Subutai: Distributed Synchronization Primitives in NoC Interfaces for Legacy Parallel-Applications

Rodrigo Cataldo[1,2], Ramon Fernandes[2], Kevin J.M. Martin[1], Johanna Sepulveda[3], Altamiro Susin[4], César Marcon[2], Jean-Philippe Diguet[1]
[1] Univ. Bretagne-Sud, CNRS UMR 6285, Lab-STICC, Lorient, [2] Pontifical Catholic Univ. of Rio Grande do Sul, [3] Technical Univ. of Munich, [4] Federal Univ. of Rio Grande do Sul
{rodrigo.cataldo, ramon.fernandes}@acad.pucrs.br, {kevin.martin,jean-philippe.diguet}@univ-ubs.fr, johanna.sepulveda@tum.de,cesar.marcon@pucrs.br

ABSTRACT

Parallel applications are essential for efficiently using the computational power of a Multiprocessor System-on-Chip (MPSoC). Unfortunately, these applications do not scale effortlessly with the number of cores because of synchronization operations that take away valuable computational time and restrict the parallelization gains. Moreover, synchronization is also a bottleneck due to sequential access to shared memory. We address this issue and introduce "Subutai", a hardware/software (HW/SW) architecture designed to distribute essential synchronization mechanisms over the Network-on-Chip (NoC). It includes Network Interfaces (NIs), drivers and a custom library of a NoC-based MPSoC architecture that speeds up the essential synchronization primitives of any legacy parallel application. Besides, we provide a fast simulation tool for parallel applications and a HW architecture of the NI. Experimental results with PARSEC benchmark show an average application speedup of 2.05 compared to the same architecture running legacy SW solutions for 36% overhead of HW architecture.

1 Introduction

Parallel applications have become ubiquitous with the huge adoption of MPSoCs in the electronics industry. Besides, the tendency is to continue to deliver high-performance computing by increasing the number of processors, which requires efficient communication architectures, such as a NoC, leading to the NoC-based MPSoC architectures and their challenges. From the parallel application point-of-view, the delay effect is felt on the synchronization primitives used to order global events among communicating processors and memories. The time spent to synchronize such events is one of the most significant parts of the execution time as such events are primarily done sequentially, delaying the application execution. Besides, a huge part of the sequential latency of parallel applications is due to interprocessor communication latency, including cache updates of shared data, and the software implementation of synchronization mechanisms. Unfortunately, Attiya et al. [3] have already formally proved that deterministic structures, such as queues and mutual exclusion, cannot eliminate the use of expensive synchronization in software.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

DOI: https://doi.org/10.1145/3195970.3196124

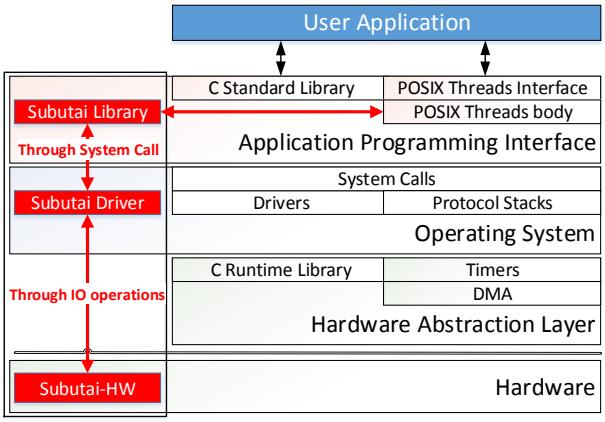


Figure 1: Subutai solution, including library, driver, hardware part and the communication procedures

Although it is feasible to relax the specification of such structures, the consequence is a significant source code redesign. The majority of recent works requires source code meddling, which is notoriously a hard task [13][15]. We developed Subutai, a hardware-software solution that tackles the synchronization overhead by offloading its processing to a new low-latency NI enhanced with dedicated resources. Our approach extends the NI architecture implementing, in a distributed way, the following synchronization primitives: mutex, barrier, and condition. Moreover, we create new types of packets to transport the synchronization primitives among NIs through the NoC. We developed an Application Programming Interface (API) that replaces the underlying synchronization API with the same function interface, and a driver for the Operating System (OS). Therefore, our approach keeps unchanged the legacy code of applications avoiding time-consuming and error-prone redesign; a recompilation is required only for static binaries. Fig. 1 depicts how our solution interacts with the various components of the system.

