HAL open science

Subutai : Distributed synchronization primitives for legacy and novel parallel applications

Rodrigo Cadore Cataldo

To cite this version:

Rodrigo Cadore Cataldo. Subutai : Distributed synchronization primitives for legacy and novel parallel applications. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Bretagne Sud; Pontifícia universidade católica do Rio Grande do Sul, 2019. English. NNT : 2019LORIS541 . tel-02865408

HAL Id: tel-02865408
https://theses.hal.science/tel-02865408v1

Submitted on 11 Jun 2020

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.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

THESE DE DOCTORAT DE

L'UNIVERSITÉ BRETAGNE SUD
COMUE UNIVERSITE BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Électronique

Par

« Rodrigo Cadore CATALDO »

« SUBUTAI: Distributed synchronization primitives for legacy and novel parallel applications »

Thèse présentée et soutenue à Porto Alegre, le 16 décembre 2019
Unité de recherche : Lab-STICC
Thèse N° : 541

Rapporteurs avant soutenance :

Eduardo Augusto BEZERRA — Professeur, Universidade Federal de Santa Catarina (UFSC)
Marcio KREUTZ — Professeur, Universidade Federal do Rio Grande do Norte (UFRN)
Avelino Francisco ZORZO — Professeur, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

Composition du Jury :

Kevin MARTIN — Maître de conférences, Université Bretagne Sud

Directeur de thèse
Jean-Philippe DIGUET — DR, CNRS

Co-directeur de thèse / Président
César Augusto MISSIO MARCON — Professeur, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

List of Figures

Figure 1.1 – The speedup of a parallel application is severely limited by how much of the application can be parallelized [Wik19]. 18

Figure 1.2 – The essential approaches to parallelizing code. Thick lines depict locks and the flash symbol denote device interrupt [Kå05]. 19

Figure 1.3 – (a) RCU usage over the years [MBWW17]; (b) RCU and Locking usage over the years [MW08]. 20

Figure 1.4 – Snippet of the locking scheme for the TTY subsystem [BC19a] [BC19b]. 23

Figure 1.5 – Execution of reader-writer lock and RCU. The green color is used to represent up to date data [McK19a]. 24

Figure 1.6 – Cost of each production step of multiple generations of transistor technology [Spe19]. 25

Figure 1.7 – Subutai components are highlighted in red in the computing stack. Subutai only requires changes in the (i) on-chip NI, (ii) OS NI driver and (iii) PThreads implementation. Additionally, a new scheduling policy (in blue) is explored in this work as an optional optimization.. 26

Figure 1.8 – Synchronization acceleration with the Subutai solution for different scenarios. 27

Figure 2.1 – Atomic increment scalability on a Nehalem Intel processor [McK19a]. 31

Figure 2.2 – Throughput of different atomic operations on a single memory position [DGT13]. 31

Figure 2.3 – Reordering example from the Intel Manual [Int17] [Pre19b]. 32

Figure 2.4 – The net effect of using lock-based procedures: implicit memory barriers (represented as blue fences) (based on [Pre19a]). 36

Figure 2.5 – Scalability of the Reader-Writer Lock mechanism. A parameter is used to simulate the critical section range in instructions, which ranges from a thousand (1K on the graph) to 100 million (100M on the graph) instructions [McK19a]. 39

Figure 3.1 – The fork-join model of OpenMP [Bar19a]. 47

Figure 3.2 – RCU deletion example [McK19a]. 50

Figure 3.3 – Milliseconds to copy a 10MB file in two threads on a 2GHz Pentium 4 Xeon (lower is better) [Boe07]. 52

Figure 3.4 – Delays to complete the barrier release process for a 16-thread application on a 16-core system (a) without and (b) with optimization [FPMR18]. . 53

Figure 3.5 – Normalized execution time of eight applications from the STAMP benchmark from 1 up to 8 threads; AVG represents the average execution time of all applications [YHLR13]. 56
Figure 3.6 – Comparison of different synchronization schemes for PhysicsSolver [YHLR13]. 57
Figure 3.7 – Simulation time comparison [PKMS17]. 58
Figure 3.8 – Ideas behind the CASPAR architecture [GMT16]. 60
Figure 3.9 – Experimental results for the CASPAR architecture design [GMT16]. . 61
Figure 3.10 – Three barrier synchronization topologies [AFA+12]. 62
Figure 3.11 – Gather phase for (a) CBarrier, (b) GBarrier, and (c) TBarrier [AFA+12]. 63
Figure 3.12 – (a) Optimal frequencies and (b) area overhead running at 600MHz for all analyzed topologies [AFA+12]. 64
Figure 3.13 – The barrier cost of (a) SW and (b) HW implementation in cycles [AFA+12]. 65
Figure 3.14 – Barrier overhead for varied-sized parallel workload [AFA+12]. 65
Figure 3.15 – Target architecture with Notifying Memories [MRSD16]. 67
Figure 3.16 – Classification of packets transported after 10 decoded frames of ice [MRSD16]. 69
Figure 4.1 – The Subutai solution; Subutai components are highlighted in red. . . . 79
Figure 4.2 – Schematic of the target architecture for Subutai; the target architecture is comprised of 64 cores. 81
Figure 4.3 – Schematic representation of Subutai-HW and the NI. The hardware elements required by Subutai-HW implementation are highlighted in orange. 82
Figure 4.4 – Subutai-HW control structure. 83
Figure 4.5 – Subutai-HW queue structure. 83
Figure 4.6 – Subutai's packet format. 87
Figure 4.7 – Futex kernel implementation [Har19]. 91
Figure 4.8 – 2-bit saturating counter for branch prediction [Dia19]. 94
Figure 4.9 – Communication flow of Subutai. 95
Figure 5.1 – The Subutai extensions are highlighted in blue. 97
Figure 5.2 – Core scheduler procedure `switch_to` steps for two Linux kernel versions [Mai19]. 99
Figure 5.3 – Overall time spent in the critical sections for all the work-related mutexes of Bodytrack on a RR scheduler. 100
Figure 5.4 – Overall time spent in mutex queues for all the work-related mutexes of Bodytrack on a RR scheduler. 101

Figure 5.5 – Total execution time spent in critical sections for all the work-related mutexes of Bodytrack on a RR scheduler. 102
Figure 5.6 – Overall time spent in critical sections for all the work-related mutexes of Bodytrack on a RR and CSA-enabled scheduler. 103
Figure 6.1 – Experimental setup for Subutai evaluation. 120
Figure 6.2 – Maximum speedup measured for a 48-core system for each benchmark, region, and input set combination. Full and ROI inputs represent the entire application and parallel portion, respectively [SR15]. 124
Figure 6.3 – Bodytrack's output. 126
Figure 6.4 – Bodytrack's core workflow. 127
Figure 6.5 – The execution time in seconds (10^9 ns) of our application set on SW-only and Subutai solutions for 16 threads. Due to the order of magnitude, Subutai-HW and NoC latencies are not visible. 131
Figure 6.6 – The execution time in seconds (10^9 ns) of our application set on SW-only and Subutai solutions for 32 threads. Due to the order of magnitude, Subutai-HW and NoC latencies are not visible. 132
Figure 6.7 – The execution time in seconds (10^9 ns) of our application set on SW-only and Subutai solutions for 64 cores. Due to the order of magnitude, Subutai-HW and NoC latencies are not visible. 133
Figure 6.8 – Relative time for every thread to reach the barrier `localS`; (a) is a plot for all barrier calls during the application execution, while (b) is a snippet of 50 barrier calls for better visualization. 137
Figure 6.9 – Relative time for every thread to reach all barriers calls of `workDone-Barrier`. 137
Figure 6.10 – Comparison of latency for the procedure `pthread_barrier_wait` for SW-only and Subutai. The results are for T_7 for a subset of barrier calls (400 up to 450). 138
Figure 6.11 – Comparison of latency for the procedure `pthread_barrier_wait` for SW-only and Subutai. The results are for T_7 for all barrier calls of `poolReadyBarrier`. 138
Figure 6.12 – Time spent, in 10^6 ns, on the mutex queue of Bodytrack and x264. (a) and (b) are plots of time spent (Y-axis) by the number of threads on the queue (X-axis) for x264 and Bodytrack, respectively; (c) and (d) are the total time spent for the same application set. 141
Figure 6.13 – Execution in seconds (10^9 ns) for multiple application sets (lower is better) (Exec = Execution). 143

Figure 6.14 – Execution in seconds (10^9 ns) for multiple application sets (lower is better) (Exec = Execution). 145
Figure 6.15 – Latency reduction in nanoseconds of the total execution time for three PARSEC applications. 148
Figure 7.1 – Polynomial Regression and recorded data for the barrier latencies for a worker thread of Bodytrack. 157
Figure 7.2 – Barrier latencies from Bodytrack for (a) recorded data, (b) one gaussian and (c) double gaussian models. 158
Figure 7.3 – A zoom for some of the first 10 barriers of Bodytrack from Figure 7.2. 159

List of Tables

Table 1.1 – Breakdown of RCU usage by kernel subsystems [MBWW17]. 21
Table 2.1 – Overview of some memory barriers [Int17] [Lea19] [SPA93]. 32
Table 2.2 – Memory barriers used by some PThread implementations for mutexes. Adapted from [Boe07] 33
Table 2.3 – An execution scenario from Listing 2.11 [Sta19]. 43
Table 3.1 – Traits and behavior of mutexes. Based on [Rei07] [Int19d]. 48
Table 3.2 – Memory barriers used by some PThreads implementations for spinlocks. Adapted from [Boe07]. 51
Table 3.3 – Gains for the release phase on TSAR and Alpha architectures [FPMR18]. 54
Table 3.4 – Unsuccessful firing rules attempts and its reasons [MRSD16]. 66
Table 3.5 – Notification memory gain for decoding 10 frames of five video sequences [MRSD16]. 68
Table 3.6 – Related work summary. 73
Table 3.7 – Related work key contribution. 74
Table 4.1 – Latency of essential queue procedures. m = memory latency. 84
Table 4.2 – Latencies of Subutai-HW states. c = cycle latency, m = memory latency, n = number of synchronization variables handled by Subutai-HW, ρ = number of threads on a barrier. 85
Table 4.3 – Latency of Subutai-HW states with parameters $c = 1ns$, $m = 2ns$, $n = 4$, $\rho = 63$, $FSMentry = 4ns$, and $FSMexit = 1ns$. 86
Table 4.4 – Cache space reduction of synchronization primitives. 92
Table 5.1 – Impact of *CSALimit* on the Bodytrack application set employing a timeslot of 1ms. CS = Critical Section. 104
Table 5.2 – Complexity of neocondition states. c = cycle latency, m = memory latency, n = number of synchronization variables handled by Subutai-HW, ϱ = number of threads on a neocondition. 116
Table 5.3 – Positive and negative attributes of neocondition compared to PThreads conditions. 117
Table 6.1 – Simulation time for some NoC simulators executing 1 second of simulated time without injecting packets. 121
Table 6.2 – Qualitative summary of key characteristics of PARSEC benchmarks [BKSL08]. 123
Table 6.3 – Breakdown of finer details of the benchmark applications for input size `simlarge` on an 8-core system [BKSL08]. 124

Table 6.4 – Number of events of synchronization primitives during the execution of PARSEC applications. 125

Table 6.5 – Number of synchronization primitives for PARSEC (`simmedium` input). *n* = number of threads. 125

Table 6.6 – Synthesis results for Subutai-HW and SPM using 28 nm SOI. 129

Table 6.7 – State-of-the-art area consumption. 130

Table 6.8 – Detailed execution time (ns) and the speedup for the application set executing on 16 cores. 134

Table 6.9 – Detailed execution time (ns) and the speedup for the application set executing on 32 cores. 135

Table 6.10 – Detailed execution time (ns) and the speedup for Bodytrack and Streamcluster executing on 64 cores. 136

Table 6.11 – Characteristics of the application set executing on CSA (CS = Critical Section; all data are averages achieved for all applications). 146

Table 6.12 – Unfairness metric for CSA and RR schedulers (lower is better). 147

Table 6.13 – Results for one producer and many consumer applications running with six threads. 151

Table 7.1 – The software exploration of neocondition: execution time of PThreads condition and SW-only neocondition. 161

