimental results. Consequently, we limit the comparison of experimental results to published results on Section VIII; here, we discuss the support for data synchronization primitives and legacy code.

Three solutions are generic API specifications (PThreads, OpenMP, TBB) for cross-platform use. All other works are optimizations on existing APIs, except for RCU, as it creates a read-write lock capable of reading and writing at the same time. Boehm optimizes memory barriers for lock and unlock PThreads procedures. The approaches proposed by France-Pillois et al. and Abellán et al. share the same idea of optimizing the use of barriers in OpenMP applications. The former achieves this through a software-only approach, while the latter uses a mixed solution. MCAS and CASPAR optimize the use of CAS procedures on lock-free applications. The Notifying Memories solution targets a specific programming

[2]HTM can be simulated in software, yet the overhead imposed by the software layer can be prohibitive [23].

model and synchronization scenario: data-flow and spinlocks, respectively. HTM allows speculative execution of critical sections guarded by mutexes or spinlocks. Finally, our solution accelerates PThreads data synchronization primitives through hardware execution while keeping legacy-code compatibility.

We define that a solution is legacy code compatible if a given application can use the set (or a subset) of the solution, either by (i) recompiling without source code changes, or (ii) dynamically linking to a library provided by the solution. Therefore, besides Subutai, the following solutions support legacy code: Boehm [10], France-Pillois et al. [11], CASPAR [16], Notifying Memories [19], and HTM [17].

The works of Boehm and France-Pillois et al. are entirely done at the software level; they are not directly related to our work, as the former does not support reordering I/O operations (which we use for Subutai-HW communication), and the latter is an optimization for OpenMP (which we only support indirectly). CASPAR accelerates a different type of application (lock-free applications) not supported directly by PThreads or Subutai. Notifying Memories can benefit from our work if the spinlocks usage is done through PThreads (i.e., `pthread_spin_lock`), which is not the case of the paper presented in [19]. Besides, Notifying Memories target the data-flow application model only, while we support any model that uses the shared-memory paradigm.

HTM has two operation modes, whereas the Hardware Lock Elision (HLE) is the only mode with legacy support. HLE extends the parallel library code (e.g., PThreads) to use a hot/slow path approach. Firstly, the operation is executed speculatively using HTM; if it fails, then the legacy code is executed. HTM uses the same approach of Subutai, making changes in the library synchronization routines only. Besides, HTM is complementary to our solution, as both can be used in unison to handle synchronization primitives.

To the best of the authors' knowledge, Subutai is the only solution that speeds up various types of synchronization primitives while keeping unchanged the userspace interface (i.e., API).

III. Software-only and Subutai Solutions

Solutions for data synchronization are implemented in software-only (SW-only) or in a hardware/software composition. The solutions provide trade-offs according to the constraints on the target design (e.g., portability, performance). This section aims to clarify the target architecture used for achieving the experimental results, as well as to clarify the control flows used to synchronize shared data, using an example based on the Linux OS.

A. Target Architecture

Fig. 2 shows a schematic representation of the target architecture. Each core communicates with caches and a local NI. An NoC with routers using a standard design that includes buffers, a crossbar switch, and a switch allocator implements the interprocessor communication.

Modern multiprocessors consist of double digits of processing core units [24]. Thus, we target an NoC-based manycore architecture composed of 64 processing cores. Each core has access to instruction and data caches. The Level 1 cache is private and is divided into instruction and data caches. The Level 2 cache is shared among the cores, and banks are distributed on the system. Therefore, our target architecture uses a Non-Uniform Cache Memory Access (NUCA) architecture with faster L2 accesses for nearby banks. We explore synchronization solutions for Symmetric Multiprocessing (SMP), because it facilitates the development of parallel applications, as developers do not need to concern themselves with data placement [25]. Hence, cache coherence is required and used.

The SW-only uses a single instance of Linux, while Subutai employs a decentralized approach, where each core has its self-governing OS. The decentralized OS design enables the scheduler to be decentralized as well. A decentralized scheduler provides a faster thread switching, which benefits parallel applications. Additionally, for dozens or more cores, message passing can be much faster than memory sharing [26].

B. Target Parallel Library

Subutai transforms software events (e.g., mutex lock, condition wait) in hardware events (e.g., NoC packets). As such, we can target any number of available library interfaces. We chose the PThreads interface because (i) it is widely employed as a *de facto* standard to parallel application implementation, and (ii) it is used internally as the base of multiple synchronization solutions, as shown in Table I. Consequently, PThreads provides Subutai a broad range of applicability.

We focused on three of the four main groups of the PThreads standard operations: mutex, barrier, and condition handling. Thread events (create, exit, join) are not on the critical path, and so are left to be handled at the OS level. An extensive description of PThreads operations is out of our scope. We limit the discussion to the essentials of the three focused groups.

The mutex group contains *locking* and *unlocking* functions. Locking is a blocking function that exclusively locks a vari-

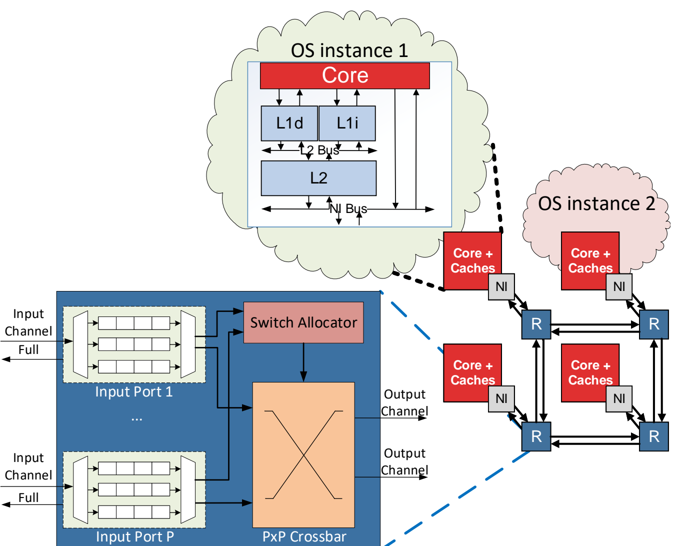


Fig. 2. Schematic of the manycore target architecture.

able. If the variable is already locked, the calling thread is blocked. Otherwise, this operation returns the variable locked by the calling thread. Unlocking is a non-blocking function that changes the variable state and wakes up blocked functions if there are any waiting threads.

The barrier group contains a single blocking function, called *wait*, which synchronizes participating threads at a user-specified code point. A barrier has a fixed number of threads decided at allocation time; participating threads are only woken up when they all hit the barrier.

The condition group contains *wait*, *signal* and *broadcast* functions. Wait is an unconditionally blocking function that inserts threads on a waiting list for a condition event. The operation of the wait function requires locking a mutex variable, which is passed as a reference to the function; this mutex is unlocked once the wait function concludes its work. The signal and broadcast are non-blocking functions that wake up one and all threads, respectively, waiting for a condition event. In these cases, the mutex is optional.

For all groups, one or more queues are required to record blocked threads. Condition functions need to handle two queues due to the associated mutex. Barrier and mutex functions deal with only one queue. Besides, the three groups have non-blocking functions that allocate and deallocate variables. This work replaces the handling of these operations from an entire software solution to a hardware/software approach.

C. Software-only Solution

Fig. 3 exemplifies the synchronization flow for the SW-only solution deployed on Linux. The example starts with the user application requesting a synchronization operation through a function call, such as a mutex lock. The function is associated with a PThreads interface that acquires the requested lock for the thread using a memory shared among application threads.

The SW-only implementation tries to acquire the mutex atomically multiple times. The first moment occurs within the PThread library, which, on success, immediately returns to the application (delay marked with t_1). Otherwise, the PThread library calls the Linux Kernel (specifically the Futex subsystem), which has another codepoint for obtaining the mutex; the last codepoint is the most time-critical point, as it implies that the current thread goes to the sleeping state, waiting for a mutex unlock event originated by the owner thread to be awakened for requesting the mutex again. t_2 and t_3 reference the delays associated with these two accesses to the shared memory, respectively. Additionally, if the thread cannot acquire the mutex lock after being awakened (due to another thread executing on t_1), then the SW-only solution implements a loop to re-execute its code. At each loop, the basic delays referenced by t_2 and t_3 may be increased by Δt_a, or $\Delta t_a + \Delta t_b$, depending on where the mutex lock is obtained [27].

Throughout the timing of the synchronization flow, extra delays may occur due to access to the shared cache memory, as well as possible cache misses in the shared data that would imply a much more significant delay. Also, since Futex is potentially distributed into the manycore caches, these two last codepoints imply communication costs through the NoC.