We demonstrate our solution by employing the Gem5 simulator with the data processing applications Streamcluster and Bodytrack from the PARSEC benchmark. Like Butko et al. [5], we produced synchronization points of them; next, we feed this information to an in-house SystemC simulator, which enables us to collect experimental results. The main contributions of this paper are: (i) a solution to accelerate application execution without requiring source code modification; (ii) an enhanced NI architecture able to compute and accelerate expensive synchronization primitives and be-

Table 1: Related work comparison.

Work	HW/SW Solution	Legacy code compatible	Primitive	Application
[16]	HW	Not addressed	Fault-tolerant barrier	Synthetic
[1]	HW	Not addressed	Barrier	Synthetic
[18]	HW	Not addressed	Locking	Synthetic
[15]	Both	No	Atomic operations	Synthetic
[7]	Both	No	Atomic operations	STAMP suite +Synthetic
[11]	SW	No	Locking	Synthetic + Mandelbrot
[6]	SW	No	Locking	OS Kernel + Synthetic
[12]	HW	Yes	FIFO events	MPEG4-SP
This	**Both**	**Yes**	**Locking, barrier and Condition**	**Synthetic + PARSEC Benchmark**

ing compliant with any NI that has access to a local memory; (iii) a set of APIs for parallel computing architectures; (iv) a trace-based simulation tool to allow fast simulations of real parallel applications; and (v) synthesis of the enhanced NI architecture with a 28 nm SOI technology.

2 Related Work

Accelerating parallel applications using hardware has been studied at least since the '90s. Sivaram et al. [16] propose a fault tolerant HW-based barrier synchronization. Their design uses a tree structure to sum up intermediate values for decreasing the number of packets injected into the network. Their work is complementary to our solution. Abellán et al. [1] explore three HW-based barrier architectures and integrate them on the OpenMP programming model. Unfortunately, their results are for synthetic applications only. Stoif et al. [18] implement an arbiter on FPGA that guarantees mutual exclusion to a portion of the shared memory area and a HW-based barrier that provides sound results; however, their work does not implement full barriers and conditions, and it is limited to simple test cases instead of real applications.

Patel et al. [15] propose a special HW instruction to change multiple memory positions atomically, optimizing the synchronization process. This instruction is an extension of the currently widely available compare-and-swap instruction, which is similar to the hardware transactional memory approach [7] where the developer delimits atomic blocks of codes. For both cases, if the code cannot be executed atomically, the developer has to re-execute the code. Unfortunately, this procedure requires lock-free parallel applications that are difficult to design and debug. Software-oriented solutions, like the ones presented in [11] and [6], focus on relaxing the constraints that force the use of expansive synchronization. Kirsch et al. [11] 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. [6] describe a synchronization technique based on the publish-subscribe mechanism called Read-Copy Update (RCU). Parallel applications that rely on RCU have to deal with stale data. These approaches suffer from the same problem described earlier: the burden of adapting the code to these solutions is passed on to the developer. Finally, Martin et al. [12] propose the Notifying Memories concept to reduce communication latencies introduced in the NoC by pruning useless memory accesses. This concept is applied to dataflow applications only. Our work is also based on NI, but targeting shared-memory systems.

Table 1 positions our work in the current state of the art. To the best of the authors' knowledge, this is the first work for dealing with multiple synchronization primitives in hardware while retaining the source code application.

3 NoC-Based MPSoC Architecture

Fig. 2 depicts the HW elements of our target scalable architecture. Each core includes a dedicated Level 1 (L1) cache, subdivided into data and instruction, and a Level 2 (L2) cache. The core communicates directly with the NI through I/O operations. The main memory of the system is kept off-chip. A packet-based NoC, implemented with a mesh topology, handles the communications. Fig. 2 also shows the modifications required to the NI for handling the synchronization primitives. The hardware architecture of Subutai is detailed in Sec. 5. Tasks share memory space, but the target architecture inhibits excessive caching by employing a distributed approach to the OS. This OS replication allows us to avoid expensive memory barrier into the scheduler [9], which is crucial for our solution. A faster scheduling implies that threads can wake up more efficiently, as shown in Sec. 6.2.