List of Listings

2.1 Lock-free message processing [HMDZ19]. . . . 34
2.2 Generated code for lock-free message processing [HMDZ19]. . . . 34
2.3 Source code from an if statement [HMDZ19]. . . . 35
2.4 Compiled source code from Listing 2.3 [HMDZ19]. . . . 35
2.5 Queue structure. . . . 36
2.6 Lock-based enqueue operation. Synchronization procedures are colored red. 37
2.7 SPARC-RMO assembly code for locking and unlocking operations [Mar19]. . . 37
2.8 C-like code for the CAS procedure. . . . 38
2.9 Pseudo-code of lock-free enqueue operation (Synchronization procedures are colored red) [PKMS17] [MS96]. . . . 40
2.10 Lock-based dequeue operation. . . . 41
2.11 Lock-free dequeue operation susceptible to the ABA problem [Sta19]. . . . 42
3.1 Conceptual implementation of the `synchronize_rcu` primitive [McK19a]. . . . 49
3.2 Example of HTM-enabled parallel code. . . . 69
3.3 Lock-free CAS algorithm [PKMS17]. . . . 70
3.4 Lock-free MCAS algorithm [PKMS17]. . . . 70
3.5 Example of Dataflow application flow. Adapted from [MRSD16]. . . . 71
3.6 Intuitive, but incorrect, implementation for an HTM-enabled application [Kle19b]. 77
3.7 A correct implementation for an HTM-enabled library. Based on [Kle19b]. . . . 77
4.1 Definition of `pthread_mutex_t`. . . . 89
4.2 SPARC locking procedure. . . . 90
4.3 Kernel space futex queue (`kernel/futex.c`). . . . 91
4.4 The `pthread_mutex_t` employed by the Subutai Solution. . . . 92
4.5 Simplified Subutai driver implementation. . . . 93
5.1 Example of two conditions and their associated predicates for a FIFO. Initialization and error-checking are not shown for simplification purposes. . . . 105
5.2 Multiple awakenings by condition signal. Numbered comments refer to the order of events. [IEE16]. . . . 108
5.3 Bodytrack's condition variable usage. Adapted from C++ to C. . . . 110
5.4 Streamcluster's condition variable usage. Adapted from C++ to C. . . . 111
5.5 Ferret's condition variable usage. . . . 112
5.6 Common API among PThreads condition and neocondition. . . . 114
6.1 Shared structure for the producer-consumer application. . . . 149
6.2 A producer-consumer solution based on synchronization provided by PThreads. Synchronization procedures are colored red (based on [ADAD15]). . . . 150
A.1 Queue-related macros for the SPM. . . . 177
A.2 Pointer-related macros for Subutai-HW. . . . 178

A.3 Queue procedures pseudo-code implementation. 179
A.4 Subutai-HW state machine pseudo-code implementation. 186

List of Acronyms

API – Application Programming Interface

BKL – Big Kernel Lock

BTM – Big TTY Mutex

CAS – Compare-And-Swap

CBarrier – Central Barrier

CMP – Chip MultiProcessor

CSA – Critical-Section Aware

DAC – Design Automation Conference

FAI – Fetch-And-Increment

FIFO – First-In-First-Out

FOSS – Free/Open Source Software

FPU – Floating Point Unit

FSM – Finite State Machine

Futex – Fast Userspace muTEX

GBarrier – Gline-based Barrier

GCC – GNU Compiler Collection

GNU – GNU's Not Unix!

HPC – High-Performance Computing

HTM – Hardware Transactional Memory

I/O – Input/Output

ICC – Intel C++ Compiler

ID – IDentification

IRQ – Interrupt Request

ISA – Instruction Set Architecture

LoC – Lines of Code

MPSoC – MultiProcessor System-on-a-Chip

mutex – MUTual EXclusion

NI – Network Interface

NoC – Network-on-Chip

NUMA – Non-Uniform Memory Access

OAT – One Application at a Time

OpenMP – Open MultiProcessing

OS – Operating System

PARSEC – Princeton Application Repository for Shared-Memory Computers

PThreads – POSIX Threads

RCU – Read-Copy-Update

RR – Round-Robin

RTL – Register Transfer Level

rwlock – Reader-Writer Lock

SGL – Single Global Lock

SMP – Symmetric MultiProcessing

SoC – System-on-Chip

SOI – Silicon on Isolator

SPM – ScratchPad Memory

Subutai-HW – Subutai-Hardware

TBarrier – Tree-based Barrier

TBB – Threading Building Blocks

tl2 – Software Transactional Memory

TSX – Transactional Synchronization eXtensions

Contents

1 INTRODUCTION **17**
1.1 CONTEXT 17
1.2 MOTIVATION 18
1.3 PROBLEM STATEMENT AND THESIS CONTRIBUTIONS 24
1.4 DOCUMENT STRUCTURE 27

2 DATA SYNCHRONIZATION IN PARALLEL APPLICATIONS **29**
2.1 SYNCHRONIZATION IN UNIPROCESSOR SYSTEMS 29
2.2 SYNCHRONIZATION IN MULTIPROCESSOR SYSTEMS 30
2.2.1 MEMORY AND COMPILER BARRIERS 30
2.2.2 LOCK-BASED APPLICATIONS 35
2.2.3 LOCK-FREE APPLICATIONS 39

3 RELATED WORK **45**
3.1 SOFTWARE-ORIENTED SOLUTIONS 45
3.1.1 POSIX THREADS (PTHREADS) 45
3.1.2 OPEN MULTIPROCESSING (OPENMP) 46
3.1.3 THREADING BUILDING BLOCKS (TBB) 47
3.1.4 READ-COPY-UPDATE (RCU) 48
3.1.5 REORDERING CONSTRAINTS FOR PTHREAD-STYLE LOCKS 50
3.1.6 OPTIMIZATION OF THE GNU OPENMP SYNCHRONIZATION BARRIER IN MPSOC 52
3.2 HARDWARE-ORIENTED SOLUTIONS 55
3.2.1 HARDWARE TRANSACTIONAL MEMORY (HTM) 55
3.2.2 A HARDWARE IMPLEMENTATION OF THE MCAS SYNCHRONIZATION PRIMITIVE 57
3.2.3 CASPAR: BREAKING SERIALIZATION IN LOCK-FREE MULTICORE SYNCHRONIZATION 59
3.2.4 DESIGN OF A COLLECTIVE COMMUNICATION INFRASTRUCTURE FOR BARRIER SYNCHRONIZATION IN CLUSTER-BASED NANOSCALE MPSOCS 61
3.2.5 NOTIFYING MEMORIES: A CASE-STUDY ON DATA-FLOW APPLICATION WITH NOC INTERFACES IMPLEMENTATION 66
3.3 DISCUSSION 72

3.3.1 THE CHOICE OF PTHREADS 75
3.3.2 SUBUTAI COMPATIBILITY WITH OTHER LEGACY-CODE COMPATIBLE SOLUTIONS 75

4 SUBUTAI SOLUTION 79
4.1 TARGET ARCHITECTURE 80
4.2 SUBUTAI HARDWARE (SUBUTAI-HW) 82
4.2.1 SUBUTAI-HW RTL IMPLEMENTATION AND VERIFICATION 87
4.3 SUBUTAI SOFTWARE 88
4.3.1 USER SPACE PTHREADS LIBRARY 88
4.3.2 KERNEL SPACE FUTEX 90
4.3.3 SUBUTAI IMPLEMENTATION 91

5 SUBUTAI EXTENSIONS 97
5.1 CRITICAL-SECTION AWARE (CSA) SCHEDULING POLICY 98
5.1.1 MOTIVATION 98
5.1.2 BASELINE SCHEDULER DESIGN 100
5.1.3 APPLICATION EXAMPLE 100
5.1.4 DESIGN AND IMPLEMENTATION CHOICES 102
5.2 NEOCONDITION 104
5.2.1 MOTIVATION 104
5.2.2 CONDITION USAGE EXAMPLES FROM THE PARSEC BENCHMARK 109
5.2.3 DESIGN AND IMPLEMENTATION CHOICES 113
5.2.4 THE POSITIVE AND NEGATIVE ATTRIBUTES OF NEOCONDITION 116

6 EXPERIMENTAL RESULTS 119
6.1 EXPERIMENTAL SETUP 119
6.2 PARSEC – BENCHMARK SUITE FOR MULTIPROCESSING 122
6.2.1 BODYTRACK COMMUNICATION MODEL 126
6.2.2 STREAMCLUSTER COMMUNICATION MODEL 127
6.2.3 FACESIM COMMUNICATION MODEL 128
6.2.4 X264 COMMUNICATION MODEL 128
6.3 EXPERIMENTAL RESULTS 128
6.3.1 SUBUTAI'S AREA CONSUMPTION AND STATE-OF-THE-ART COMPARISON 129
6.3.2 PARSEC EXPERIMENTAL RESULTS 130
6.3.3 MICRO-BENCHMARK 148

7 CONCLUSIONS **153**
7.1 CONTRIBUTIONS OF THIS WORK 154
7.1.1 OTHER CONTRIBUTIONS 155
7.2 DISCUSSION AND FUTURE WORK 155
7.2.1 PSY: SYNTHETIC DATA SYNCHRONIZATION COMMUNICATION CREATOR . 156
7.2.2 ENERGY-AWARE DESIGN EXPLORATION FOR SUBUTAI-HW 159
7.2.3 BARRIER-AWARE POLICY FOR SCHEDULERS INTENDED FOR PARALLEL APPLICATIONS 159
7.2.4 SW-ONLY NEOCONDITION IMPLEMENTATION 160
7.2.5 QUEUE OPTIMIZER: SCHEDULER-AWARE HARDWARE QUEUE 161

REFERENCES **163**

APPENDIX A – Subutai-HW Pseudo-Code Implementation **177**
A.1 QUEUE SIZES 177
A.2 MACROS 178
A.3 QUEUE PROCEDURES 179
A.4 SUBUTAI-HW STATE MACHINE 186

APPENDIX B – List of Published Articles **193**

1. INTRODUCTION

> There are 3 rules to follow when parallelizing large codes. Unfortunately, no one knows what these rules are.
>
> W. Somerset Maugham, Gary Montry

Parallel applications are essential for efficiently using the computational power of any system comprised of multiple processing cores. Unfortunately, these applications do not scale effortlessly with the number of cores for various architectural and functional constraints. A key factor for limiting the scalability of parallel applications is the use of synchronization operations that take away valuable computational time and restrict the parallelization gains. Moreover, many of the synchronization operations are implemented to be sequentially executed, further diminishing the parallelization potential. Multiple solutions in research and industry have been proposed to tackle the synchronization bottleneck. They range from software-based, hardware-based, and mixed solutions. However, these solutions restrict the implementation to a subset of synchronization primitives, require refactoring the source code of applications, or both. Hence, a solution that eliminates both limitations is needed. This Thesis provides a step toward the realization of a solution capable of executing and accelerating any legacy parallel application without refactoring its source code. We chose, for this work, parallel applications that use the POSIX Threads (PThreads) as the basis for synchronization operations. Besides, novel applications can also be developed to employ our solution. This chapter exposes the motivation, goals, and contributions of this Thesis.

1.1 Context

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 of cores due to the stagnation of processing frequency [DKM+12]. Recently, this paradigm has evolved to incorporate hundreds or even thousands of simple cores to continue to deliver higher performance. For instance, Ephiphany-V is a 1024-processor System-on-Chip (SoC) designed in 16nm transistor technology [Olo16]. Intel also provides manycore architecture such as Xeon Phi (formerly known as Knight's Landing) with up to 72 cores [Int19c].

Unfortunately, the multiplication of cores by itself does not translate directly to the increase of performance as the applications must be parallel-compatible to exploit the chip parallelism paradigm. Therefore, parallel programming techniques had to be adopted. The shift from sequential to parallel execution demands a detailed understanding of algorithms, architecture designs, and synchronization libraries. Where once a single sequential thread

could do the execution, now the developer must divide the workload into multiple execution threads and synchronize the data and threads themselves. Also, the parallel execution has to deal with deadlock, livelock, race, and non-deterministic events [McK19a]. Decisions regarding both partitioning and synchronization are crucial to determine the achievable performance of the application on a multicore design because even small sequential portions of execution can have a significant performance impact. This is known as Amdahl's law [Gus11] shown in Figure 1.1. Because of such impact, parallelization is mainly done manually, as to allow fine-grained performance optimizations.