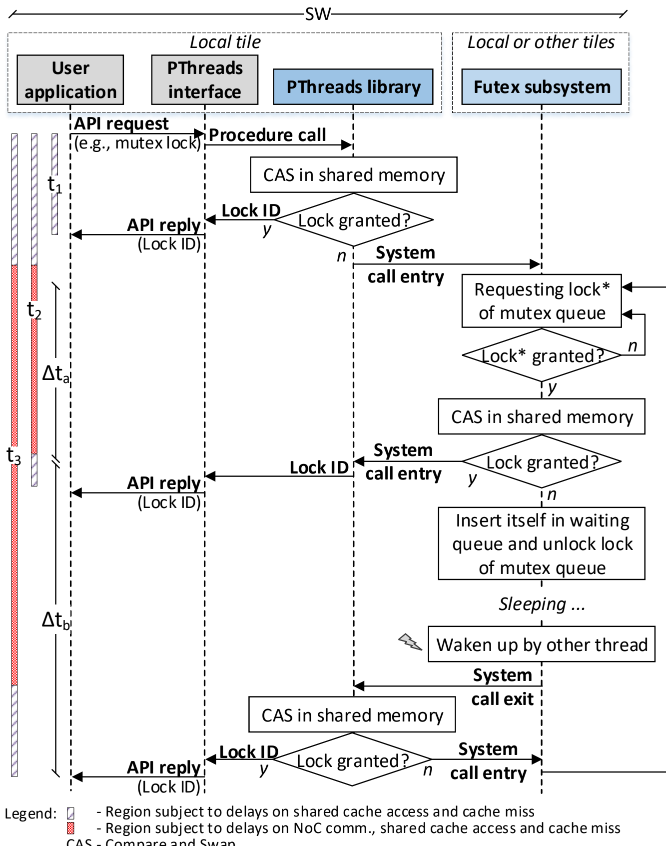


Fig. 3. Synchronization control flow employed on Linux-only based solution.

D. Subutai Solution

Subutai is a synchronization solution for legacy and novel parallel applications comprised of a software/hardware co-design to perform fast synchronization operations. This section describes the high-level interaction of the Subutai's components, which are illustrated in Fig. 1, together with a general-purpose computing stack.

Subutai encompasses a userspace library, a kernelspace driver, a hardware module, and an optional scheduler policy (discussed in Section VII). The userspace library mimics an existing synchronization solution intended for parallel applications. Therefore, the Subutai library procedures provide the same interfaces (i.e., API) with different implementations. The ability to mimic existing synchronization libraries is an essential feature of Subutai to speed up parallel applications.

Each core in the system has a Subutai-HW module that extends the NI and is responsible for accelerating synchronization operations. Subutai-HW is a Finite State Machine (FSM) coupled with a small dedicated memory (details in Section V). Once the user application calls a procedure, the Subutai library employs kernel services through system calls, providing the link between the hardware and software parts. Thus, the userspace library abstracts the hardware protocol (Subutai-SW - details in Section IV).

Fig. 4 depicts the Subutai communicating flow. A unique identifier (ID) on the entire system addresses each synchronization variable. An incremental counter determines the NI

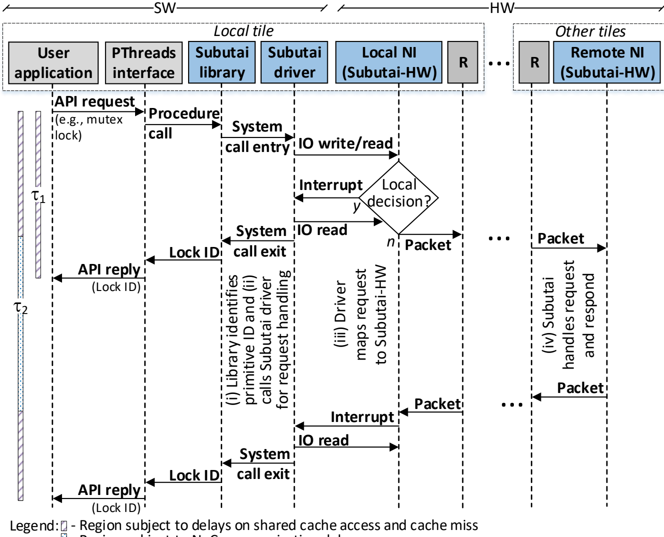


Fig. 4. Synchronization control flow employed on Subutai.

that hosts the synchronization primitive: NI_0 hosts the first primitive; NI_1 hosts the second one, and so on, following a fairness method. Other dynamic allocation strategies can be further studied, but this is out of the scope of this work.

The communication flow of Subutai starts with the application making a PThreads interface request through any function described in this section; the Subutai library identifies the unique ID for this primitive and passes it to the driver along with the interface request. Then, the driver writes to either registers or a memory that the NI has access to; this decision is made at the driver level with the capabilities available in the system. Next, the driver writes in a control register to inform the command to the NI and waits for an interrupt to receive the remote response.

In case the local Subutai-HW hosts the lock, the NI can respond immediately, performing a prompt request from the driver. Thus, the driver does not use the router, avoiding the injection of packets in the NoC; the delay of this procedure is marked with τ_1. Therefore, situations where the local Subutai-HW hosts the synchronization primitive implies a quick response as the request does not propagate across the NoC.

If the local Subutai does not host the lock, then the local NI injects a packet into the NoC targeting the remote Subutai-HW, which handles the request and responds to the local NI with a new packet. The address of the remote Subutai-HW is embedded into the ID packet field (discussed in Section V-A). This procedure implies an additional delay of packet traffic on the network, being noted by τ_2.

E. Subutai vs SW-only Solutions

The comparison of Fig. 4 with Fig. 3 allows us to understand the differences between Subutai and SW-only approaches. Synchronous flows marked by t_1 and τ_1 exemplify situations where the local processing manages the lock. Thus, regardless of the approach used, the response latency is lower compared to the latencies of the decentralized processing flows.

Subutai offers a more efficient hardware-level solution for a decentralized decision; thus, the flows marked with t_2 and t_3 have higher latencies than the one marked with τ_2.

The reasons for the lower latency of Subutai are: (i) locking for the mutex queue (marked with * in Fig. 3) is required only in the SW-only approach, as Subutai-HW has access to private memory area to handle the concurrent threads (Section V-A); (ii) susceptibility to data conflicts in distributed shared caches, which does not occur in Subutai that implements this functionality using dedicated queues in Subutai-HW (Section V-A); (iii) the efficiency of a lock event when another thread is using it. The Subutai implementation returns this information to the local tile as soon as it is available, while the SW-only implementation is delayed by the OS scaling (Fig. 4 – HW events can occur concurrently to the execution of the thread); (iv) the use of dedicated control packets allows to employ Quality-of-Service (QoS) techniques, providing differentiated priority for the traffic of control packets (Section V-A). The consequence is that control packets are propagated with lower average latency and that the variability between latencies is also lower compared to data packet latencies.

As a conclusion, either because of cache conflicts or packet latency variability in the NoC, Subutai ensures more predictability than the SW-only solution.

IV. SUBUTAI-SOFTWARE (SUBUTAI-SW)

Subutai provides a new PThreads library for parallel applications to use our solution. Every time the user application requests an operation on a mutex, barrier, or condition, the library passes on the request through a system call for the OS driver (items (i) and (ii) from Fig. 4). The request is changed in terms of structure, as the user application handles these synchronization variables by variable names, which are memory positions, while the hardware tracks these variables with a unique ID unrelated to the memory and name of the variable. The driver receives the unique ID for the variable, which is known by the library (not known by the application) and decodes the packet destination through reserved fields in this ID (Section V-A) (item (iii) from Fig. 4). The request is processed in hardware, and, eventually, the response is received in the local NI. Then, the local NI interrupts the software to notify it of a packet arrival event; the OS driver reads it and is able to finish the user application request (last four steps of Fig. 4). Finally, other types of PThreads operations (i.e., thread management) are kept unchanged.

Because most operations of PThreads are offloaded to be handled on the hardware, valuable cache space can be saved (up to 91.7% compared to x86_64) for the respective structures of the synchronization variables, as shown in Table II. All synchronization primitives have the same size in Subutai since they only contain the 4-byte unique ID (refer to Section V-A). The on-chip NI driver implementation was based on an existing driver that performs basic procedures for sending and receiving packets. We reuse these procedures for the requests from the PThreads library. Additional logic is introduced in the driver to understand the packets sent and received, as Subutai makes the driver an active component to change thread states on its own (e.g., wake up a thread when it owns a mutex).

TABLE II
MEMORY SPACE REDUCTION OF SYNCHRONIZATION PRIMITIVES.