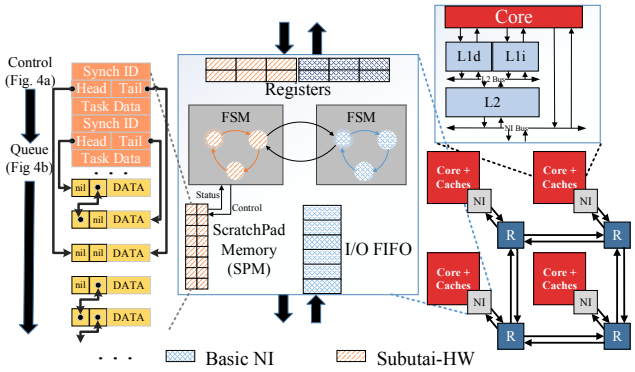


Figure 2: Schematic representation of the NoC-based MPSoC architecture and the Subutai-HW

4 API Implementation and HW/SW Communication

As a proof-of-concept, we chose to overwrite the implementation of the POSIX Threads (PThreads) library, as it is one of the most well-known interfaces used in parallel applications. We focused on three of the four main groups of PThreads operations: mutex, barrier, and condition handling. Thread events (create, exit, join) are not on the critical path and so can be achieved 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 variable. If the variable is already locked, the calling thread is blocked. Otherwise, this operation returns with the variable locked by the calling thread. Unlocking is a non-blocking function that changes the variable state and, if there are any waiting threads, wakes up previously blocked functions. 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. When all participating threads reach the barrier, all threads are woken up. The condition group contains three functions: *wait*, *signal* and *broadcast*. Wait is an unconditionally blocking function that puts the thread on a waiting list for a condition event. It requires that designers previously lock a mutex variable and pass a reference to it. Then, the wait function unlocks the mutex once it finished its work. Next, when the thread is woken up, it reacquires the mutex. The mutex dependency is due to race conditions. The signal and broadcast are non-blocking functions that wake up one and all threads respectively waiting for a condition event. Mutex, in these cases, is optional.

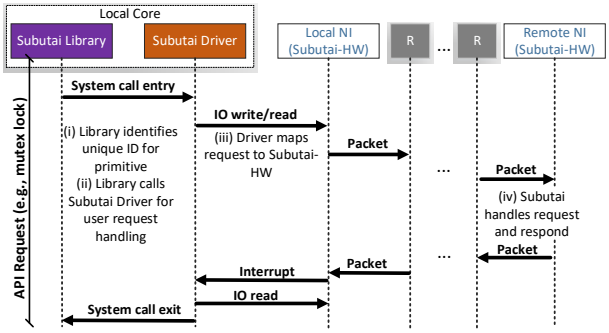


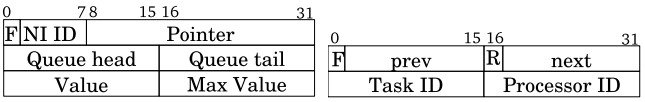
Figure 3: The main message path chart of Subutai

For all groups, one or more queues are required to record blocked threads. Condition functions need to handle two queues because of the mutex dependency. Barrier and mutex functions deal with only one queue. Besides, they have non-blocking functions that allocate and deallocate variables. This work replaces the handling of these operations from an entire SW solution to a HW/SW approach. Fig. 3 depicts the communicating flow of our system. First, the application makes a PThreads interface request; the Subutai library identifies the unique ID for this primitive and passes it on to the driver, along with the interface request. Depending on the implementation, the driver writes to either registers or a memory that NI has direct access to (DMA). Next, the driver writes in a control register to inform the NI of a command and waits for an interrupt to receive the remote response. 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. There are two complementary scenarios for Fig. 3. One when there is no response packet and no backward procedure; thus, the driver returns immediately after writing in the control register. The other one happens when the driver accesses the local Subutai-HW. The same procedure is followed, but without injecting packets into the NoC. An incremental counter per core determines the NI that hosts a synchronization primitive. The first primitive is hosted in NI_0; the second one in NI_1, and so on following a fairness method. Other dynamic allocation strategies can be further studied, but this is out of the scope of the paper.

5 Subutai-HW

Subutai-HW extends the NI for handling synchronization primitives. As shown in Fig. 2, its main components are: (i) a Finite State Machine (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. We have implemented and validated the HW architecture by RTL simulation.