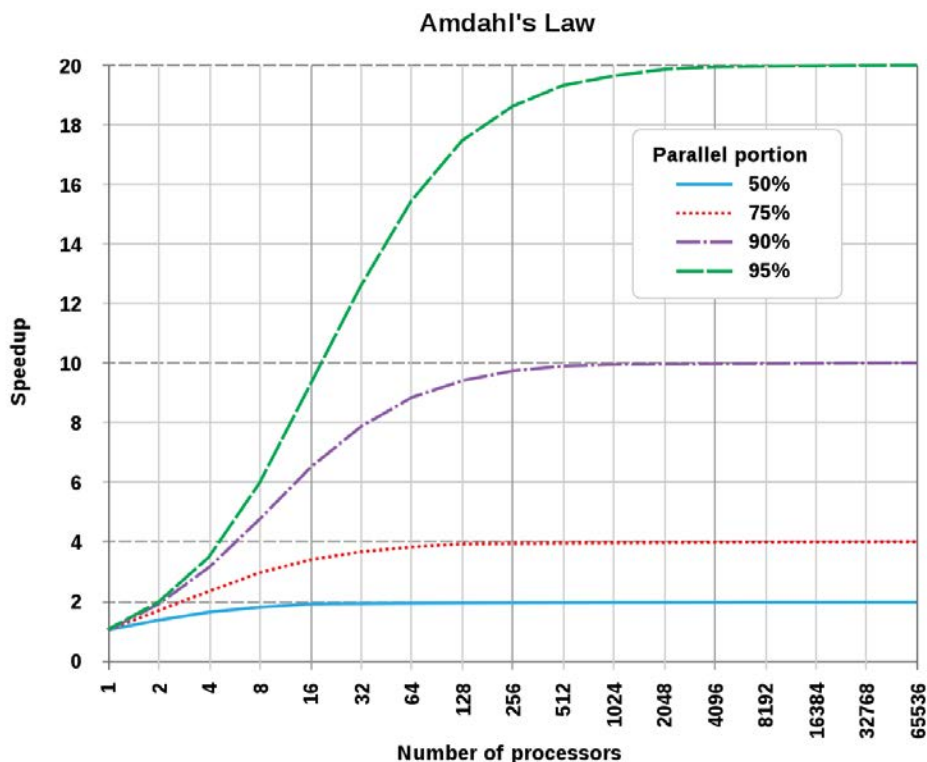


Figure 1.1 – The speedup of a parallel application is severely limited by how much of the application can be parallelized [Wik19].

1.2 Motivation

A developer has multiple alternatives to design a parallel application from a legacy sequential or parallel codebase. According to the constraints of a given project, it can use software-based, hardware-based, or a mixed solution to provide the parallel software primitives. Software-based solutions require the implementation of libraries and kernel support for some of the operations through system calls. PThreads and Open MultiProcessing (OpenMP) are examples of available and widely employed software-based solutions. Hardware-based solutions, however, require specific hardware to offload operations that are generally done in

software. Compare-and-swap (CAS), for example, is an instruction capable of comparing and swapping a value atomically. Such a solution is limited to the available instruction set of the processor and hardware modules. A mixed solution can be employed when the hardware does not handle some application scenarios. For instance, CAS instructions are limited to a single memory position; therefore, if the developer has to change multiple memory positions, this has to be handled in software.

An essential concept for synchronization is mutual exclusion, also called locking. For deterministic data structures, some operations have to be limited to a single instance at a given time [AGH+11], which can be achieved by a mutual exclusion mechanism. Figure 1.2 shows three essential types of locking schemes to deal with parallelism: giant, coarse-grained and fine-grained lock. The giant lock serializes the access to the entire code to a single user. The coarse-grained approach allows different processes to access independent parts of the system. Finally, the fine-grained approach allows different processes to access the same part of the system. Naturally, the complexity of the approach is proportional to its ability to execute in parallel.

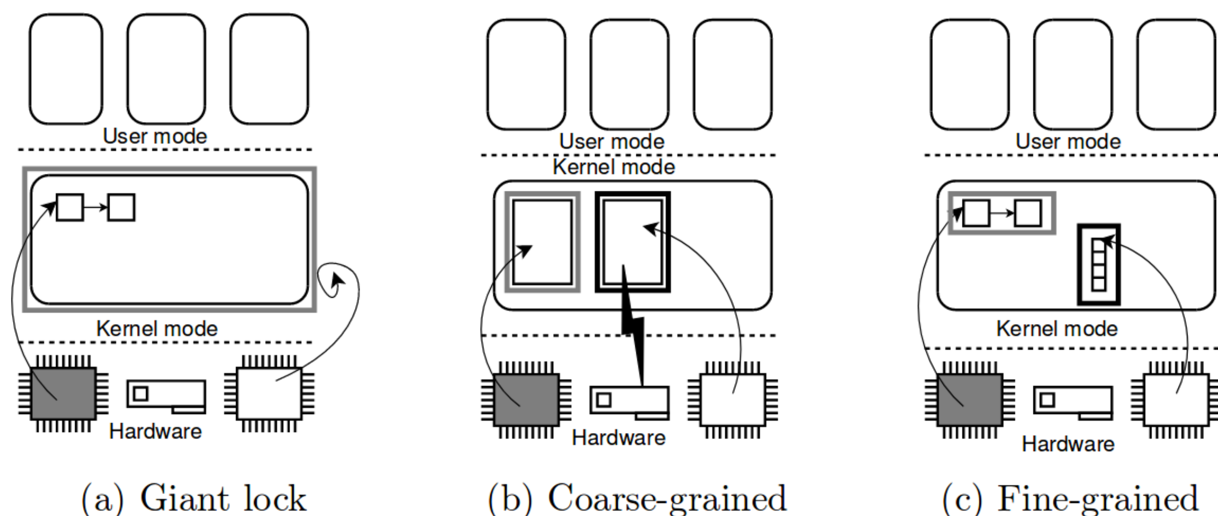


Figure 1.2 – The essential approaches to parallelizing code. Thick lines depict locks and the flash symbol denote device interrupt [Kå05].

Unfortunately, the limitation to all of these solutions is that the developer must refactor the source code to be able to use them. The redesign also applies to the already parallel-compatible codes, as the procedure interfaces of different solutions are not the same. Additionally, the refactoring of source code has some limitations described as follows.

- **Software redevelopment cost** - The software cost is difficult to estimate [Nas06] as an industry-grade application has many additional costs besides Lines of Code (LoC). They include but are not limited to the use of managers, technical documentation, tests, administrative activities (meetings, milestones), and support for post-release. The COCOMO model proposed by Boehm [Boe81] can estimate models with multiple

parameters to estimate the software cost. Using such model, COCOMO can calculate the cost of redesigning a complex software such as the Linux kernel. We use the Linux kernel as an example for two reasons: (i) as stated before, parallel libraries can demand new features to the kernel to execute properly; and (ii) the kernel has always been open-sourced, a crucial feature to understand the impact of parallel source code refactoring. The discussion focuses on two openly available aspects: LoC and estimation of time spent.

First, redesigning the entire Linux kernel in 2004 was estimated to cost 612 million dollars [Whe19]. In 2011, the cost jumped to 3 billion dollars [sz19]. Additionally, there are two examples of adding new concurrency features to the kernel: the addition/removal of the Big Kernel Lock (BKL), and the addition of Read-Copy-Update (RCU). None of them had to redesign the entire kernel, but they did refactor multiple, up to all, subsystems of the kernel code.

The BKL was the first attempt of the Linux kernel to support Symmetric MultiProcessing (SMP) that allowed only one process to enter in the kernel space at a given time (i.e., BKL was a giant lock). This restriction required that every entry and exit kernel calls had to be refactored to acquire/release the BKL. Initially, there were only nine calls to the BKL code; yet, as the kernel code grew, it reached 761 calls [LH02] and over 200 source code files [Ber19b]. The increase of BKL calls is due to the effort of adding kernel concurrency. The giant lock was being shifted to a coarse-grained lock that allowed multiple processes in the kernel space when possible. Finally, the shift from a giant lock to a coarse-grained and then to a fine-grained lock system was the work result of countless developers and over ten years of refactoring [Ber19a].

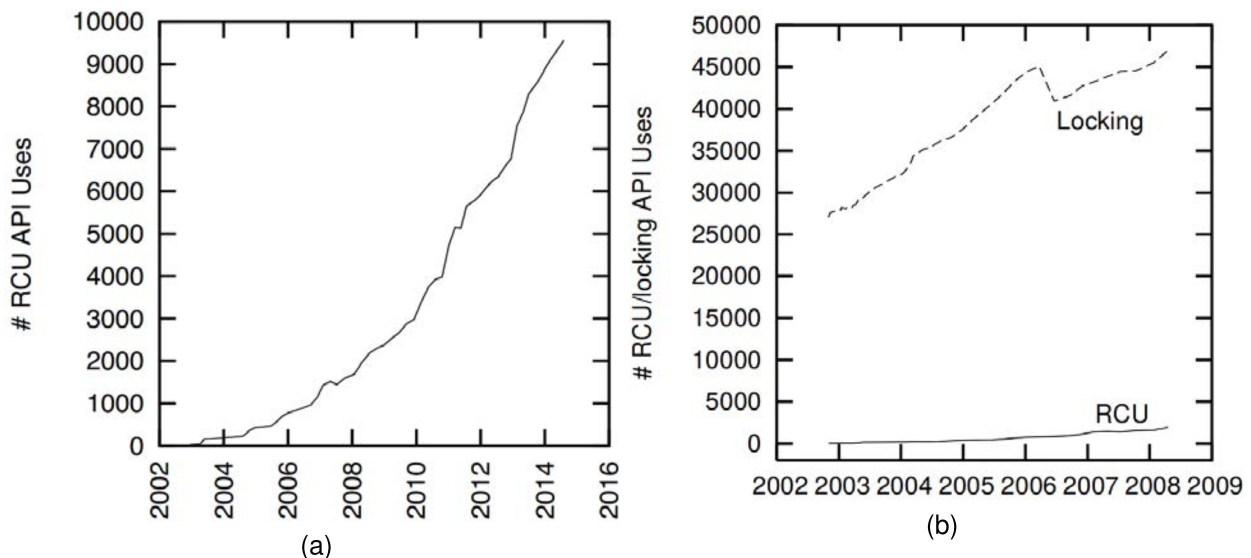


Figure 1.3 – (a) RCU usage over the years [MBWW17]; (b) RCU and Locking usage over the years [MW08].

RCU is a specialized synchronization technique that can replace reader-writer locking. It defers operations to achieve a better read-side performance, which comes at the cost of having to deal with possible stale data. RCU was introduced to the Linux kernel in 2002 [MW08]. Figure 1.3a shows that RCU has continuously been employed for more than ten years – over 9000 calls in 2015. It did not replace all cases of locking, as shown in Figure 1.3b. A critical factor in this is the complexity of understanding the RCU mechanism, which is discussed shortly. Figure 1.1 shows all subsystems of the Linux kernel that employs RCU for synchronization. RCU influences over 16 million LoC across 15 subsystems.

Table 1.1 – Breakdown of RCU usage by kernel subsystems [MBWW17].

Subsystem	Uses	LoC	Uses / KLoC
ipc	92	9,094	10.12
virt	82	10,037	8.17
net	4519	839,441	5.38
security	289	73,134	3.95
kernel	885	224,471	3.94
block	76	37,118	2.05
mm	204	103,612	1.97
lib	75	94,008	0.80
fs	792	1,131,589	0.70
init	2	3,616	0.55
include	331	642,722	0.51
drivers	1949	10,375,284	0.19
crypto	12	74,794	0.16
arch	249	2,494,395	0.10
tools	2	144,181	0.01
Total	9,559	16,257,496	0.59

We presented two synchronization mechanisms introduced into the Linux Kernel: BKL and RCU. Both of them share essential aspects to the software development cost: many years of development to assimilate with existing code and increased difficulty in refactoring code. For instance, the substitution of RCU in 2014 with an alternative synchronization mechanism would affect 15 kernel subsystems and over 16 million lines of kernel code (Table 1.1).