Primitive (name)	GNU LibC x86_64 (bytes)	Subutai (bytes)	Reduction (Percentage)
mutex	40	4	90.0%
barrier	32	4	87.5%
condition	48	4	91.7%

V. SUBUTAI-HARDWARE (SUBUTAI-HW)

A. Architecture and Implementation Choices

Subutai-HW extends a standard NI architecture for handling synchronization operations fastly. Fig. 5 shows the schematic representation of Subutai-HW and its location on the target architecture. The main components of Subutai-HW are (i) an FSM, (ii) a set of registers; and (iii) a local ScratchPad Memory (SPM), which is entirely controlled in HW by the FSM, except for memory initialization. Initialization is done through the OS driver and requires the creation of a free double-linked queue. We validated and implemented the Subutai-HW architecture by Register-Transfer Level (RTL) simulation [28] and synthesis [29]. Besides, we developed an analytical model to demonstrate its operation latencies and scalability.

The left-hand side of Fig. 5 shows that Subutai-HW employs double-linked queues to record events. As an alternative to statically allocating for the worst case, the double-linked queues allow Subutai-HW to employ a dynamic allocator for reducing memory consumption to the minimum, at the cost of additional pointer arithmetic logic. Besides, condition variables are dealt more efficiently with such structure, as it avoids the thundering herd problem [30]. We based the queue manipulation on the futex implementation of the Linux kernel [31].

Subutai-HW operates using two structures for recording information. Fig. 6 shows the first one, which records the metadata of the synchronization primitives. Software only knows the first 32-bit field, which is employed as an ID of this primitive. However, for Subutai-HW, the first bit "F" is used to allocate/deallocate this structure. The next 7-bit field is the unique ID for the NI on the system. Lastly, the furthest 24-bit field is used as a pointer to itself; we employ this technique to avoid the cost of searching for an element of the structure every time a new request has arrived. The second 32-bit field encompasses the head and tail of the double-linked queue. The last 32-bit field records values used for some of the primitives. The first 16-bit field is employed to (i) record the thread and core that owns a mutex, and (ii) store the current number of threads waiting on a barrier. The barrier primitive uses the furthest 16-bit field to record the maximum number of threads allowed in a barrier.

Fig. 7 shows the second structure – a double-linked queue element composed of six fields. The first bit is employed to allocate/deallocate the element. The "prev" and "next" fields are pointers to the previous and next elements, respectively, or nil if they do not exist. The 16th bit "R" is reserved and used for memory alignment. The last 32-bit field identifies the requesting thread. The "Core ID" field is padded with zeroes because the NoC packet uses only 8-bit to identify the core.

TABLE III
LATENCIES FOR SUBUTAI-HW FSM STATES. c = CYCLE LATENCY, m = MEMORY LATENCY, n = NUMBER OF SYNCHRONIZATION VARIABLES HANDLED BY SUBUTAI-HW, ρ = NUMBER OF THREADS ON A BARRIER.

State	**Best response time**	**Worst response time**	**Packet Injection**
Allocation	$4m + c$	$(n \times m) + (3m + c)$	$(n \times m) + (m + c)$
Deallocation	$3m$	$3m$	None
Mutex Lock	$2m + c$	$11m$	$2m + c$
Mutex Unlock	$2m$	$10m + c$	$2m + c$
Barrier Wait	$7m$	$(m + c) + \rho \times (11m + 3c)$	$(m + c) + (12m + 4c) + (23m + 7c)... = (m + c) + \rho \times (11m + 3c)$
Condition Wait	$5m + c +$ Mutex Unlock	$10m + c +$ Mutex Unlock	None
Condition Broadcast	m	$18m + c$	$11m + c$
Condition Signal	m	$29m + 2c$	$11m + c$

The minimum memory requirement for the SPM is one control element and 63 queue positions, regarding a target 64 core architecture. Since we have to record up to $p - 1$ cores, the minimum SPM size is $\frac{1\times96+63\times64}{8} = 516$ bytes. Note that Subutai-HW is incorporated into every NI; consequently, we handle up to 64 primitive variables even with minimum sizing. The target architecture employs an SPM of 1 KiB (4 control elements and 122 queue elements) that handles up to 256 primitive variables in hardware. A double-linked queue allocates elements dynamically, allowing Subutai to consume memory on demand. A static allocator, on the other hand, cannot handle more than one control element with only 122 positions available ($< 2\times63$) – since the worst-case scenario is 63 positions per element[3], as explained earlier. Thus, a static solution would be either too limited or a waste of memory resources.

Although the number of primitives used in the experimental results is far from the SPM memory limit, there are two scenarios where the SPM cannot handle a request. In one scenario, the system does not have more primitive space available in any SPM; thus, Subutai rolls back to provide the SW-only implementation of the primitive. In the other scenario, there are no more queue elements available in a given primitive; therefore, we respect the POSIX standard and set `errno` to `EAGAIN` [7], hinting to the developer that it should try again later.

B. Response Time

Table III shows the latencies of the states as dependent on the Subutai-HW cycle c, the SPM write/read latency m, the number of synchronization primitives handled n, and the maximum number of threads on a barrier ρ. Each memory operation can either be a write or read operation in a given m cycle. The first column identifies the Subutai-HW state. The second and third columns identify the fastest and slowest

[3]We assume for the sake of size estimation that the number of threads does not exceed the number of cores. However, the queue is capable of handling such a scenario.

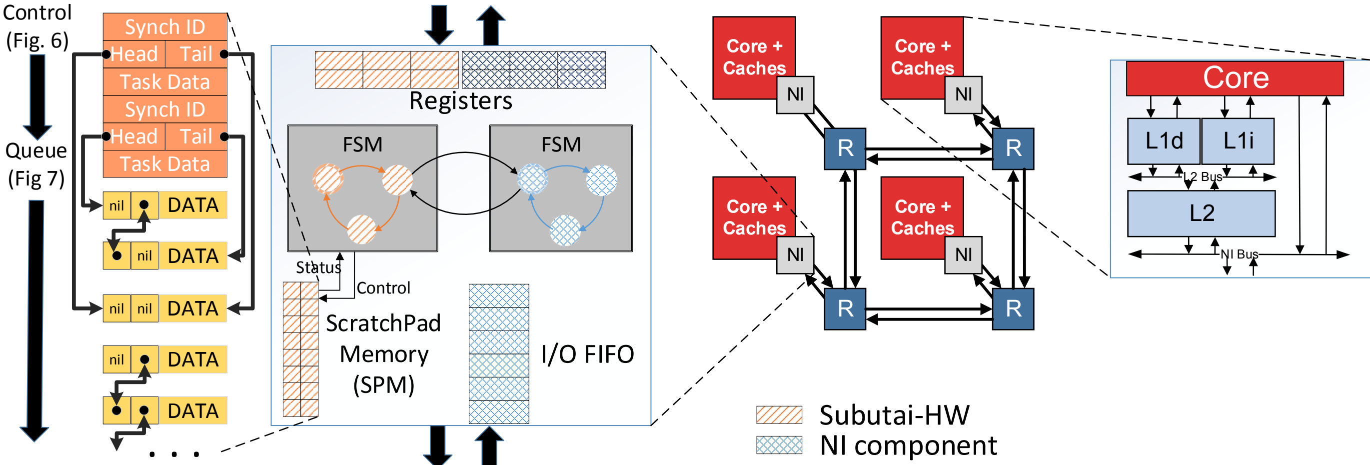


Fig. 5. Subutai-HW organization - specific Subutai-HW elements are highlighted in orange, whereas standard NI components are in blue.

latencies for the state, respectively. Finally, the last column shows when the packet is ready to be injected into the NoC – as, for some states, packets can be injected before finalizing the request processing. Additionally, some states (e.g., Deallocation) do not need to generate packets at all.

To illustrate the best and worst response time of Table III, we describe the Mutex Lock state, which models the `pthread_mutex_lock` operation. The fastest scenario, whose latency is $2m + c$, happens when the mutex is unlocked. It requires two memory operations: (i) fetch the control structure (field "Value" from Fig. 6) to check the owner of the mutex (latency = m); and (ii) rewrite this field with the requesting thread (latency = m). Finally, the NI is notified that a new packet can be injected (latency = c). The injected packet is the same as the requesting packet except for the header. The worst scenario takes more time (latency = $11m$) because the state deals with two queues entries. It starts with the same memory operation that reads the control structure for this primitive. Thus, the circuit realizes there is already an owner, which demands to queue up the request. First, Subutai-HW allocates a free queue entry and updates its queue pointers (takes up to 4 memory operations); then, it writes the requesting thread information into it and the tail information in the primitive metadata (6 more memory operations), performing 11 memory operations in total. The latency for the other states follows a similar procedure.

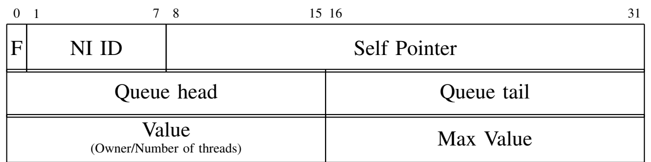


Fig. 6. Subutai-HW control structure.