Subutai-HW employs double-linked queues to record events, as shown on the left-hand side of Fig. 2. As an alternative to a garbage collector, the double-linked queues allow Subutai-HW to employ a dynamic allocator to reduce memory consumption to the minimum at the cost of 33 additional sub-states on its FSM. The queue manipulation is based on the futex implementation of the Linux Kernel [8]. Subutai-HW operates using two record information structures. The first one, shown in Fig. 4a, records the synchronization primitives' metadata. The first 32-bit field is the only one known by software and is employed as a unique ID of this primitive. However, for Subutai-HW, the first bit F is employed to allocate/deallocate this structure. The next 7-bit field is the unique ID for the NI on the system *NI ID*. Lastly, the furthest 24-bit is used as a *pointer* to itself; we



(a) Control structure. (b) Queue structure.

Figure 4: Subutai-HW

employ this technique to avoid the cost of searching for an entry every time a new request has arrived. The next 32-bit field is the *head* and *tail* of the double-linked queue implemented in the second structure. The queue records threads waiting for new events on the primitive. Finally, the last 32-bit field record *values* used for some of the primitives. The first 16-bit is employed to (i) record the thread and CPU that owns a mutex; and (ii) store the current number of threads waiting on a barrier. The furthest 16-bit is applied only for the barrier primitive to record the maximum number of threads allowed in a barrier. Fig. 4b shows the entry to the double-linked queue composed of six fields. The first bit is employed to allocate or deallocate the entry. The *prev* and *next* fields are pointers to the previous and next entries, respectively, or nil if they do not exist. The 16th bit R is reserved and can be used for memory alignment. The last 32-bit field identifies the requesting thread. The *Processor ID* field is padded with zeroes because the NoC packet uses only 8-bit to identify the processor.

The bare minimum memory requirement for SPM is one control entry and 63 queue entries, regarding a 64-processor architecture as described in Sec. 3. Since we have to record up to $p-1$ processors, the minimum SPM size is $(1\times96+63\times64)/8 = 516$ bytes. Note that Subutai-HW is incorporated into every NI; consequently, we handle up to 64 primitives even with minimum sizing. For our experimentation, we use an SPM of 1 KiB (4 control entries and 122 queue entries) that handle up to 256 primitives. A double-linked list of events is employed to allocate dynamically queue entries, allowing Subutai to consume memory on demand. A static allocator, on the other hand, would not be able to handle more than one control entry with only 122 queues ($< 2\times63$) – since the worst case scenario is 63 queues per entry, as shown earlier. Thus, a static solution would be either too limited or a waste of memory resource.

Table 2 shows the latencies of the states as dependent on the Subutai-HW cycle c, SPM latency m, number of synchronization primitives handled n, and maximum number of threads on a barrier max. Each memory operation can either be a write or read in SPM in a given cycle. The table is organized as follows. The first column identifies the Subutai-HW state. The second and third columns identify the fastest and slowest 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, we can inject

Table 2: Latency of Subutai-HW states. c = cycle latency, m = memory latency, n = variables handled by Subutai-HW, max = threads on a barrier.

State	Best response time	Worst response time	Packet Injection
Allocation	$4m + 1c$	$(n \times 1m) + 3m + 1c$	$(n \times 1m) + 1m + 1c$
Deallocation	$3m$	$3m$	None
Mutex Lock	$2m + 1c$	$11m$	$2m + 1c$
Mutex Unlock	$2m$	$10m + 1c$	$2m + 1c$
Barrier Wait	$7m$	$max \times (11m + 3c) + 1m + 1c$	$1m1c + 12m4c + 23m7c... = 1m+1c+max\times(11m+3c)$
Condition Wait	$5m + 1c$+ Mutex Unlock	$10m + 1c$+ Mutex Unlock	None
Condition Broadcast	$1m$	$18m + 1c$	$11m + 1c$
Condition Signal	$1m$	$29m + 2c$	$11m + 1c$

packets before we have finalized processing the requests. Additionally, some primitives (e.g., *Deallocation*) do not need to generate packets at all.