- **Challenge of parallel code refactoring** - Source code modification is always susceptible to introducing additional flaws in the software (i.e., software bug). McConnell estimates that software bugs range from 0 to 100 per thousand LoC as a consequence of the development methods employed [McC04]. Refactoring parallel code is even more susceptible than sequential code, as it is common for developers to be befuddled

with the use of synchronization techniques. The same synchronization mechanisms discussed previously can illustrate such a challenge.

The kernel concurrency became paramount to fulfill the multicore machine requirements. Therefore, the kernel changed, over time, its locking technique from a giant lock to a fine-grained approach. First, the transformation allowed the execution of several processes in independent subsystems, and, then, the execution of several processes into the same subsystem. The refactoring took over ten years and is described by Lindsay as [LH02]:

> *"Sometimes it's not clear that even the authors understood why it [BKL] was needed; they appear to have invoked it either because the code they were copying from invoked it, or simply because they feared angering the ancient gods of coding by omitting it."*

Igno Molnár, one of the current maintainers of the scheduler and locking subsystems, said the following about removing the BKL [Mol19]:

> *"This task is not easy at all. 12 years after Linux has been converted to an SMP OS we still have 1300+ legacy BKL using sites. There are 400+ `lock_kernel()` critical sections and 800+ ioctls. They are spread out across rather difficult areas of often legacy code that few people understand and few people dare to touch."*

Lindsay also created a series of documents to detail every usage of the BKL in different kernel releases [Ber19b]. This document describes that multiple instances of the BKL use were: (i) confusing, and (ii) contradicted comments left by the original developers. The following is an example of BKL usage on the TTY subsystem:

> *"Held during `do_tty_hangup()` – code suggests it is protecting a data structure I can't find. A comment here screams "FIXME! What are the locking issues here?" which suggests the reasons for grabbing this lock may not be well understood."*

Now that the kernel uses fine-grained locks it is even harder to refactor it again. Figure 1.4 shows the use of a coarse-grained lock specifically for the TTY subsystem called Big TTY Mutex (BTM). Developers for the TTY subsystem now must understand and respect these rules for locking. The challenge is that there are procedures that: (i) do not deal with BTM; (ii) acquire BTM; (iii) release BTM; and (iv) acquire and release BTM. However, there is no easy indicator of the specific case for each procedure besides documentation and source code commentary, both susceptible to be out-of-date. The

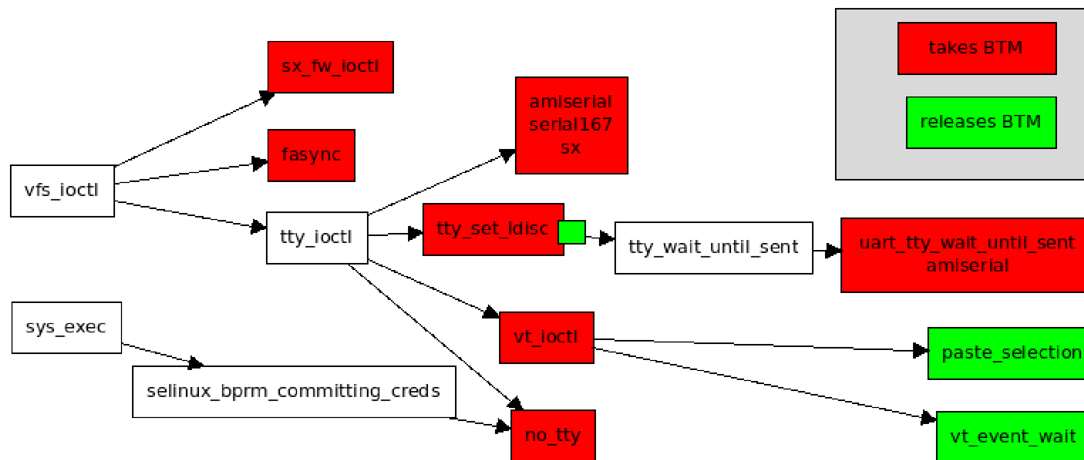


Figure 1.4 – Snippet of the locking scheme for the TTY subsystem [BC19a] [BC19b].

complexity is even higher for fine-grained locks because it requires dealing with multiple locks instead of a single one.

On the fine-grained approach, RCU is intended to be used for code that requires fast read-side performance. Different from reader-writer locks, the RCU readers never spin nor block. Figure 1.5 shows the effects of using either of these synchronization techniques. While RCU shows impressive results, it demands a thorough understanding of computer architecture design. Similar to other fine-grained approaches, RCU presents a trade-off: it offers performance gains with increasing code and maintainability complexity. McKenney and Walpole, leading developers of RCU, stated the following about the RCU gains and its design complexity [MW08] [McK19a]:

> *"This leads to the question "what exactly is RCU?", and, not infrequently, "how could RCU **possibly** work??", to say nothing of the assertion that RCU cannot possibly work."*

> *". . . RCU readers might access stale data, and might even see inconsistencies, either of which can render conversion from reader-writer locking to RCU non-trivial."*

McKenney and Walpole also discuss their experience working with the Linux community to bring RCU to the kernel as RCU being dramatically changed by Linux than by Linux being changed by RCU. Free/Open Source Software (FOSS) developers understand that changing source code results in non-trivial refactoring of code; hence they require assurances. From the Linux community, McKenney and Walpole noted that they had to refactor the RCU code even before being accepted into the kernel. They also propose that source code be considered a liability instead of an asset, due to the complexity of servicing, supporting, and maintaining it [MW08].

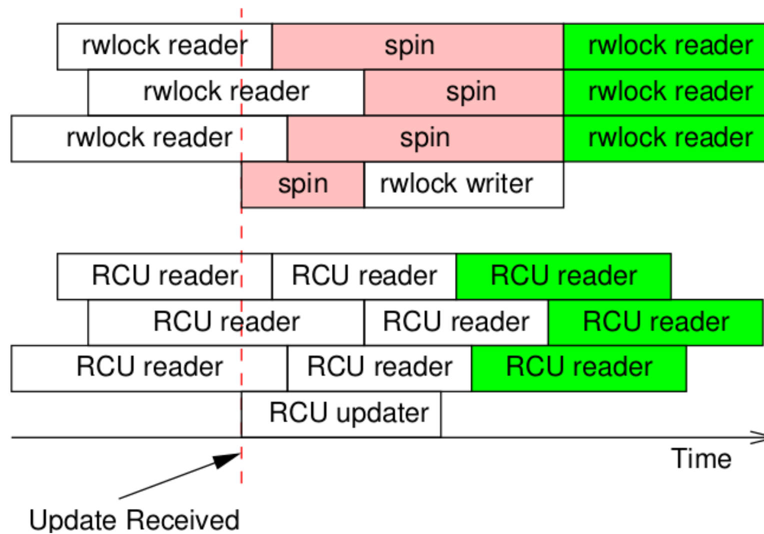


Figure 1.5 – Execution of reader-writer lock and RCU. The green color is used to represent up to date data [McK19a].

- **Lost legacy source code** - The essential requirement to refactor a legacy parallel application is the availability of the application source code. However, it is common for the legacy source code to be lost [Cur19] [McA19] [Wal19]. Even when the source code is available, it may be out-of-date [Mat19]. Hence, developers may prefer to rewrite the entire code than to rely on existing code by simulating the legacy application behavior [Fle19]. Redeveloping existing code, for either of the reasons listed earlier, goes against the expected practice of reusing software, as it increases the total software cost.

1.3 Problem Statement and Thesis Contributions

As previously explained, architectures powered by multiple cores require parallel applications to exploit its potential, and software development is costly regarding return on investment. Figure 1.6 shows that software development has been the dominant cost of developing new products for multiple generations of transistor technology, and it is getting worse.

Using the Linux kernel, as an illustration, we saw that refactoring a parallel code is also costly, demanding years of work. BKL was easily manageable in the beginning, with a dozen of calls, but it became strenuous when developers wanted to substitute it for a more refined approach, and its code has been widely distributed on the kernel, with over thousands of calls. For the adoption of the RCU, developers wanted assurances to make the shift for a new synchronization technique, as it required an understanding of a new paradigm for synchronizing existing code.

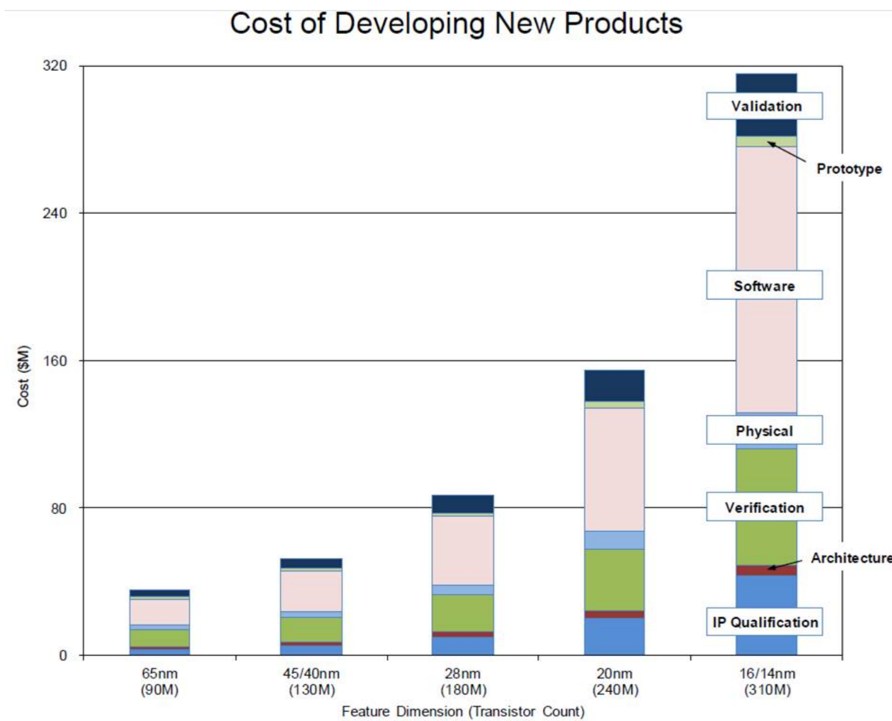


Figure 1.6 – Cost of each production step of multiple generations of transistor technology [Spe19].

Given the exposed motivation and problem description, this Thesis aims to provide faster application execution time without source code modification. We employ an HW/SW co-design for implementing our solution. The following set of specific objectives were defined to accomplish our goals:

1. The definition of a solution, namely Subutai, to provide fast synchronization for legacy and novel parallel applications. The solution is demonstrated by performance benefits on analytical models, informal micro-benchmarks, and real applications achieved with no increase in application complexity;

2. The presentation and development of two software components to interact with our hardware – an Operating System (OS) driver for HW/SW communication and a user space library that provides the PThreads Application Programming Interface (API);

3. The presentation of an analytical and Register Transistor Level (RTL) implementation of the hardware component, namely Subutai Hardware (Subutai-HW). It further allows for future development; and

4. The definition of two Subutai extensions – optional features for accelerating parallel applications in particular scenarios. Firstly, we propose a scheduler policy, called Critical-Section Aware (CSA), for accelerating parallel applications in a highly-contended scheduling scenario, while maintaining the fairness of scheduler timeslot distribution. Secondly, we define the ‘neocondition’ synchronization variable – a variable that behaves

as the condition variable from PThreads while removing the serialization of access to it (i.e., no mutex is required).

Figure 1.7 depicts the Subutai solution with a general-purpose computing stack, highlighting the components required (in red) and optional (in blue) for its operation. Subutai is comprised of: (i) a hardware module specialized in accelerating the essential synchronization operations (Subutai-HW); (ii) an OS driver for hardware/software communication (Subutai Driver); and (ii) a custom user space library, with the same function signature as PThreads, for parallel programming to use our solution without modifying the application source code. These components are discussed in-depth in the next chapters.