0	1 ... 15	16	17 ... 31
F	Prev	R	Next
Thread ID		Core ID	

Fig. 7. Subutai-HW queue structure.

TABLE IV
SUBUTAI-HW STATES LATENCY WITH $c = 1ns$, $m = 2ns$, $n = 4$, $\rho = 63$, $FSMentry = 4ns$, $FSMexit = 1ns$.

State	Best response time (empty queue)	Worst response time (queued)	Packet Injection Best	Packet Injection Worst
Allocation	14 ns	20 ns	10 ns	15 ns
Deallocation	11 ns	11 ns	None	
Mutex Lock	10 ns	27 ns	None	10 ns
Mutex Unlock	9 ns	26 ns	None	12 ns
Barrier Wait	19 ns	1583 ns	None	7, 32, 57, ... ns
Condition Wait	20 ns	47 ns	None	
Condition Broadcast	7 ns	42 ns	None	27 ns
Condition Signal	7 ns	65 ns	None	27 ns

Table IV shows the latencies used in the experimental results. We clocked Subutai-HW at the same frequency as the NI (1 GHz). SPM employs the previously discussed 1 KiB single-port SRAM-based implementation with uniform access of 2 cycles, 4 control structures, and 122 queue entries. Besides the Subutai-HW state latencies of Table III, Table IV includes the values of the NI used in this work; let $FSMentry$ and $FSMexit$ be the entry and exit latencies for Subutai-HW, then $FSMentry = 4$ ns (3 cycles for 3 flits of 32 bits and 1 cycle to decide the next state) and $FSMexit = 1$ ns (1 cycle to set a flag) to reach any state. A detailed report of equations and values described in Tables III and IV, and the pseudo-code implementation of Subutai-HW can be found in [32].

The latency required to release threads on a barrier exceeds one thousand nanoseconds due to the queue size of threads waiting on the barrier – it does not represent the packet injection latency. Thus, some threads execute much earlier than the total value. As shown in the last column, the packets are injected periodically at every 25 ns, except for the first packet, which is injected in 7 ns. Thus, the total number of cycles is 1583 ns, which is composed of the following parameters: $FSMentry + FSMexit + m + c + \rho \times (11m + 3c)$.

The *Condition Broadcast* and *Condition Signal* states show interesting latency results. At first glance, it would seem more plausible that releasing one thread (signal) would be faster than

releasing all threads (broadcast). However, this conjecture is not valid due to the following reasons. First, by releasing all threads, the state has to deal with only one queue (mutex) instead of two queues (mutex and condition). Second, due to the way condition works, only a single thread is indeed released since a mutex is associated with it. Consequently, the broadcast state avoids the scenario previously described for the barrier state – only the owner of the mutex will be released.

Subutai-HW also includes six 32-bit and three 1-bit registers; three are used for the packet fields (Fig. 8), and six more to (i) handle the free queue; (ii) memory swapping operations; and (iii) control flags to receive and send packets. For receiving/sending packets, Subutai-HW reuses the already available registers of the NI. The packet structure is combined with the recorded information in the two control structures (Figs. 6 and 7) to handle any request.

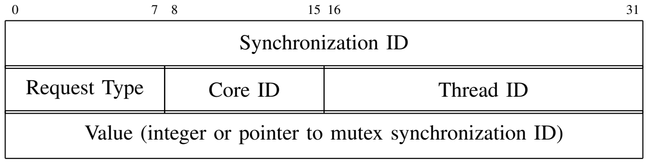


Fig. 8. Subutai's packet format.

VI. Application Synchronization Model

The performance of Subutai is evaluated through the widely used PARSEC benchmark, as it provides a wide range of application domains, parallelization models and data sharing. From their application set, we employ Bodytrack, Streamcluster, and x264; we limit our discussion to the synchronization model used by these applications. An extensive overview of these applications is outside the scope of this paper (more information can be obtained in [33]).

Bodytrack is a computer-vision application that tracks a 3D pose of a mark-less body. It uses mutexes for data sharing, and conditions and barriers to make sure all threads are synchronized and able to handle more requests. The workflow of Bodytrack starts with a single 'master' thread (T_0) responsible for creating synchronization primitives, creating T_{n-1} threads, and sending computation requests for them. Then, the threads $T_1, \ldots, T_{n-2}$ do the actual computation through the requests from T_0. Finally, the last thread (T_{n-1}) performs asynchronous I/O operations (e.g., loading images from disk to memory).

Fig. 9 depicts the workflow of Bodytrack. Initialization is done exclusively by T_0, where the synchronization variables and threads are created. Then, T_0 divides the computational work among the worker threads and sends a condition broadcast for all worker threads. Meanwhile, each worker thread checks if its work is available - if so, the thread skips the condition and moves on to the next phase; otherwise, the thread waits for the condition variable.

Each thread uses mutexes to access shared memory while performing the computational work. Meanwhile, T_0 uses a barrier to wait for all worker threads to conclude their jobs. The barrier guarantees that all worker threads are ready to handle the next work request. As the worker threads finish

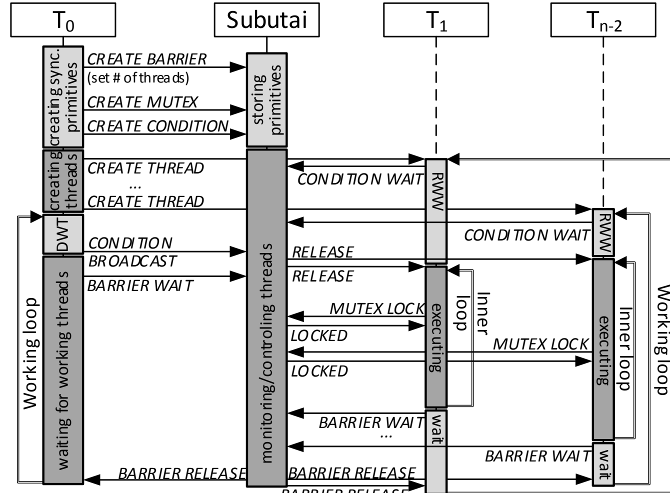


Fig. 9. Bodytrack's synchronization scheme. T_{n-1} is not shown as it does not participate in the Bodytrack workflow.

their work, they join the barrier as well. Only when all threads have joined the barrier, they are released to execute the next phase, which loops back to the generation of more work to the worker threads. This loop is executed until no more work is available.

The workflow plotted in Fig. 9 simplifies three aspects of the work of the Bodytrack application. Firstly, after the worker threads have received a request through the condition, they acknowledge it using another barrier (not shown in Fig. 9), and the associated mutex of the condition. Secondly, Fig. 9 does not show the thread responsible for asynchronous I/O (T_{n-1}) because it communicates only with T_0 and the number of requests is at most 10 events, which is tiny compared to the core workflow. Thirdly, Fig. 9 abstracts the application process conclusion because this process does not use data synchronization.

Streamcluster is a data-mining application that solves the online clustering problem for a stream of input points; it computes an approximation for the optimal clustering of them. This application has a simpler communication model than Bodytrack, using a single instance of mutex, barrier, and condition. Nonetheless, Streamcluster shares the same barrier-based synchronization scheme as Bodytrack.

x264 is a lossy video encoder for high-quality streams that do not employ barrier synchronization primitives, and all mutexes variables are associated with condition variables. This application uses a sliding pipeline model, whose number of pipeline stages equals the number of video frames, while the sliding window is determined at runtime by the number of thread requested. The total number of stages created is $1 + 2 \times videoframes$ [34].

Table V displays the number of synchronization primitives used by each PARSEC application. Additionally, Table VI depicts the number of synchronization events for the same set of applications. On the one hand, neither the number of threads nor the input size affects the number of synchronization

TABLE V
NUMBER OF SYNCHRONIZATION PRIMITIVES FOR PARSEC (SIMMEDIUM INPUT).

Application	Mutex	Condition	Barrier
Bodytrack	3	1	4
Streamcluster	1	1	1
x264	95	95	0

TABLE VI
NUMBER OF EVENTS OF SYNCHRONIZATION PRIMITIVES DURING THE EXECUTION OF PARSEC APPLICATIONS (SIMMEDIUM INPUT).

Application	Type	Events per number of threads		
		16	32	64
Bodytrack	Barrier[1]	2101	4293	13416
	Condition	447	750	1529
	Mutex	9000	10472	8677
Streamcluster	Barrier[1]	208048	364480	728960
	Condition	381	802	1274
	Mutex	510	1054	2142
x264	Barrier	0	0	0
	Condition	86	310	354
	Mutex	4154	4340	4344

(1) Every packet is counted as an event. Thus, in a 64-thread barrier, for instance, 64 events are generated for waiting on a barrier, and other 64 events are generated for releasing them, as the NoC does not support broadcast messages.

variables, except for x264, where the number of frames (i.e., input size) affects the number of primitives. On the other hand, Table VI displays that both affect the number of calls of these primitives.