To illustrate the best and worst response time of Table 2, we describe the modeling of *Mutex Lock* state, which is responsible for modeling the *pthread_mutex_lock* operation. The fastest scenario, whose latency is $2m+1c$, happens when the control variable is unlocked. Hence, it requires two memory operations: (i) fetch the control structure (field *Value* from Fig. 4a) to check the owner of the mutex (latency = 1m); and (ii) rewrite this field with the requesting thread (latency = 1m). Finally, NI is notified that a new packet can be injected (latency = 1c). The injected packet is the same as the requesting packet except for the header. The worst scenario takes more time (latency = 11m) because *Mutex Lock* state requires dealing with two queues. 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 the empty queue pointers (takes up to 4 memory operations); then, it writes the requesting thread information into it and the tail information in the primitive (six more memory operations), performing 11 memory operations in total. We do not describe the latency models for the other Subutai-HW states in detail; however, they follow a similar procedure.

Table 3 shows the resulting latency model used for this work. 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 a uniform access taking 2 cycles [19], 4 control structures and 122 queue positions. We also considered 4 ns for an entry latency (3 flits of 32 bits and 1 cycle to take a decision) and 1 ns for an exit latency (1 cycle to set a flag) to reach any state shown in Table 3.

Table 3: Subutai-HW latencies with parameters set to c=1ns, m=2ns, n=4, max=63, entry=4ns, exit=1ns

State	Best response time (empty queue)	Worst response time (queued)	Packet Injection	
			Best	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

The latency required to release threads on a barrier exceeds one thousand ns. However, this latency is due to the queue size of threads waiting on the barrier and does not represent the packet injection latency. Therefore, some of the threads execute much earlier than the total value. As shown in the last column, the packets are injected periodically at every 25 ns, except the first packet, which is injected in 7 ns. Thus, the total number of cycles is 1583 ns, which is composed by the following parameters: $entry + exit + 1m + 1c + max \times (11m + 3c)$.

The *Condition Broadcast* and *Condition Signal* states present interesting latency results. At first glance, it would seem more reasonable that releasing one thread (*Signal*) would be faster than releasing all threads (*Broadcast*). However, it does not happen 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 one thread is truly released since a mutex is associated with it. Therefore, the broadcast state avoids the scenario previously described

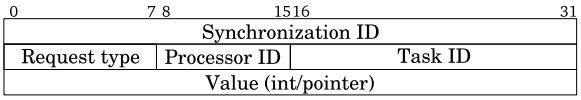


Figure 5: Fields of Subutai packet

for the barrier state – only the owner of the mutex will be released.

Subutai-HW includes six 32-bit and three 1-bit registers; three are used for the packet fields (see Fig. 5) and six more to: (i) handle empty queue entries; (ii) memory swapping operations; and (iii) control flags to receive and send packets. For receiving and sending packets, Subutai-HW reuses the already available registers of the NI. The packet structure is combined with the recorded information on the two control structures described earlier (Fig. 4) to handle any request.

The synthesis of Subutai-HW was achieved using Synopsis DC with an ST SOI 28 nm technology with 1 GHz clock frequency. The SPM area was calculated by CACTI 6.5 [14] using a 32 nm SRAM technology. Subutai-HW comprises of a register-based NI, an FSM that controls synchronization and manipulates linked pointers, and a 1 KiB SPM to store metadata and events. We use a basic NI with 32 bits links, packing and unpacking logic, no virtual channel and 2 I/O buffers of 16×32 bits. Besides, it is worth noting that using HW synchronization operations releases valuable memory and cache space that would otherwise be required. Additionally, the memory requirement is negligible when compared to a typical processor cache (less than 10%, if cache size is 16 KiB). Table 4 summarizes the synthesis results showing that our solution increases by 36% the basic NI area including the local SPM. However, the overhead can be negligible compared to the chip area; for a 400 mm^2 chip [15], 64 Enhanced NI have an overhead of 0.03%.

6 Experimental Results

6.1 Validation Methodology and Exp. Setup

Considering the prohibitive simulation time when using Gem-5 for simulating complex applications (billions of ns) with a large number of processors and distributed OS, we choose an approach that extracts only the useful information of the application and uses a high-level model to measure the speedup provided by our solution. Fig. 6 shows the methodology flow of our experimental setup in four steps.