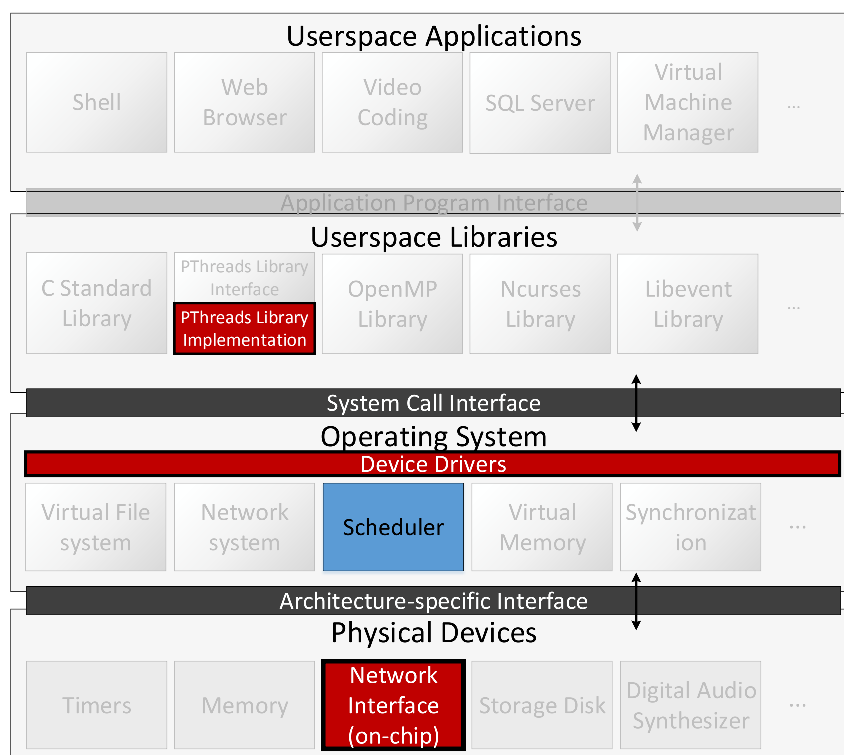


Figure 1.7 – Subutai components are highlighted in red in the computing stack. Subutai only requires changes in the (i) on-chip NI, (ii) OS NI driver and (iii) PThreads implementation. Additionally, a new scheduling policy (in blue) is explored in this work as an optional optimization.

Figure 1.8 shows the mechanisms employed in Subutai for accelerating synchronization operations of PThreads. We target all the data synchronization operations supported by PThreads, namely: mutex, condition, and barriers. We accelerate them by making use of our hardware module while maintaining the same functionality as provided by the software solution (i.e., libpthread). We currently only support the standard variants of these primitives; in other words, the attribute parameter `attr` must be nil. Generally, that is the case, as, for instance, the applications provided by PARSEC.

Additionally, we provide a new synchronization primitive called neocondition, which is a derived primitive from the condition definition of PThreads. Its key difference is the absence of the use of mutexes; hence, no serialization is required to access it. Finally, the CSA policy was designed to accelerate critical sections of parallel applications for highly-contended scheduler scenarios without reducing the performance of other applications (i.e., a fair scheduler). This policy directly accelerates the mutex primitive since it creates the critical sections of an application. Besides, conditions also indirectly profited from the policy as it employs mutexes as well.

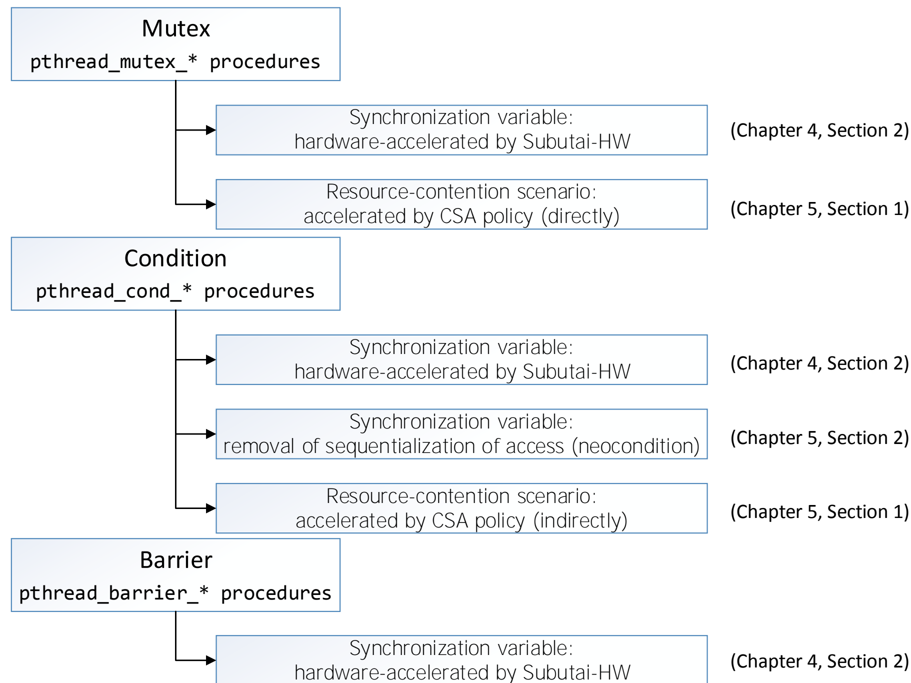


Figure 1.8 – Synchronization acceleration with the Subutai solution for different scenarios.

1.4 Document Structure

The remainder of this Thesis is organized into six chapters. Chapter 2 discusses synchronization operations on uni- and multiprocessor designs; this chapter also defines a basic terminology that is used throughout the document. Chapter 3 discusses the state-of-the-art related work on data synchronization. Chapter 4 discusses the design of hardware and software components required by Subutai; additionally, it presents the target architecture intended for using Subutai. Chapter 5 explores two optional extensions for Subutai: (i) a scheduling policy called CSA; and (ii) a new synchronization variable called neocondition. Chapter 6 presents the experimental results on area consumption, real parallel applications, and micro-benchmark. Finally, Chapter 7 presents the final considerations of this work and directions for future work.

2. DATA SYNCHRONIZATION IN PARALLEL APPLICATIONS

> Take heed. You got it wrong. Admit it. Locking is _hard_.
> SMP memory ordering is HARD.
> So leave locking to the pro's. They _also_ got it wrong, but they got it wrong several years ago, and fixed up (...)
> This is why you use generic locking. ALWAYS.
>
> Linus Torvalds

A program can be comprised of many computational units. These units range from threads, processes, coroutines, interrupt handlers, etcetera. When they work, the result of each computational unit might affect or be affected by those of the other computational units [McK04]. We use the term thread as a generic word to encompass these computational units.

A major concern in any parallel application is the access and update of application data. This problem is called synchronization, and many solutions have been studied and proposed over the past decades [McK04]. Solutions can be focused on proposing new software or hardware designs. Yet, all solutions need basic hardware operations to deal with atomicity.

This chapter reviews the design of data synchronization for parallel applications. Section 2.1 presents a brief discussion of synchronization for uniprocessor systems, while Section 2.2 discusses in-depth the challenging synchronization techniques for multiprocessor systems.

2.1 Synchronization in Uniprocessor Systems

The use of synchronization primitives in uniprocessor systems may seem superfluous at first glance, as only a single thread may be running at a given time. However, this is not the case even for a sequential – one thread – application since interrupting and preemptive scheduling events can affect such systems. Functions that are called by both the application and the interrupt/schedule event do not behave correctly unless a reentrant version of the function exists. Unfortunately, multiple functions provided by the standard C library, such as `malloc` and `fprintf` [GCC19b], are not reentrant since they use static data. Thus, other solutions are required to deal with such a situation. A common solution is to disable interrupts and preemptive scheduling at the cost of loss of system responsiveness. Another solution is to provide locking primitives to the user application, although it may also be necessary to disable interrupts for sensitive locking operations [McK04].

2.2 Synchronization in Multiprocessor Systems

Techniques that depend on disabling interrupts, like the ones described in the previous section, will fail on multiprocessor systems, as disabling interrupts affect only the local core [Moy13]. Rather than successfully suspending the execution of the interrupt code, the code will execute concurrently in another core. Therefore, it is not possible to rely solely on disabling interrupts; the ability to perform a set of operations without interruption is necessary. This is achieved by atomic operations.

An atomic operation is either entirely successful or entirely unsuccessful, and algorithms have to deal with both cases [Moy13]. Generally, unsuccessful cases retry the operation with either the same request or an updated one. Unfortunately, atomic operations are more expensive than simple instructions, and their cost increases as the number of threads access the same memory position. Figure 2.1 shows the scalability of a Fetch-And-Increment (FAI) operation on a simple counter variable: every thread reads the current value and increments it by one atomically. Figure 2.2 shows the throughput of different atomic operations, including FAI, for four architectures comprised of 48 (Opteron), 80 (Xeon), 8 (Niagara), and 36 (Tilera) cores [DGT13]. None of them has improved throughput for FAI operations after six threads; hence, the scalability is far from the ideal. A key factor in this phenomenon is the cache line bouncing [McK19a]. For every thread requesting to write, there will be multiple invalidation messages to the other caches through the interconnection architecture. In addition, these caches will need to fetch the line with the new value as well. In this case, the elimination of cache bouncing requires redesigning the application. A second factor influencing scalability is the necessity of memory barriers[1]. Developers that worked with lock-based algorithms may never have to deal with memory barriers directly; the same cannot be said for lock-free and operating system designers. The importance of memory barriers is explained in the following section.

2.2.1 Memory and Compiler Barriers

Architecture optimizations have been developed to make user applications run faster while running the same source code. A relaxed memory model is an optimization that affords opportunities to improve application performance [Mar19] for uniprocessoring as well as multiprocessoring systems. They include but are not limited to [HMDZ19]: reordering instructions, reordering memory operations, deferral of memory operations, speculative loads, and speculative stores. However, some optimizations may produce an improper result; for instance, the processor may change the order of memory read/write operations. Figure 2.3a

[1]The use, for instance, of GCC atomic builtin operations imply full memory barriers [GCC19a].

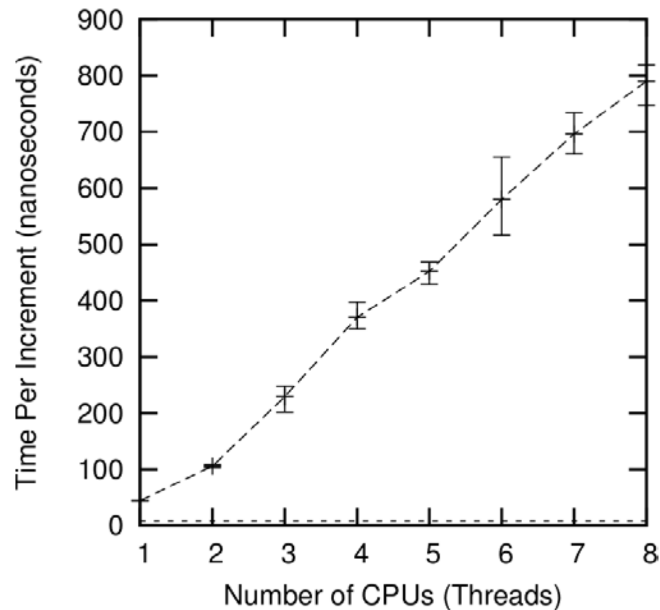


Figure 2.1 – Atomic increment scalability on a Nehalem Intel processor [McK19a].