VII. SCHEDULING

We employ multiple parallel applications that share computing resources to minimize the global idle time and maximize the rate of application instances. This work proposes a new scheduling policy to accelerate the critical sections of parallel codes. Additionally, we evaluate its performance impact on parallel applications against the Round-Robin (RR) scheduler, which does not differentiate the sections of parallel applications. We do not propose a new scheduler, instead, a scheduler policy that any scheduler can adopt in its decisions.

We target the scheduler rather than the application, as it does not require modifying the application code. Thus, we intend to further speed up applications by aggregating multiple parallel applications with a critical section-aware policy. Running multiple applications makes every application slower (i.e., increases the execution time), as they have to contend for computational resources. However, the scheduling impact on execution time can be mitigated by the policies employed on the scheduler.

The fair scheduler employs equal-priority for all applications making them have the same slowdown to enforce fairness. Eq. 1 shows the lower-is-better unfairness metric [35] that can be used to evaluate the fairness of the scheduler.

$$Unfairness = \frac{MAX(Slowdown_1, \ldots Slowdown_n)}{MIN(Slowdown_1, \ldots, Slowdown_n)} \quad (1)$$

Where n is the number of applications in the workload and $Slowdown_i = \frac{ET_{schedi}}{ET_{alonei}}$, where ET_{schedi} denotes the execution time of application i under a given scheduler, and ET_{alonei} is the execution time of the application i when executing alone.

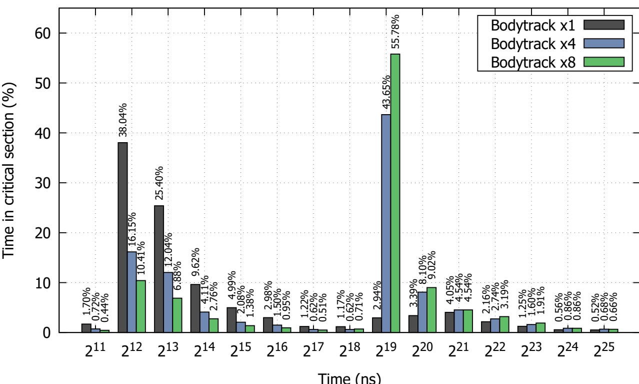


Fig. 10. Comparison of three application sets for the distribution of time spent in critical section on a RR scheduler with a timeslot of 1ms.

The baseline scheduler employs the RR policy that avoids starvation by running the application set in a deterministic order and uses a fixed share of execution time (timeslot). These features provide a fair distribution of execution runtime as all applications have the same number of timeslots regardless of the application type (i.e., low unfairness)[4]. The experimental unfairness values obtained will be discussed in the experimental results section.

Parallel applications can be roughly divided into sequential and parallel execution modes. Every parallel application contains at least a small sequential part for initialization, such as thread creation and parsing of application parameters. Mutual exclusion data access is another sequential execution commonly used among parallel portions. By using a mutex, either independently or associated with a condition, a thread is exclusively executing a given portion of code (i.e., a critical section) and potentially limiting all other threads. Consequently, delaying the execution of critical section code should be avoided to decrease the overall sequential time of an application.

Fig. 10 compares the critical section latency for three sets of the Bodytrack application: (i) standalone ($\times 1$), with four ($\times 4$), and with eight instances of Bodytrack ($\times 8$). The Y-axis is the percentage of overall critical section time on a given time interval (X-axis). The X-axis comprises the last value until the current value, except for the first case, where it starts at zero and goes until 2^{11} ns. For instance, the X value equals to 2^{12} comprises the time spent on a critical section of $[2^{11}, 2^{12})$ ns. As discussed previously in Section VI, all threads of Bodytrack, except the last one, participate in the synchronization scheme; since this example employs 64 threads, up to 63 threads share the same critical section.

We compare the same application on these three scenarios; consequently, the number of accesses into the critical section is approximately the same. However, the time spent into a critical section by a thread is not the same, as the scheduler

[4]Assuming the scheduler shares the same timeslot for all threads of a given application.

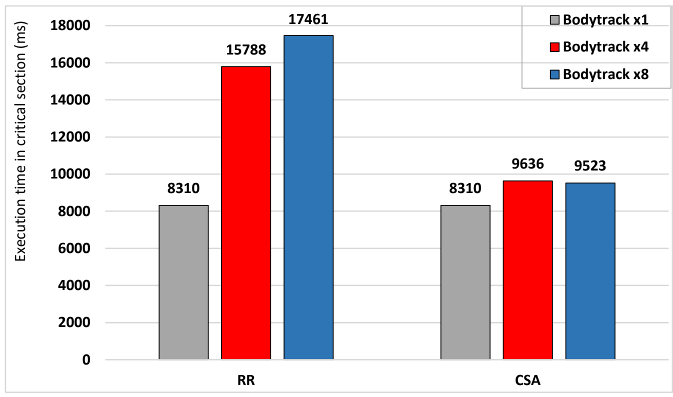


Fig. 11. Overall time spent in critical section for the sum of the work-related mutexes of Bodytrack on RR and CSA-enabled schedulers.

can interrupt the application execution. Fig. 10 shows that as the number of applications increases, the time each thread spent into a critical section also tends to increase since the scheduler does not differentiate execution on a sequential or parallel mode. The figure shows two situations where the time spent in critical sections spikes upward: from 2^{11} to 2^{12} ns and 2^{18} to 2^{19} ns. The first spike results from the application synchronization usage: this interval has the most number of cases for Bodytrack, as shown by the standalone case. The second spike happens only when multiple applications are introduced (Bodytrack$\times$4 and Bodytrack$\times$8). As we use a timeslot of 1 ms, the time spent in the critical section revolves around half this value (i.e., 2^{19}). The threads do not necessarily use the entire timeslot, as they can request to be scheduled out to wait for an event, for instance. Bodytrack has shown significant enough cases of such scenarios that the spike happens around half of the total timeslot. Nonetheless, this experiment shows that a scheduler based on the RR policy affects the synchronization latency and execution time.

Fig. 11 (legend RR) shows the total time spent in the critical section for the same set of applications. As expected, the time spent increases as more applications content for core usage. The behavior shown here by not taking note of the critical sections of parallel applications motivates the proposal of introducing the context of critical sections into the scheduler decisions. Therefore, the gains of utilizing Subutai can be maintained in a massive scheduler contention scenario. We call this proposal the **critical section-aware policy (CSA)**, whose results are depicted in Fig. 11 (legend CSA). The critical section execution time is kept as close as possible to the single application execution by applying the CSA policy.

A. *Critical Section-Aware Policy (CSA)*

We introduce CSA into the scheduler policies for executing critical section code as fast as possible. The policy works as follows. Every time a given thread has CSA enabled and is currently inside a critical section (i.e., holding a mutex), it has priority over the execution of all other threads that are not in the same scenario. In case another thread also has CSA enabled and is inside another critical section, an RR policy is applied to switch between them until either one finishes. Finally, if there are no threads that meet those requirements, an RR policy is applied to switch between the entire application set. We use RR as the baseline scheduler policy, yet more complex policies can also be applied.

Unfortunately, increasing the priority of a given thread over all others without any limitations generates two issues: (i) the scheduler deadlocks if the application also deadlocks; and (ii) it affects negatively on the performance of all other threads (i.e., high unfairness). Therefore, a time limit, defined in Eq. 2, was implemented in the CSA to deal with both issues.

$$CSALimit = (ThrReady + ThrRun - 1) \times (2 \times TS) \quad (2)$$

where $ThrReady$ and $ThrRun$ are the numbers of threads currently in the ready and running states, respectively. For both cases, the idle thread is ignored. TS is the chosen timeslot for the RR policy, generally in milliseconds. For instance, for a scheduler with a sum of 8 threads on the $ThrReady$ and $ThrRun$ states and a TS of $1\mathrm{ms}$, when one of these threads gains CSA priority, its time limit is $14\mathrm{ms}$.

$CSALimit$ has a direct proportionality between a parallel application execution time and the scheduler's unfairness. In other words, a high $CSALimit$ value will produce a fast parallel application execution for an unfair scheduler, and the opposite is also true. As we aim to keep the scheduler's fairness, we chose a limit that accelerates parallel application without increasing the scheduler's unfairness. Eq. 2 is a first empirical proposal, but a dynamic limit can be used to rebalance the CSA policy according to the scheduler profile. We chose the limits defined in Eq. 2 as it restricts the delay on other threads at most three times compared to the RR policy. When all threads are running on the RR policy, the maximum delay is $(ThrReady + ThrRun - 1) \times TS$. Thus, the scheduler changes to the RR policy to maintain a fair scheduling, if the execution of the critical section reduces the priority of the other threads. The time limit deals with deadlock situations; however, to avoid livelocks, the scheduler requires a system-specified limit on the use of CSA policy for a given timeframe. Such methodology has been used effectively against other types of scheduler livelocks [36].