The first step is the application execution using Gem5 in full-system simulation mode to provide an accurate characterization of the applications. The second step is the application execution trace, from start to finish, as the sequential portions of code can hinder the real speedup of a parallel application [17]. The application trace, which was based on [5], provides the execution times among synchronization calls, and the number and execution time of each synchronization call. The trace is further annotated with every synchronization primitives – so that we can simulate these functions accurately. Moreover, we make sure to employ synced POSIX clocks for recording each thread start. The third step is the execution of our NoC-based MPSoC simulator

Table 4: Synthesis results for Subutai-HW using 28 nm SOI (logic) and 32 nm CACTI ITRS LSTP (memory) technologies.

Component	Area (μm^2)	Technology	Overhead
Basic NI	13539.23	28 nm	-
Subutai FSM	2626.21	28 nm	19 %
SPM	2269.97	32 nm	17 %
Enhanced NI (Basic NI + Subutai-HW)	**18434.60**	**28/32 nm**	**36 %**

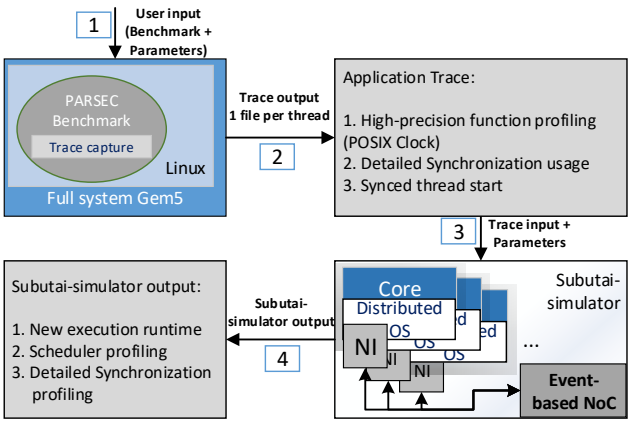


Figure 6: Validation methodology

called Subutai-simulator (modeled in SystemC). The simulator reads the traces to generate tasks in the OS. Then, these tasks mimic the execution of the applications, considering execution times from the traces, and execute the synchronization functions. We reproduce NoC communication, queues and hardware latencies in our SystemC environment. Additionally, we adapted an existing NoC implementation to be event-based; otherwise, the simulation would be far too slow. The NoC was set up to the configuration of Gem5's NoC (Garnet) [2]. The OS latencies were extracted from FreeRTOS [10]. The processing cores are clocked at 1 GHz and are kept the same for both Gem5 and our simulation. Thus, our solution does not speed up any application computation portion. The results from our simulation tool are clustered in the fourth step of Fig. 6 and will be discussed in the following section.

6.2 Results Evaluation

Fig. 7 shows the results obtained on Subutai-simulator for two data processing applications: (i) Bodytrack, which is a computer-vision application that tracks a 3D pose of a markless body, and (ii) Streamcluster, which is a data-mining application that solves the online clustering problem for a stream of input points [4]. We have tested these applications for three system configurations (16, 32, and 64 cores) to visualize the scalability of our solution. For fair comparisons, we consider the same scheduler (round robin) for both. We plot the results for two threads for each application: the master thread (T0), responsible for global synchronization, and a worker thread instance (T7). Fig. 7 illustrates that our solution reduces the application total time by handling synchronization faster.

Table 6 depicts the detailed execution values and the speedup achieved by our solution. These values are (i) Processing time; (ii) Waiting time for all synchronization primitives; (iii) NoC time for Subutai communication; and (iv) Subutai-HW time. For the SW solution, both (iii) and (iv) are handled by software; consequently, their values are computed

Table 5: Number of synchronization-primitives events in Bodytrack and Streamcluster executions

Application	*Type*	Events per number of threads		
		16	**32**	**64**
Bodytrack	Barrier	2,101	4,293	13,416
	Condition	447	750	1,529
	Mutex	9,000	10,472	8,677
Streamcluster	Barrier	208,048	364,480	728,960
	Condition	381	802	1,274
	Mutex	510	1,054	2,142