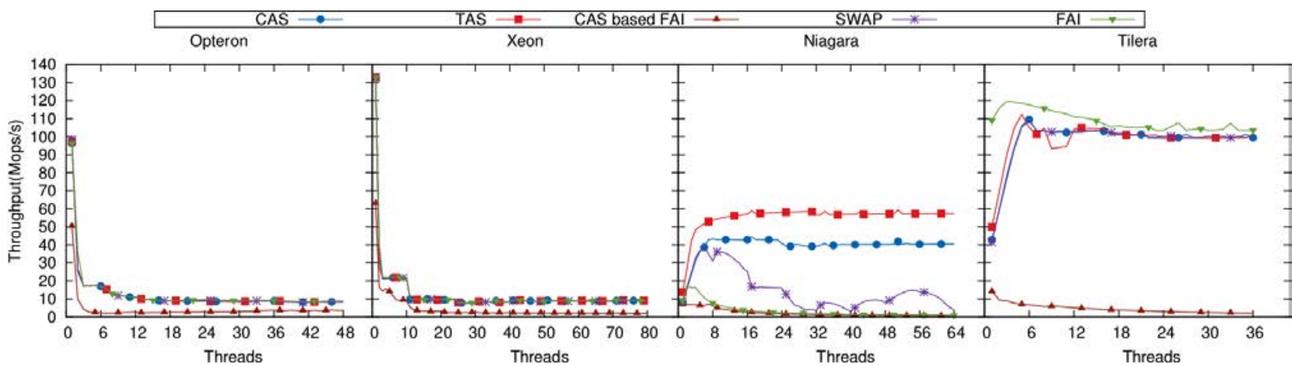


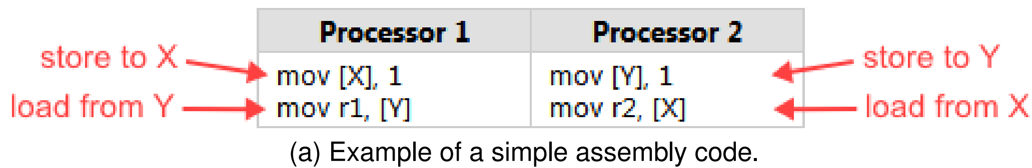
Figure 2.2 – Throughput of different atomic operations on a single memory position [DGT13].

depicts a reordering example provided by the software manual of Intel [Int17]. Suppose that both variables `X` and `Y` are initialized as `0`, and both processors[2] are running in parallel. It is natural to assume that the parallel execution produce `[r1 = r2 = 1]`. Nevertheless, according to the Intel manual [Int17], it is perfectly valid that the result can also be `[r1 = r2 = 0]`, `[r1 = 1, r2 = 0]`, or `[r1 = 0, r2 = 1]`. The reordering of operations, demonstrated in this example, is not restricted to Intel processors.

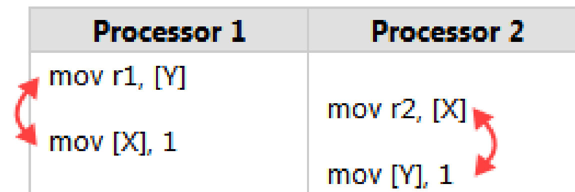
The assumption of `[r1 = r2 = 1]` may not hold because the processor can, unless stated otherwise, reorder a load with an earlier store to a different memory location, as they do not have an explicit dependency. Figure 2.3b shows one possibility of the application execution, where the assumption does not hold. This behavior breaks a simple but fundamental assumption that generated code will be executed in the order described by the source code.

Therefore, it is mandatory to inform the processor of the data/control dependency of instructions to avoid reordering, as shown in Figure 2.3b. This is done by using a memory

[2]Although some authors use the term ‘processor’ to designate the processing unit, instead of ‘core,’ we emphasize that they are equivalent for the purposes of this work.



(a) Example of a simple assembly code.



(b) One of many execution possibilities of Figure 2.3a.

Figure 2.3 – Reordering example from the Intel Manual [Int17] [Pre19b].

barrier. There are multiple types of memory barriers, and not all Instruction Set Architectures (ISAs) need all of them. Thus, the lack of a standard behavior makes creating portable code harder, as applications can execute as expected in one ISA, but not in another. Following, we detail some of the memory barriers currently employed by different ISAs.

Table 2.1 is organized in three columns. The first column is a mnemonic name for the barrier. The second column is the guarantee that the barrier provides the application. Note that we use a generic specification of the guarantee; the real guarantee may vary according to each ISA. Finally, the third column identifies which ISAs need to use the barrier to receive the guarantee. For ISAs not specified, they do not need to use any barrier as it is already guaranteed by the ISA specification. The SPARC specification allows three implementations for its ISA (RMO, PSO, and TSO).

Table 2.1 – Overview of some memory barriers [Int17] [Lea19] [SPA93].

Barrier	**Guarantees** (All (1) instructions executed prior to the barrier commit before the core execute any additional (2) instructions)	**Required by ISA** (x86, ARM, PowerPC, ALPHA, SPARC-{RMO, PSO, TSO})
`LoadLoad`	(1) Load; (2) Load	ARM, PowerPC, ALPHA, SPARC-RMO
`LoadStore`	(1) Load; (2) Store	ARM, PowerPC, ALPHA, SPARC-{RMO, PSO}
`StoreStore`	(1) Store; (2) Store	ARM, PowerPC, ALPHA, SPARC-{RMO, PSO}
`StoreLoad`	(1) Store; (2) Load	x86, ARM, PowerPC, ALPHA, SPARC-{RMO, PSO, TSO}

Table 2.2 establishes the use of memory barriers for the locking and unlocking operations on different ISAs for the GNU implementation of the standard C library. The acquire barrier is comprised of `LoadLoad` and `LoadStore` barriers. The release barrier is

comprised of `LoadStore` and `StoreStore` [HMDZ19]. The full barrier is the combination of all barriers presented in Table 2.1. However, it is important to state that the barriers of Table 2.1 are only applied when the architecture requires them. For instance, the x86 architecture applies only the `StoreLoad` barrier for the full memory barrier mentioned in Table 2.2.

Table 2.2 – Memory barriers used by some PThread implementations for mutexes. Adapted from [Boe07]

Environment	lock memory barrier	unlock memory barrier
glibc 2.4 Itanium	full	full
glibc 2.4 x86	full	full
glibc 2.4 ALPHA	acquire	release
glibc 2.4 PowerPC	acquire	release

The estimation of the general cost of memory barriers is challenging, as it: (i) affects ISAs in different ways; and (ii) inhibits speculative operations that exist or not depending on the application employed. Memory barriers must be employed only when strictly necessary to avoid performance degradation. Agner Fog found that the minimum latency for a full memory barrier is 23 clock cycles for an Intel Nehalem architecture [Fog19]. Boehm [Boe07] shows that the cost of a pair of `lock` and `unlock` operation calls can vary by roughly a factor of two depending on whether a memory barrier is needed in the `unlock` operation.

Unfortunately, reordering instructions and memory operations are not limited to the processing unit [Boe07]. The reorder can occur even earlier by optimizations done at the compilation time. Until C11 (C standard revision), concurrency was not built on the C language itself. Hence, the C compiler assumes, unless stated otherwise, that there is only one thread of execution in a given application [Cor19]. The effect of assuming a single-threaded application is shown in Listings 2.1, 2.2, 2.3, 2.4.

Listing 2.1 is a lock-free message processing application. The message is captured on process execution, and a flag called `ready` is set for this event. The message and flag are shown in lines 4 and 5. Periodically, an interrupt arrives at the application and, if the flag is set, the packet is processed. The corresponding code is shown in lines 11 and 12. Alas, there is nothing to prevent the compiler from switching lines 4 and 5 (as shown in Listing 2.2), as there is no indication of dependency. Now the flag is set before the message is fetched, resulting in an application execution with undefined behavior.

Listing 2.3 is a simple application composed of an if-then-else statement. The variable `var_b` may be equal to the value of 42 or variable `var_a`, if `var_a` is zero or not, respectively. The compiler can optimize this code by removing one of the branching scenarios; it speculates that the value of variable `var_a` is zero, so `var_b` is 42 and writes to `var_b` again otherwise. The generated code is shown in Listing 2.4. For a single-threaded application,

Listing 2.1 – Lock-free message processing [HMDZ19].

```
void
process_level(void)
{
        msg = get_message();
        ready = true;
}

void
interrupt_handler(void)
{
        if (ready)
                process_message(msg);
}
```

Listing 2.2 – Generated code for lock-free message processing [HMDZ19].

```
void
process_level(void)
{
        ready = true;
        msg = get_message();
}

void
interrupt_handler(void)
{
        if (ready)
                process_message(msg);
}
```

this represents a performance gain, and it maintains the expected application behavior. For multi-threaded applications, this generates a subtle spurious value of 42 that can be seen by other threads. Once again, this behavior breaks the assumption that compiled code will be executed in the order expected by the source code [HMDZ19].

Listings 2.1 and 2.3 demonstrate cases that demand compiler barriers to prevent the compiler from moving memory accesses from one side of the barrier to the other side. However, memory barriers already imply the use of compiler barriers [HMDZ19], so they are not explicitly required when a memory barrier is employed.

Listing 2.3 – Source code from an if statement [HMDZ19].

```
1 int var_a;
2 int var_b;
3 (...)
4 if (var_a)
5         var_b = a;
6 else
7         var_b = 42;
```

Listing 2.4 – Compiled source code from Listing 2.3 [HMDZ19].

```
1 int var_a;
2 int var_b;
3 (...)
4 var_b = 42;
5 if (var_a)
6         var_b = a;
```

Developers used to lock-based algorithms may never have employed directly any of the barrier types mentioned. Their use was avoided because the locking mechanism already provides them intrinsically [Boe07]. Because these barriers are lost in lock-free algorithms, by removing the calls for locking procedures, that they are harder to design and debug.

2.2.2 Lock-based Applications

Lock-based applications employ locking procedures provided by a user space library. The library offers multiple types of synchronization procedures that will be explained in Section 3.1. For now, we focus on the basic synchronization procedure called mutex. A mutex is a mutual exclusion primitive that allows one, and just one, thread to hold it. Even if multiple threads try to hold it at the same time, it is guaranteed that only one can hold it. The guarantee relies on employing atomic operations. The process of holding a mutex is also called by two other names: locking and owning it.

Listing 2.6 is a lock-based implementation of the enqueue operation on a linked-list queue. The linked-list queue is shown in Listing 2.5. The lock-based implementation performs the enqueue operation with five lines of code (6-10). Lines 6 and 10 restrict the critical section protected by the mutex called `queue->q_lock`. Therefore, Lines 7 to 9 can be understood as an atomic block of operations that are perceived by other threads. Consequently, the developer can choose the order of operations freely, as the rest of the system perceives them as a single event. In addition, the compiler and processor can also reorder these instructions

freely for the same reason. Besides, as only one writer is allowed at a time, the code is straightforward (only sequential statements), enhancing its maintainability.

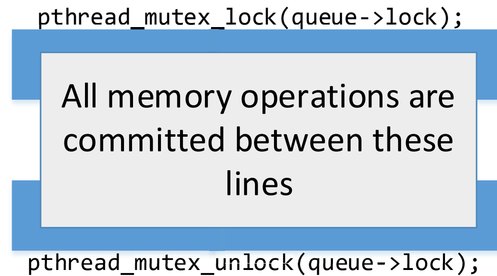


Figure 2.4 – The net effect of using lock-based procedures: implicit memory barriers (represented as blue fences) (based on [Pre19a]).

Listing 2.5 – Queue structure.

```
1  #include <pthread.h>
2
3  struct queue_item {
4          void                   *qi_ptr;
5          struct queue_item      *qi_next;
6  };
7
8  struct queue {
9          pthread_mutex_t        *q_lock;
10         struct queue_item      *q_head;
11         struct queue_item      *q_tail;
12 };
```

As mentioned in Subsection 2.2.1, the developer does not have to handle memory and compiler barriers explicitly because the synchronization library does it for him. Listing 2.7 shows the generated locking and unlocking procedures for the SPARC-RMO architecture. Moyer [Moy13] presents the equivalent of locking and unlocking for the PowerPC ISA. Locking is done in lines 2 to 7, and unlocking is done in lines 11 to 13. Line 2 uses the CAS procedure to execute an atomic operation. The CAS procedure receives three parameters in the following order: memory location, expected value, and new value[3]. The memory location is updated with the new value if, and only if, the previous value is the same as the expected value. Otherwise, no memory write is executed. The procedure returns the new or previous value if the memory write is or not succeeded, respectively [Moy13] [Mar19]. An example of a C-like code for the CAS procedure is shown in Listing 2.8. If the operation has succeeded,

[3]Some implementations choose to swap the order of the second and third parameters (for instance, Listing 2.9). Regardless, the operation is the same.