The fairness restriction of $CSALimit$ allows accelerating only a subset of critical sections; this is the reason why the critical section time is lower with Bodytrack $\times$8 than with Bodytrack $\times$4 (Fig. 11 - legend CSA). Table VII shows the impact of $CSALimit$ on the Bodytrack application set. The second, fourth, and fifth columns show the number of scheduling requests made outside of any critical section, inside a critical section with CSA enabled, and inside a critical section where RR has been enabled due to $CSALimit$, respectively. The third column shows that all scheduling requests for a critical section are either using CSA or RR policy. Approximately 10% and 8% of the total critical sections had CSA disabled as their time surpassed the $CSALimit$ time on Bodytrack $\times$4 and $\times$8, respectively. Even though we are analyzing the same Bodytrack, while running in a set of 4 and 8 applications, there are some discrepancies in the total number of requests for scheduling due to the use of synchronization primitives. Streamcluster and x264 presented

TABLE VII
IMPACT OF $CSALimit$ ON THE BODYTRACK APPLICATION SET EMPLOYING A TIMESLOT OF 1ms. CS = CRITICAL SECTION.

Application set	Schedule requests (not CS)	Schedule requests (CS)	CSA (CS)	RR (CS)
Bodytrack ×4	305517	CSA (CS) + CSA (RR)	15267	1558
Bodytrack ×8	323379		15274	1274

TABLE VIII
NI, SUBUTAI-HW AND SPM SYNTHESIS WITH 28 NM SOI.

Components	Area (µm^2)	Overhead
Basic NI	13539.23	–
Subutai FSM	2626.21	19 %
SPM	3702.00	27 %
Basic NI + Subutai-HW*	**19867.44**	**46 %**

*Subutai FSM + SPM

shorter critical sections on our experimental results, and they never triggered the $CSALimit$.

VIII. EXPERIMENTAL RESULTS

We demonstrate our solution results using a two-fold approach. Firstly, the system area and scalability of our solution are evaluated through an RTL implementation of Subutai-HW. Secondly, the system performance and scheduler are evaluated through architecture simulation and parallel applications from the PARSEC benchmark. Like Butko et al. [37], we employ the Gem5 simulator [38] to produce synchronization points of the applications; next, we feed this information into an inhouse SystemC simulator [32], which enables us to collect experimental results. We run applications with and without Subutai: the former will henceforth be called Subutai, and the latter SW-only (i.e., Linux Kernel).

A. Area

Subutai-HW comprises a register-based NI, an FSM for synchronization control and linked pointer manipulation, and a 1 KiB SPM to store metadata and events. We use a very basic NI with 32-bit links, packing and unpacking logic, no virtual channel and 2 I/O buffers of 16×32 bits. It is worth noting that using HW synchronization operations releases valuable memory and cache space that would otherwise be required. Besides, the memory requirement is negligible if compared to a typical processor cache (less than 10%, if the cache size is 16 KiB). Table 8 summarizes the synthesis results showing our solution increases by 46% the basic NI area, including the local SPM; however, the overhead is amortized when the entire chip area is considered. For instance, using the Patel et al. [15] chip area of 400mm^2, the percentage of total area consumption of Subutai-HW is $\frac{64\times0.00632821}{400} = 0.101\%$, while the enhanced NI is $\frac{64\times0.01986744}{400} = 0.317\%$ for 64 cores. We synthesized all hardware elements using Synopsis DC [29] with 28 nm Silicon on Insulator (SOI) technology and 1 GHz clock frequency. Additionally, the SPM was synthesized with Cut Explorer [39].

B. State-of-the-Art Area Comparison

We compare our solution to those related work that provide enough data about the absolute area consumption (i.e., not in percentages) and technology used. Table IX depicts the area consumption of five hardware-based solutions. For a fairer analysis of the area consumption of each solution, we divided the total area consumed by the estimated number of cores in the system (i.e., area per core).

Subutai is second-to-last in terms of area consumed per core in the system. Additionally, Subutai and HTM have an additional area requirement per core; i.e., HTM needs to change the first cache level of the system for its functionality, and Subutai needs an SPM memory for synchronization handling. Even so, Subutai is third-to-last in terms of area consumption when both areas are combined. The hardware of Abellán et al. [13] has the overall smallest consumption as it is mainly comprised of wires and controllers. The last line of Table IX shows the estimation of area consumption for a 400mm^2 chip [15] for the same set of related work. Subutai only consumes approximately 0.1% of the total chip. Once again, it is third-to-last in overall area consumption.

C. Acceleration of Single Parallel Application

Fig. 12 shows the results obtained for the three PARSEC applications analyzed in this work. We analyze the entire application execution but plot the results for two threads for each application: the master thread (T_0), responsible for global synchronization, and a worker thread instance (T_7). Besides, the results are divided into two: synchronization operations and processing. The former aggregates all calls to PThreads (e.g., mutex lock), while the latter collects the processing needed by the application. NoC communication and Subutai-HW latencies did not contribute significantly for the execution time; thus, they are not visually perceivable on the figure, although they are present. Nonetheless, the figure shows that our solution reduces the application total time by handling synchronization faster.

From the designer point-of-view, the master thread (T_0) shows the effective speedup, as it is responsible for initializing and finalizing the application. Bodytrack achieved a speedup of $1.78\times$, and $1.77\times$ for 32 and 64 cores, respectively. Streamcluster achieved a speedup of $2.71\times$, and $2.20\times$ for the

TABLE IX
STATE-OF-THE-ART AREA CONSUMPTION.

	HTM [18]	MCAS [15]	Abellán et al. [13]	Notifying Memories [19]	**Subutai**
Area per core (mm^2)	0.32800	0.01824	0.00022	0.00534	**0.00262**
Additional area per core (mm^2)	0.01560	No	No	No	**0.00370**
Target Frequency (GHz)	Not addressed	3.40	0.62	0.50	**1.00**
Target System	8-core	32-core	64-core	12-core	**64-core**
Technology (nm)	65	14 (scaled)	45	65	**28**
Technique	Estimation	Synthesis	Synthesis	Synthesis	**Synthesis**
Overhead for a 64-core 400mm^2 chip	5.497 %	0.291 %	0.003 %	0.008 %	**0.101 %**

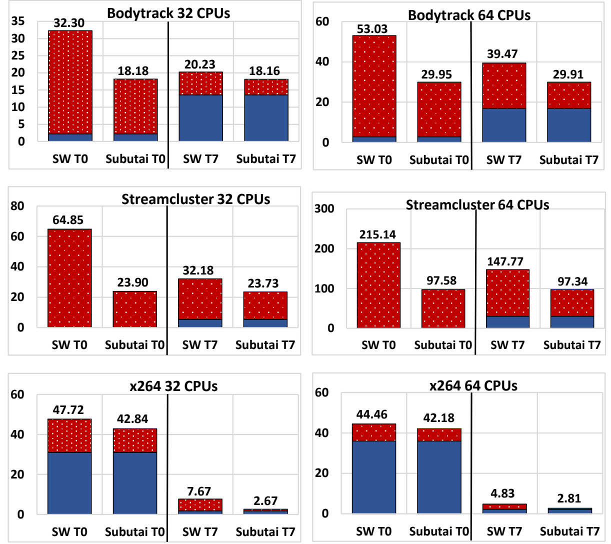


Fig. 12. Experimental results showing acceleration for a single parallel application. Values are in seconds of execution; the dotted red color is the sum of synchronization operations; the flat blue color is processing time.

same core set. Finally, x264 achieved a speedup of $1.11\times$ and $1.05\times$ for the same core set. Therefore, our solution achieved a speedup of $1.77\times$, on average. Table VI displays the number of synchronization calls, explaining the speedup difference; for instance, Streamcluster requires, roughly, 18, 23, and 31 times the equivalent of Bodytrack for 16, 32, and 64 cores, respectively. Thus, we can better optimize worker threads, as they are the ones using these primitives. The results also show that Bodytrack and Streamcluster are not scalable to 64 cores. Southern et al. [40] have independently corroborated this limitation as well. Our solution works the same regardless of the application scaling – as will be shown with a producer-consumer application on Section VIII-E.

The x264 application does not employ barriers because it uses hundreds of synchronization variables instead of dozens (Table V), and it does not have a logical dependency that involves all threads; therefore, x264 has less contented synchronization primitives. While Bodytrack and Streamcluster utilize synchronization in all worker threads, some of the worker threads of x264 have almost no synchronization; in turn, the application is not penalized with significant synchronization overhead. Another application like x264 from PARSEC, named Facesim, is available in [32], and it shows similar speed up results: $1.10\times$ and $1.27\times$ for 32 and 64 cores.