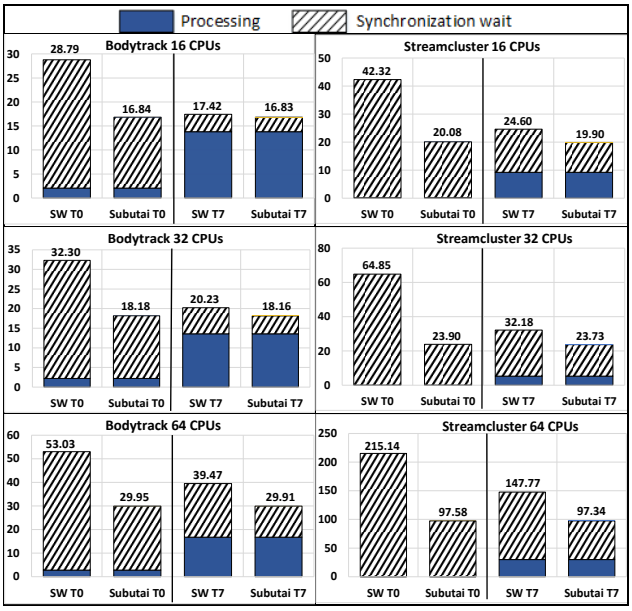


Figure 7: Execution times in seconds of Bodytrack and Streamcluster on SW and Subutai architectures

together with (i) and (ii). The comparison between SW and Subutai solutions shows a significant speedup for our solution in all examined cases.

From the designer point-of-view, the master thread (T0) shows the effective speedup, as it is responsible for initializing and finalizing the application. Bodytrack achieved a speedup of 1.71, 1.78, and 1.77 for 16, 32, and 64 cores, respectively. Streamcluster achieved a speedup of 2.11, 2.71, and 2.20 for the same core set. Therefore, our solution achieved a speedup of 2.05, on average. Table 5 shows the number of synchronization calls, explaining the speedup difference; 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. Table 6 shows for all cases the worker threads of Streamcluster achieve a higher speedup when compared to the worker threads of Bodytrack. The results also show that both applications are not scalable to 64, or even, 32 cores. Southern et al. [17] 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.

The application results give us a systemic view of Subutai, but it does not convey the optimization on the synchronization itself. The lack of a microcosm view happens because our applications use at least tens of thousands of synchronization primitives during its execution. Consequently, we employ a one producer many consumers application encompassing the three synchronization primitives (*mutex*, *barrier*, and *condition*) using six threads. Table 7 shows the average absolute time of Subutai and SW for these primitives.

Subutai significantly speeds up every synchronization primitive compared to the SW implementation (Linux Kernel 2.6). The comparison is made from the application perspective. For instance, the *Condition Broadcast* and *Mutex Unlock* scenarios 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 implementation depends on the following costs to handle synchronization primitives: (i) context

Table 6: Detailed execution time, in seconds, and the speedup for Bodytrack and Streamcluster.

Application	*Type*	*16*				*32*				*64*			
		Thread 0		Thread 7		Thread 0		Thread 7		Thread 0		Thread 7	
		SW	Subutai	SW	Subutai	SW	Subutai	SW	Subutai	SW	Subutai	SW	Subutai
Bodytrack	Processing	*2.1	2.1	*14.0	14.0	*2.2	2.2	*14.0	14.0	*2.8	2.8	*17.0	17.0
	Synch. Wait	27.0	15.0	3.6	3.0	30.0	16.0	6.6	4.6	50.0	27.0	23.0	13.0
	Synch. NoC	-	6.2 E-06	-	9.5 E-06	-	1.4 E-05	-	1.0 E-05	-	3.6 E-05	-	2.0 E-05
	Subutai-HW	-	2.4 E-05	-	4.6 E-05	-	4.5 E-05	-	8.9 E-05	-	8.0 E-05	-	1.8 E-05
	Total	29.1	17.1	17.6	17.0	32.2	18.2	20.6	18.6	52.8	29.8	40.0	30.0
	Speedup	1	1.71	1	1.04	1	1.78	1	1.11	1	1.77	1	1.32
Streamcluster	Processing	*0.1	0.1	*9.2	9.2	*0.1	0.1	*5.4	5.4	*0.2	0.2	* 30.0	30.0
	Synch. Wait	42.0	20.0	15.0	11.0	65.0	24.0	27.0	18.0	215.0	97.0	118.0	67.0
	Synch. NoC	-	2.8 E-06	-	3.0 E-04	-	8.0 E-06	-	3.5 E-04	-	2.3 E-05	-	1.7 E-04
	Subutai-HW	-	3.9 E-06	-	2.8 E-03	-	8.3 E-06	-	6.7 E-03	-	1.7 E-05	-	1.6 E-02
	Total	42.1	20.1	24.2	20.2	65.1	24.1	32.4	24.4	215.2	97.2	148.0	97.0
	Speedup	1	2.11	1	1.24	1	2.71	1	1.36	1	2.20	1	1.52

(*) Processing times of SW solution are a little greater, but the differences are in order of µs, which is insignificant from the order of seconds.