Listing 2.6 – Lock-based enqueue operation. Synchronization procedures are colored red.

```
1  #include <pthread.h>
2
3  void
4  enqueue(struct queue_item *qitem, struct queue *queue)
5  {
6          pthread_mutex_lock(queue->q_lock);
7          queue->q_tail->qi_next = qitem;
8          qitem->qi_next = NULL;
9          queue->q_tail = item;
10         pthread_mutex_unlock(queue->q_lock);
11 }
```

it means that this thread holds the lock. In this case, line 6 is executed to use two memory barriers: a `LoadLoad` and a `LoadStore` barrier. The other two barriers described in Table 2.1 are executed when the lock is granted back by calling the unlocking procedure. Line 11 executes the other two barriers. The net effect of using lock-based procedures is represented in Figure 2.4, where the blue fences represent the use of multiple memory barriers.

Listing 2.7 – SPARC-RMO assembly code for locking and unlocking operations [Mar19].

```
1  _check_lock:
2          cas     [%o0],%o1,%o2           ! try the CAS
3          cmp     %o1,%o2
4          mov     0,%o0                   ! assume it succeeded - return FALSE/0
5          movne   %icc,1,%o0              ! may have failed - return TRUE/1
6          membar  #LoadLoad|#LoadStore    ! memory barrier (RMO)
7          retl
8          nop
9
10 _clear_lock:
11         membar  #StoreStore|#LoadStore  ! memory barrier (RMO)
12         st      %o1,[%o0]               ! store the word
13         retl
14         nop
```

2.2.2.1 Scalability of Reader-Writer Locks

Before discussing lock-free algorithms, we must address the scalability of reader-writer locks (rwlock), which are specialized locks for read-mostly critical sections [McK19a];

Listing 2.8 – C-like code for the CAS procedure.

```
int
CAS(void *ptr, int expected_val, int new_val)
{
        int     r;
        ATOMIC {
                r = (int *)*ptr;
                if (r == expected_val) {
                        r = new_val;
                        (int *)*ptr = r;
                }
        }
        return (r);
}
```

rwlock provides greater scalability than an exclusive lock as it allows multiple threads to read the shared data concurrently if no writer is present. Only when a writer is present, the behavior of rwlock is reversed to a mutex. Hence, it seems the scalability issues of mutexes are solved as long as writes are performed few and far between. This intuition has been acknowledged by other software developers [Bou19] [McK19a].

In practice, however, the performance is far from ideal. Interestingly, a limiter of scalability is found on the reader side. Figure 2.5 shows the results of a scalability test of a reader-only rwlock application for a Power-5 system [McK19a]. The rwlock ideal scenario is a constant performance of a single thread acquiring the read rwlock. The application is described by McKenney in [McK19a].

Critical sections are built to be as short as possible because it serializes execution. Yet, as shown in Figure 2.5, the performance of rwlocks in smaller critical sections is drastically inferior compared to the ideal performance. For instance, a lock-based queue search critical section comprises a dozen lines which results in two orders of magnitude less than the worst case of Figure 2.5[4].

The scalability issue shown in Figure 2.5 goes back to the same issues of atomic operations shown in Figure 2.1. Every time a reader enters and exits the critical section, it must update a variable that counts the number of threads present in the critical section. That is how a writer knows if any readers are present. Yet, because multiple readers can try to update the variable at the same time, we need atomic operations which serializes the execution limiting performance. Additionally, the serialization only gets worse as threads are added to the application [McK19a].

[4]As shown in Listing 2.7, the `_check_lock` and `_clear_lock` procedures can add instructions of their own to the critical section. However, the number of instructions added is restricted; otherwise, they will not provide acceptable performance.

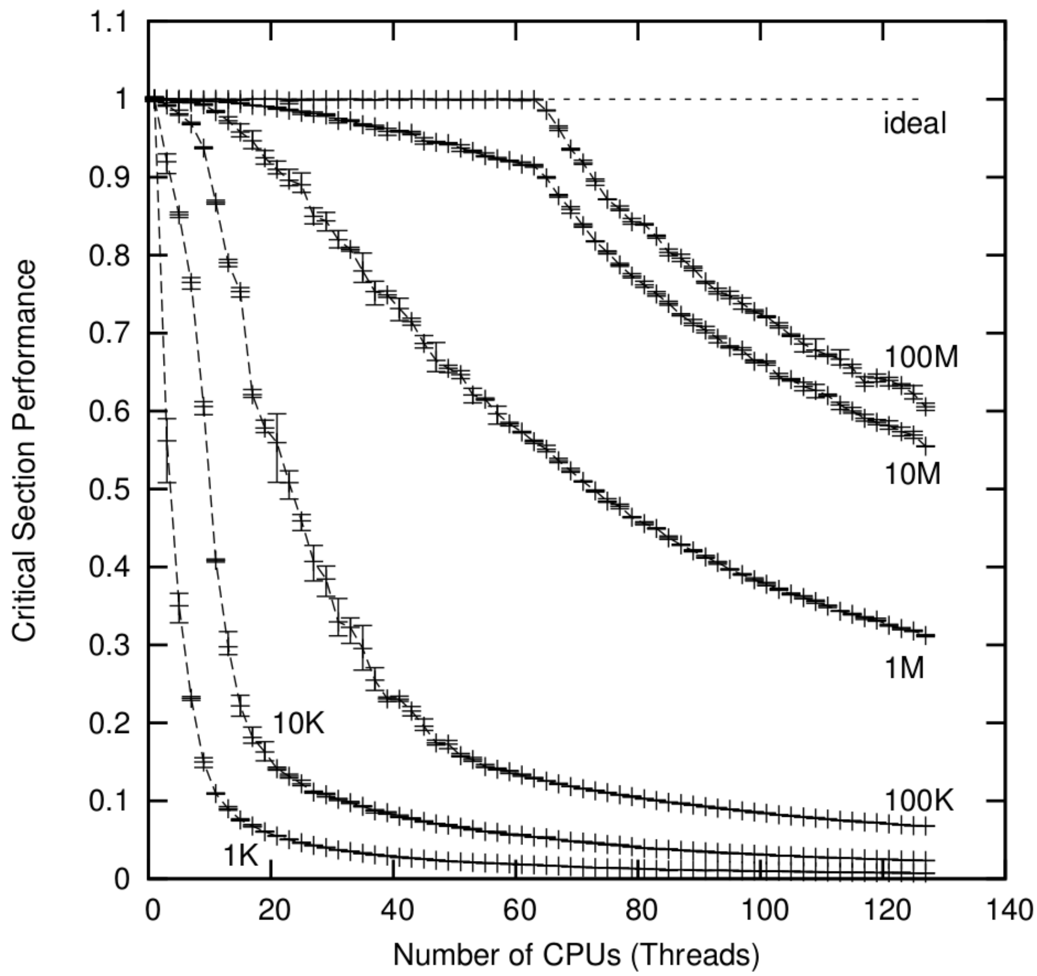


Figure 2.5 – Scalability of the Reader-Writer Lock mechanism. A parameter is used to simulate the critical section range in instructions, which ranges from a thousand (1K on the graph) to 100 million (100M on the graph) instructions [McK19a].

2.2.3 Lock-free Applications

Lock-free applications allow multiple threads to work together to achieve better performance than its lock-based counterpart by avoiding thread suspension. It uses atomic operations to achieve its functionality. The challenge of implementing lock-free applications is illustrated by two examples: lock-free enqueue and dequeue operations.

Listing 2.9 is the lock-free counterpart to the lock-based code shown in Listing 2.6. They have the same functionality and share the same structures showed in Listing 2.5; however the lock pointer `q_lock` is unused for the lock-free implementation.

Earlier, the lock-based implementation achieved its functionality with five lines of code. The lock-free implementation of Listing 2.9 performs the same with 11 lines of code, at a minimum (Lines 8, 9, 11-18, 20, 21). We exclude line 19 as it used only to make the code cleaner. Hence, we had to double the LoC to achieve the same result, which

Listing 2.9 – Pseudo-code of lock-free enqueue operation (Synchronization procedures are colored red) [PKMS17] [MS96].

```
#include <stdlib.h>

#define CAS(ptr, newval, oldval) /* compiler or library implementation */

void
enqueue(struct queue_item *qitem, struct queue *queue)
{
        struct queue_item *last;
        struct queue_item *next;

        while (1) {
                last = queue->q_tail;   /* bug */
                next = last->qi_next;
                if (last == queue->q_tail) { /* bug */
                        if (next == NULL) {
                                if (CAS(last->qi_next, item, next) == item) {
                                        CAS(queue->q_tail, item, last);
                                        return;
                                }
                        } else
                                CAS(queue->tail, next, last);
                }
        }
}
```

decreases simplicity and maintainability. Also, while the lock-based operation is done entirely sequentially, the lock-free operation has an undefined number of loops and six conditional states. Therefore, the performance gain comes with complex and obscure algorithms that are complicated even for experienced programmers to debug [PKMS17].

As the implementation of Listing 2.9 does not use locks, we are limited to execute atomic operations of a single memory position at a given time. Also, as multiple writers are allowed, we must deal with every intermediate state of the procedure. The intermediate states are checked in lines 14, 15, 16, and 20 and corrected in lines 12, 13, and 21. The actual enqueue operation is done with only two lines: 16 and 17. Finally, this C code represents a straightforward adaptation of the algorithm proposed by Michael and Scott [MS96] over 20 years ago. Unfortunately, it has an intermittent bug[5] in lines 12 and 14. If the developer employs code optimization, these lines can result in a single memory operation; all other reads would be done with local registers, as there is no indication on the code that these variables can be changed externally. Consequently, the developer must either make the

[5]The chosen programming language is responsible for this bug – the algorithm description is correct.

variable volatile or use an auxiliary macro to force the memory operation. Note that this bug does not happen when code optimizations are disabled.

2.2.3.1 The ABA Problem

Developers that are transitioning from lock-based to lock-free algorithms are tempted to try to write their lock-free code instead of using existing algorithms. A common problem in lock-free algorithms is the ABA problem [MS96]. We demonstrate this issue next by the faulty implementation of a dequeue operation.

The dequeue operation is the inverse of the enqueue operation; i.e., a node is removed from the queue head. Listing 2.10 shows the lock-based implementation that has a similar structure to the enqueue operation from Listing 2.6. However, it has three differences: (i) the dequeue operation returns a node; (ii) it must check for an empty queue; and (iii) it deals with the head instead of the tail of the queue. The lock-based implementation also achieves its functionality with a critical section of 7 lines (excluded line 14 as it does not generate executable code). For the dequeue operation, we have a conditional test for an empty list to avoid dereferencing the `queue->q_head` pointer (line 12 and 13).

Listing 2.10 – Lock-based dequeue operation.

```
1  #include <stdlib.h>
2  #include <pthread.h>
3
4  struct queue_item *
5  dequeue(struct queue *queue)
6  {
7          struct queue_item       *qi;
8
9          pthread_mutex_lock(queue->q_lock);
10         qi = queue->q_head;
11         if (queue->q_head != NULL) {
12                 queue->q_head = qi->qi_next;
13                 qi->qi_next = NULL;
14         }
15         pthread_mutex_unlock(queue->q_lock);
16         return(qi);
17 }
```

A user attempt at transforming the lock-based to a lock-free operation is shown in Listing 2.11. The code is adapted from [Sta19] to our queue structure from Listing 2.5. [Sna19] shows another implementation susceptible to the ABA problem.

Listing 2.11 – Lock-free dequeue operation susceptible to the ABA problem [Sta19].

```
1  #include <stdlib.h>
2
3  #define ACCESS_ONCE(x)           (*(volatile typeof(x) *)&(x))
4  #define CAS(ptr, oldval, newval) /* compiler or library implementation */
5
6  struct queue_item *
7  dequeue(struct queue *queue)
8  {
9          struct queue_item *qi;
10         struct queue_item *next;
11
12         while ((qi = ACCESS_ONCE(queue->q_head)) != NULL) {
13                 next = ACCESS_ONCE(qi->qi_next);
14                 if (CAS(queue->q_head, qi, next) == qi) {
15                         if (CAS(queue->q_tail, qi, next) == qi ||
16                             next != ACCESS_ONCE(queue->q_tail)
17                                 return (qi);
18
19                         while (ACCESS_ONCE(queue->q_head) == next)
20                                 queue->q_head = ACCESS_ONCE(qi->qi_next);
21                         return (qi);
22                 }
23         }
24         return(NULL);
25 }
```

The lock-free dequeue operation is guarded against the intermittent bug found on the implementation of the lock-free enqueue operation from Listing 2.9, which is achieved using a macro on line 3 – the macro is available from the Linux kernel [Cor19]. Therefore, accesses from lines 12, 13, 16, 19, and 20 are not optimized away. However, the code is not guarded against the ABA problem, since it can reference a node that is not present anymore in the queue or has been reclaimed by the system (i.e., freed). The access to the node can result in fatal access violation errors [Sna19]. Another way to understand the ABA problem is to consider the following statement [Nee19]:

> *"if a CAS operation has succeded, nothing has happened since we read the previous value."*

Unfortunately, the statement only holds for monotonic values, like an increasing counter. For non-monotonic values, like a memory pointer, the statement does not hold and leads to the ABA problem. Table 2.3 shows a possible execution scenario where the ABA problem has occurred. This execution scenario reuses a node that is not present anymore, node `B`, and discard the pointer reference to a pushed node, node `C`.