Our solution provides less direct benefit to x264 compared to the other two applications since it is designed to accelerate synchronization overhead. In other words, when synchronization primitives are not used to control most threads, their acceleration may not affect significantly the execution time since the synchronization may not be in the critical path.

Since we aim for legacy code compatibility, no changes have been made to any applications, either to increase the use of PThreads or to insert metadata for Subutai. Therefore, we target scenarios of running multiple applications to improve the speedup of our solution further.

D. Accelerating Multiple Parallel Applications

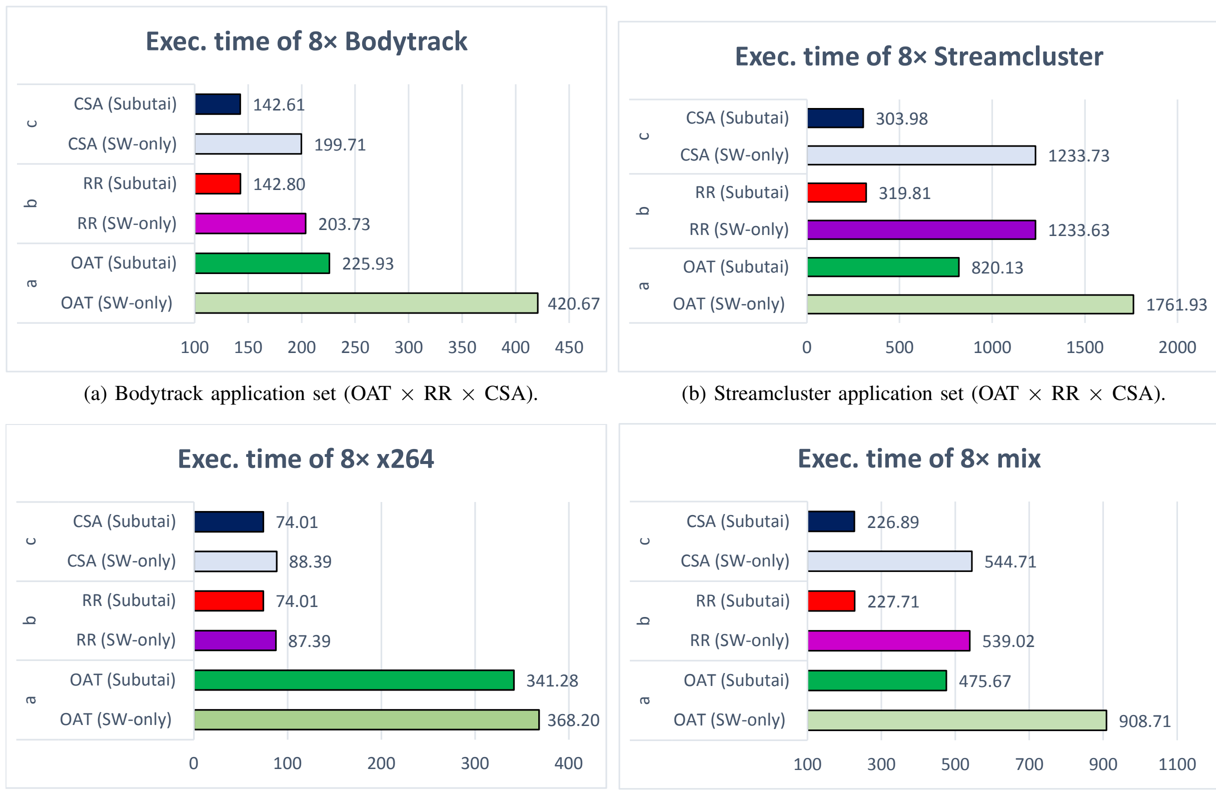
Fig. 13 displays the experimental results organized into sets of eight applications each: (a) eight instances of Bodytrack, (b) eight instances of Streamcluster, (c) eight instances of x264, and (d) a combination of 3, 2 and 3 instances of Bodytrack, x264, and Streamcluster, respectively. All applications have been set to use 64 threads and cores without restriction regarding mapping threads to cores.

Figs. 13a to 13d illustrate the entire execution time in seconds of an application set (i.e., from initialization to termination of all applications), comparing RR, CSA, and a One Application at a Time (OAT) scheduler. The latter scheduler is used for representing a mono application system (i.e., OAT can only execute one application). Lines `a` in Figs. 13a through 13d show that Bodytracks, Streamclusters, x264s, and mixed application sets have accelerations of $1.86\times$, $2.13\times$, $1.07\times$, and $1.91\times$, respectively, when running with Subutai compared to SW-only implementations with an OAT scheduler.

Additionally, lines `b` and `c` of Figs. 13a to 13d show that placing these applications in a competitive scheduling scenario increases the gains further because the idle time for a given application can be used as working time for another application; i.e., comparing CSA with OAT the speedups for Bodytracks, Streamclusters, x264s, and mixed applications are $1.58\times$, $2.69\times$, $4.61\times$, and $2.09\times$, respectively. The SW-only implementation has also presented gains, but the execution time of it is always higher compared to Subutai for the set of applications analyzed here. For the Streamcluster and mixed applications (Fig. 13b and 13d), executing them on Subutai with an OAT scheduler is faster than executing them on SW-only with either scheduler policies used in this work.

Table VII shows the impact of a scheduling policy restricted to critical sections is limited, as for an application such as Bodytrack, this section is approximately 5.12% of the total execution time. For Streamcluster and x264 application sets, not shown in Table VII, the critical section scheduling requests are 0.26% and 9.29% of the total number of requests, respectively. The set of Streamclusters with the CSA-enabled scheduler presented the highest speedup when compared to the same set of applications with an RR scheduler. Bodytrack and x264 presented a less significant speedup of less than $1.01\times$.

Table VI shows that Streamcluster has by far the most significant number of synchronization events of the application set. The number of synchronization events is a crucial factor for both Subutai and CSA in terms of their capacity to accelerate applications. For Subutai, these events are accelerated through the HW/SW co-design proposed by our work. For CSA, the same set of events are the only moments where it can apply its policy. Additionally, CSA relies on the premise that accelerating critical sections will decrease the overall execution time. This premise works well on barrier-based workloads, such as Streamcluster and Bodytrack, where the application is always working on the worst-case scenario (i.e., all worker threads blocked waiting for the slowest thread to join the barrier). However, pipeline applications, such as x264, can start working on new data as soon as the first thread



(a) Bodytrack application set (OAT × RR × CSA). (b) Streamcluster application set (OAT × RR × CSA).

(c) x264 application set (OAT × RR × CSA). (d) mix application set (OAT × RR × CSA).

Fig. 13. Execution in seconds for multiple eight-application sets (lower is better) - (Exec = Execution).

TABLE X
UNFAIRNESS METRIC FOR CSA AND RR SCHEDULERS (LOWER IS BETTER).

Application set	SW-only		Subutai	
	RR	CSA	RR	CSA
Bodytrack ×8	1.04	1.04	1.16	1.15
Streamcluster ×8	1.11	1.11	1.19	1.19
x264 ×8	1.27	1.24	1.12	1.20
mix ×8	2.00	1.71	1.88	1.83

has finished; therefore, CSA has a lesser impact on such applications.

Table X presents the unfairness metric. For all cases, CSA either maintains or decreases the unfairness of the scheduler for the application set, except for x264. Nonetheless, Fig. 13c shows that x264 has the same overall execution time in both cases. Consequently, these results indicate that the use of the CSA policy keeps the fairness of the baseline scheduler.

E. Synthetic Benchmark

The results presented in the previous sections provide a systemic view of Subutai, but they do not convey the optimization in the synchronization itself. The lack of a microcosm view happens because the applications use at least thousands of synchronization primitives during their execution. Consequently, we employ a one producer many consumers synthetic application encompassing a few calls to the three synchronization primitives (mutex, barrier, and condition) using six threads. Table XI shows the average absolute time of Subutai and SW-only for these primitives.

Subutai speeds up significantly every synchronization primitive compared to the SW-only implementation. The comparison is made from the application perspective; for instance, the condition broadcast and mutex unlock operations have no response packet; consequently, Subutai can return to the application immediately after the request packet is sent. Thus, the processing is offloaded to the HW, and the primitive is handled faster from the caller perspective. The SW-only implementation depends on the following costs to handle synchronization primitives (Fig. 3): (i) context switching; (ii) synchronization for queue operations; and (iii) kernelspace switching. Item (i) is reduced in Subutai by using a distributed OS. As stated in Section III-A, we can use a faster context switch with a distributed OS. The faster OS is useful for functions that are blocking, and every group handled by Subutai has these functions. Item (ii) is reduced by offloading

TABLE XI
RESULTS FOR PRODUCER-CONSUMER APPLICATION.