Table 7: Results for producer-consumer applications

Synchronization	**Type**	**Avg. SW (ns)**	**Avg. Subutai (ns)**
Mutex	Lock Empty	1,537	127
	Lock Queued	64,178	916
	Unlock	4,400	60
Barrier	Wait (released)	102,467	1,183
Condition	Broadcast	25,209	60
	Queued	42,844	1,022

switching; (ii) synchronization for queue operations; and (iii) kernel space switching. Item (i) is reduced in Subutai by using a distributed OS. As stated in Sec. 3, we can use a faster context switch with a distributed OS. The faster OS is useful for functions that are blocking, and, as discussed in Sec. 4, every group handled by Subutai has these functions. Item (ii) is reduced by offloading 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, which is not present in the SW solution. Nonetheless, these factors explain the gains shown in Table 7.

7 Conclusion

This paper presents Subutai, an HW/SW approach to accelerate parallel applications with no source code modification. Subutai is composed of an enhanced NI with an efficient local memory management. It includes an OS driver and a library to overwrite the underlying synchronization API. We provide an in-house tool to simulate fast MPSoCs using applications traces. Results show an average application speedup of 2.05 compared to the same architecture running an entire SW solution. Finally, the HW implementation shows a limited overhead of 36% compared to a basic NI, including a 1 KiB SPM that relieves the processor memory and cache from synchronization data. In the next steps, we will take benefit from the saved time to maximize the processor usage in HPC/Cloud domains. It means (i) Subutai-aware cooperative schedulers running multiple applications; and (ii) strategies for dynamic allocation of Subutai resources.

8 References

[1] J. L. Abellán, J. Fernández, M. E. 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 *DATE Conf.*, March 2012.

[2] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *2009 IEEE Int. Symp. on Performance Analysis of Systems and Software*, April 2009.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, 2011.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *17th PACT Conf.*, 2008.

[5] 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 *ASP-DAC Conf.*, Jan 2015.

[6] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE Trans. on Parallel and Distributed Systems*, 23(2), Feb 2012.

[7] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *23rd PACT Conf.* ACM, 2014.

[8] U. Drepper. Futexes are tricky. https://www.akkadia.org/drepper/futex.pdf.

[9] D. Howells, P. McKenney, W. Deacon, and P. Zijlstra. Linux kernel memory barriers. https://www.kernel.org/doc/Documentation/memory-barriers.txt.

[10] D. Howells, P. E. McKenney, W. Deacon, and P. Zijlstra. Freertos. https://www.freertos.org.

[11] C. M. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-fifo queues. In *Int. Conf. on Parallel Computing Technologies*, pages 208–223. Springer, 2013.

[12] K. J. M. Martin, M. Rizk, M. J. Sepulveda, and J.-P. Diguet. Notifying memories: A case-study on data-flow applications with NoC interfaces implementation. In *53rd DAC Conf.*, New York, NY, USA, 2016. ACM.

[13] P. E. McKenney and J. Walpole. Introducing technology into the linux kernel: A case study. *SIGOPS Oper. Syst. Rev.*, 42(5):4–17, July 2008.

[14] N. Muralimanohar et al. Cacti 6.0: A tool to understand large caches. Tech. Report, 2007.

[15] S. Patel, R. Kalayappan, I. Mahajan, and S. R. Sarangi. A hardware implementation of the MCAS synchronization primitive. In *DATE Conf.*, March 2017.

[16] R. Sivaram, C. B. Stunkel, and D. K. Panda. A reliable hardware barrier synchronization scheme. In *11th Int. Parallel Processing Symposium*, Apr 1997.

[17] G. Southern and J. Renau. Deconstructing PARSEC scalability. In *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, 2015.

[18] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase. Hardware synchronization for embedded multi-core processors. In *IEEE ISCAS Conf.*, May 2011.

[19] P. Wang, G. Sun, T. Wang, Y. Xie, and J. Cong. Designing scratchpad memory architecture with emerging stt-ram memory technologies. In *IEEE ISCAS Conf.*, May 2013.