Primitive	Type	Avg. SW	Avg. Subutai
Mutex	Lock empty	1537 ns	127 ns
	Lock queued	64178 ns	916 ns
	Unlock	4400 ns	60 ns
Barrier	Wait (released)	102467 ns	1183 ns
Condition	Broadcast	25209 ns	60 ns
	Queued	42844 ns	1022 ns

all queue operations to hardware. Finally, item (iii) is not present in our OS. Subutai adds the cost of I/O operations to deal with Subutai-HW (Fig. 4), which is not present in the SW-only solution. Nonetheless, these factors explain the gains shown in Table XI.

IX. Conclusion

This paper presents Subutai, an HW/SW co-design solution for accelerating legacy and novel parallel applications through data synchronization. Unlike other synchronization solutions [1] [9] [15], our approach does not require any user-level modification, such as source code changes. Subutai overrides the shared library of PThreads while maintaining its functionality and API. Ergo, any binary using PThreads for data synchronization can benefit from the proposed solution.

Subutai relies on hardware-handled operations to accelerate common synchronization techniques found on parallel applications. By doing so, the overall execution time speeds up to $1.77\times$, on average. Besides, we show that our solution is efficient in the general case of multiple applications sharing computing resources as we propose the CSA scheduling policy to accelerate applications further on a resource-contention scenario by providing priority to threads that are currently running in a critical section. We have implemented this policy using an approach that improves the balance of the scheduler (i.e., low unfairness), making the policy highly portable across different scheduling techniques. Even with such limitations, we achieved a speedup of up to $4.61\times$ for shared-memory parallel applications.

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior — Brasil (CAPES) — Finance Code 001.

References

[1] C. DeLozier, A. Eizenberg, B. Lucia, and J. Devietti, "SOFRITAS: Serializable Ordering-Free Regions for Increasing Thread Atomicity Scalably," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 286–300.

[2] P. McKenney, *Is Parallel Programming Hard, And, If So, What Can You Do About It?*, kernel.org, Corvallis, USA, 2019.

[3] E. Sperling, "How Much Will That Chip Cost?," http://semiengineering.com/how-much-will-that-chip-cost/, Aug. 2020.

[4] P. McKenney, S. Boyd-Wickizer, and J. Walpole, "RCU Usage In the Linux Kernel: One Decade Later," https://pdos.csail.mit.edu/6.828/2017/readings/rcu-decade-later.pdf, Aug. 2020.

[5] S. McConnell, *Code Complete, Second Edition*, Microsoft Press, Redmond, USA, 2004.

[6] K. Fürlinger, T. Fuchs, and R. Kowalewski, "DASH: A C++ PGAS library for distributed data structures and parallel algorithms," *CoRR*, vol. 1610.01482, 2016.

[7] IEEE, "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Rev. of IEEE Std 1003.1-2008)*, pp. 1–3951, Jan 2018.

[8] R. Cataldo, R. Fernandes, K. Martin, J. Sepulveda, A. Susin, C. Marcon, and J.-P. Diguet, "Subutai: Distributed Synchronization Primitives in NoC Interfaces for Legacy Parallel-applications," in *Annual Design Automation Conf. (DAC)*, 2018, pp. 83:1–83:6.

[9] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole, "User-Level Implementations of Read-Copy Update," *IEEE Trans. on Parallel and Distributed Systems*, vol. 23, no. 2, Feb 2012.

[10] H.-J. Boehm, "Reordering Constraints for Pthread-style Locks," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007, pp. 173–182.

[11] M. France-Pillois, J. Martin, and F. Rousseau, "Optimization of the GNU OpenMP Synchronization Barrier in MPSoC," in *International Conference of Architecture of Computing Systems (ARCS)*, Apr. 2018, pp. 57–69.

[12] R. Sivaram, C. Stunkel, and D. Panda, "A Reliable Hardware Barrier Synchronization Scheme," in *Intl. Parallel Processing Symp. (IPPS)*, Apr 1997, pp. 274–280.

[13] J. Abellán, J. Fernández, M. Acacio, D. Bertozzi, D. Bortolotti, A. Marongiu, and L. Benini, "Design of a Collective Communication Infrastructure for Barrier Synchronization in Cluster-Based Nanoscale MPSoCs," in *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2012, pp. 491–496.

[14] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase, "Hardware Synchronization for Embedded Multi-Core Processors," in *IEEE Intl. Symp. of Circuits and Systems (ISCAS)*, May 2011.

[15] S. Patel, R. Kalayappan, I. Mahajan, and S. Sarangi, "A Hardware Implementation of the MCAS Synchronization Primitive," in *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2017, pp. 918–921.

[16] T. Gangwani, A. Morrison, and J. Torrellas, "CASPAR: Breaking Serialization in Lock-Free Multicore Synchronization," *SIGPLAN Not.*, vol. 51, no. 4, pp. 789–804, Mar. 2016.

[17] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and Limitations of Commodity Hardware Transactional Memory," in *Intl. Conf. on Parallel Architecture and Compil. Tech. (PACT)*, 2014, pp. 3–14.

[18] A. Shriraman, S. Dwarkadas, and M. Scott, "Flexible Decoupled Transactional Memory Support," in *Intl. Symp. on Computer Architecture (ISCA)*, 2008, pp. 139–150.

[19] K. Martin, M. Rizk, M. Sepulveda, and J.-P. Diguet, "Notifying Memories: a case-study on Data-Flow Applications with NoC Interfaces Implementation," in *Design Automation Conf. (DAC)*, 2016, pp. 1–6.

[20] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly Media, 1 edition, July 2007.

[21] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev, "Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated," in *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2011, pp. 487–498.

[22] C. Kirsch, M. Lippautz, and H. Payer, "Fast and scalable, lock-free k-FIFO queues," in *Intl. Conf. on Parallel Computing Technologies (PaCT)*, 2013, pp. 208–223.

[23] C. Cascaval, C. Blundell, M. Michael, H. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software Transactional Memory: Why Is It Only a Research Toy?," *Queue*, vol. 6, no. 5, pp. 40:46–40:58, Sept. 2008.

[24] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Intl. Symp. on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[25] J. Hennessy D. Patterson, *Computer Organization and Design: The Hardware Software Interface [RISC-V Edition]*, vol. 1, Morgan Kaufmann, 1st edition, 2017.

[26] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *ACM SIGOPS Symp. on OS Principles (SOSP)*, 2009, pp. 29–44.

[27] Ulrich Drepper, "Futexes Are Tricky," https://cis.temple.edu/˜giorgio/cis307/readings/futex.pdf, Aug. 2020.

[28] Mentor, "ModelSim ASIC and FPGA Design – Mentor Graphics," www.mentor.com/products/fv/modelsim/, Aug. 2020.

[29] Synopsys, "DC Ultra," www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html, Aug. 2020.

[30] Linux man page, "futex(2)," https://linux.die.net/man/2/futex, Aug. 2020.

[31] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," in *Ottawa Linux Summit*, 2002, pp. 479–495.

[32] R. Cataldo, *Subutai: Distributed Synchronization Pritimives for Legacy and Novel Parallel Applications*, Ph.D. thesis, Université Bretagne-Sud, Lorient, France, 2019.

[33] C. Bienia, S. Kumar, J. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," Tech. Rep., Princeton University, 01 2008.

[34] R. Cataldo, "Design and Exploration of 3D MPSoCs with on-Chip Cache Support," M.S. thesis, Pontifícia Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2015.

[35] A. Garcia-Garcia, J. Saez, and M. Prieto-Matias, "Contention-Aware Fair Scheduling for Asymmetric Single-ISA Multicore Systems," *IEEE Transactions on Computers*, vol. 67, no. 12, pp. 1703–1719, Dec 2018.

[36] G. Nakagawa and S. Oikawa, "Fork Bomb Attack Mitigation by Process Resource Quarantine," in *Intl. Symp. on Computing and Networking (CANDAR)*, 2016, pp. 691–695.

[37] A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, G. Sassatelli, and C. Adeniyi-Jones, "A Trace-driven Approach for Fast and Accurate Simulation of Manycore Architectures," in *Asia and South Pacific Design Automation Conf. (ASPDAC)*, 2015, pp. 707–712.

[38] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[39] STMicroelectronics, "Standard Technology offers at CMP in 2017 Deep Sub-Micron, SOI, and SiGe Processes," https://mycmp.fr/IMG/pdf/2017_cmp_usersmeeting_st_mpw_services.pdf, Dec. 2020, slide 16.

[40] G. Southern and J. Renau, "Deconstructing PARSEC Scalability," *Annual Workshop on Duplicating, Deconstructing and Debunking*, vol. 1, no. 1, pp. 1–10, 2015.