Table 2.3 – An execution scenario from Listing 2.11 [Sta19].

Time	Execution	Queue state	Comment
1	Thread1: pushes `A` and then `B`	Head: `A`, Tail: `B`	`A` and `B` are arbitrary nodes
2	Thread1: executes dequeue procedure lines 8 to 13; Scheduled before executing line 14	Head: `A`, Tail: `B`	Thread1 executes lines 8 to 13 from Listing 2.11. For this thread, `qi` variable is `A` and `next` variable is `B` and it is about to atomically change `queue->q_head` from `A` to `B`
3	Thread2: pops `A` and then `B`	Head: `nil`, Tail: `nil`	Queue is empty after Thread2 has executed
4	Thread2: pushes `A` again and a new node `C`	Head: `A`, Tail: `C`	`C` is an arbitrary node
5	Thread1: Continues execution from line 14	Head: `B`, Tail: `nil` `B` is a wild pointer; `C` is lost	Thread1 tries and achieves the atomic change of `queue->q_head` from `A` to `B`. Yet, node `B` is not present in the queue anymore and the pointer to node `C` is lost.

To correctly address the ABA problem the code must be redesigned. There are multiple solutions for this – Michael and Scott [MS96] propose one solution free from the ABA problem; Michael [Mic04] also proposed another solution based on hazard pointers[6].

[6] Also independently invented by other researchers [McK19a].

3. RELATED WORK

> Men are more ready to repay an injury than a benefit, because gratitude is a burden and revenge a pleasure.
>
> Tacitus

Data synchronization on concurrent systems has been studied over the past several decades [McK04]. As previously discussed, basic atomic operations, that form the foundation for data synchronization, has been provided by hardware designers, and many techniques have flourished to provide complex synchronization mechanisms for parallel applications.

This chapter reviews major areas of synchronization research and addresses their compatibility with legacy parallel applications. Section 3.1 and 3.2 discuss software- and hardware-oriented state-of-the-art solutions, respectively. Section 3.3 summarizes key characteristics of the discussed works, comparing them to the proposed Subutai solution, and identifying the works that are compatible with legacy parallel applications.

3.1 Software-oriented Solutions

Software-oriented solutions permit developers to synchronize application data with API based on industry-established hardware operations. Developers can use these solutions without the cost of extra hardware components. Also, solutions typically employ the OS to handle some of its functionality to handle sensitive operations such as scheduling policies of threads.

3.1.1 POSIX Threads (PThreads)

PThreads is a standardized C language interface described by the IEEE POSIX 1003.1c standard [IEE16] that specifies a set of thread APIs to do thread synchronization and management. PThreads procedures can be organized into four major groups [Bar19b]: (i) thread management; (ii) mutexes; (iii) condition variables; and (iv) synchronization (rwlocks, barriers). We focus on the last three groups, as they are responsible for dealing with data synchronization. In addition, we assume the default behavior provided by PThreads (i.e., no particular attribute is used).

A mutex is useful for protecting shared data from concurrent access. A mutex has two possible states: unlocked (not owned by any thread) and locked (owned by one, and only one, thread). The mutex procedure group contains locking and unlocking.

Locking is a blocking procedure that exclusively locks a variable. If the variable is already locked, the calling thread is suspended; otherwise, the operation returns with the variable locked by the calling thread. Unlocking is a non-blocking procedure that changes the variable state and, if there are any waiting threads, wakes up previously blocked procedures. If the developer prefers the thread to spin on the lock instead of suspending it, it may use a spinlock, which has the same behavior as the mutex.

The condition procedure group contains: wait, signal, and broadcast. Wait is an unconditionally blocking procedure that puts the thread on a waiting list for a condition event. It requires that designers previously locked a mutex variable and passed a reference to it. Then, the wait procedure unlocks the mutex once it has finished working. Next, when the thread is woken up, the wait procedure reacquires the mutex. The signal and broadcast are non-blocking procedures that wake up one and all threads respectively waiting for a condition event. Mutex, in these cases, is optional.

The last procedure group comprises barriers and rwlocks. The barrier procedure group contains a single blocking procedure, 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 reached the barrier, all threads are woken up. Rwlocks has a similar behavior as a standard lock; however, it differentiates readers from writers. Multiple readers can access the shared data, while only one writer is allowed to modify it. Also, no reader can access the data while there is a writer thread.

Both GNU's Not Unix! (GNU) LibC[1] and FreeBSD LibC utilize operating system calls to do sensitive operations such as putting threads to sleep; however, they operate mainly in user space, as shifting to kernel space may incur performance penalties.

3.1.2 Open MultiProcessing (OpenMP)

OpenMP is an API specification for shared-memory parallel applications. The API supports C, C++, and Fortran for multiple architectures. OpenMP uses a fork-join model of parallel execution, as shown in Figure 3.1. All OpenMP programs start as a single thread called the master thread. It executes sequentially until the first parallel region is encountered. Then, the master thread creates multiple threads to handle the parallel work. The master thread waits for all other threads to finish and then continues to execute sequentially; this process can be repeated arbitrarily [Ope15]. Besides procedures, OpenMP relies on compiler directives to control the application behavior.

OpenMP provides atomic operations that are not provided by PThreads using a compiler directive before the line that is to be executed atomically. For PThreads, the user must

[1] LibC is an implementation of the standard C libraries.

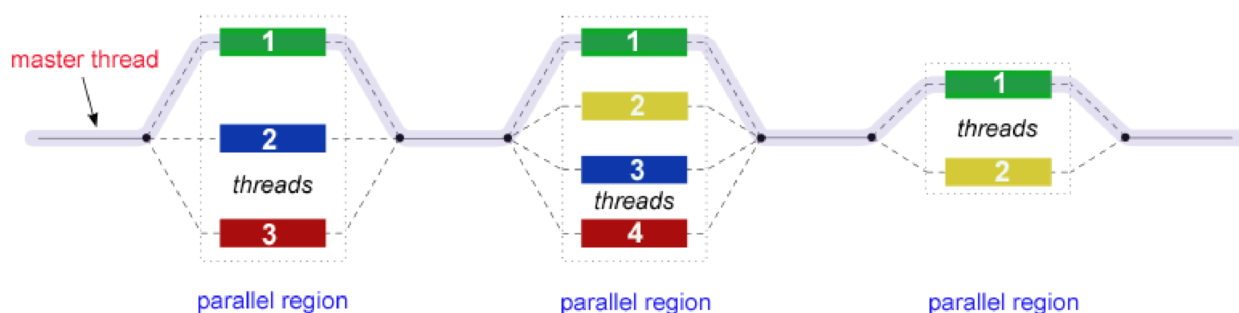


Figure 3.1 – The fork-join model of OpenMP [Bar19a].

use either a mutex lock or a procedure provided by the compiler. GNU Compiler Collection (GCC) and Intel C++ Compiler (ICC) use libgomp and libomp respectively to accomplish the OpenMP specification [GCC19c] [Int19b]. There are two methods for libgomp to handle synchronization: (i) PThreads (Section 3.1.1) if it is available; (ii) own synchronization primitives implementation. Libomp does not use PThreads and implements its synchronization primitives.

3.1.3 Threading Building Blocks (TBB)

Threading Building Blocks (TBB) is an Intel library for parallel applications developed in C++ for the x86 architecture. It offers task-based parallelism that abstracts some of the threading mechanisms. Instead of compiler directives, as done by OpenMP, TBB uses generic programming to fit the object-oriented/template-based programming style of C++ better [Int19a]. Moreover, TBB provides concurrent-friendly data structures for the developer. Hence, the structure handles the synchronization process by itself, either by fine-grained locking or lock-free algorithms [Rei07].

For synchronization, TBB provides the same three basic synchronization primitives as PThreads: mutex, barrier, and condition. Yet, barriers are executed implicitly in template calls and implemented with an additional task with the sole propose of synchronization. TBB implements conditions with the same characteristics and restrictions as described in PThreads (Section 3.1.1). TBB provides several types of primitives for mutexes with contrasting behavior. Table 3.1 shows the traits of some of the mutex types available in TBB, which can be described by the following features [Rei07] [Int19d]:

- **Scalable**[2] – A scalable mutex is one that does no worse than forcing single-threaded performance. A mutex can perform worse than serialize execution if it consumes

[2]As stated by the official manual: "In a strict sense, this is not an accurate name, because a mutex limits execution to one thread at a time" [Int19d].

excessive processor cycles/memory bandwidth. Scalable mutexes are often slower than non-scalable ones under light contention.

- **Fair** – Mutexes can be fair or unfair. A fair mutex lets tasks through in the order they arrive, meaning that fair mutexes avoid starving tasks. On the other hand, unfair mutexes can be faster because they let tasks that are currently running through first, instead of the queued task, which may be sleeping.

- **Sleeps** – Mutexes can cause a task to spin in user space or sleep in kernel space while it is waiting. Spinning is undesirable for long periods, as it consumes multiple processor/cache cycles. For short waits, spinning is faster than sleeping, because putting and waking up tasks takes multiple cycles.

- **Size** – The size requirement for recording the mutex data.

Table 3.1 – Traits and behavior of mutexes. Based on [Rei07] [Int19d].

Mutex	Scalable	Fair	Sleeps	Size
`mutex`	OS-dependent	OS-dependent	Yes	$\geq$ 3 words
`spin_mutex`	No	No	No	1 byte
`queuing_mutex`	Yes	Yes	No	1 word

PThreads provides the `mutex` type for TBB in Linux systems; for this case, the `mutex` type is scalable and fair. `spin_mutex` is implemented with atomic operations, which was discussed in Section 2.2.2. `queuing_mutex` is built with a combination of atomic operations and a queue of waiting tasks. Also, `spin_rw_mutex` and `queuing_rw_mutex` are specialized versions of `spin_mutex` and `queuing_mutex`, respectively, that support reader/writer separation for readers-only concurrency.

3.1.4 Read-Copy-Update (RCU)

RCU is a synchronization strategy that relies on deferring work to a later point in time. The key feature of RCU is that readers can access data even when it is in the process of being updated. Like other lock-free techniques, RCU needs careful use of memory/compiler barriers on the code. Fortunately, the developers have provided a set of procedures that handle pointers, lists, and hash-tables with the appropriate barriers, removing the burden of correct barrier usage from the developer [McK19b].

The basic functionality of RCU is described by the following three steps [McK19c]:

1. Make a change in some structure. For instance, removing a node from a queue.

2. Wait for all pre-existing RCU readers to finish. The wait can be achieved by using a procedure called `synchronize_rcu`. Note that readers are guaranteed to read the updated list if they enter the RCU critical section after the change has been made. For our example, they would not be able to access the newly removed node.

3. Finish any remaining tasks. For instance, freeing the memory area used by the node on the first item.

This functionality resembles rwlocks described in Section 2.2.2.1, yet there are subtle differences that are crucial for the performance of RCU. Rwlocks can only write data if no other reader is present, and rwlock readers must give some information that they are present (frequently by writing to a shared value). RCU does not force any of them. Then, the challenge is how to identify RCU readers if they do not manipulate any shared data. This identification is achieved by the `synchronize_rcu` primitive, whose conceptual implementation is presented in Listing 3.1.

Listing 3.1 – Conceptual implementation of the `synchronize_rcu` primitive [McK19a].

```
1  void
2  synchronize_rcu(void)
3  {
4          for_each_online_cpu(cpu)
5                  run_on(cpu);
6  }
```

The `synchronize_rcu` procedure works by making sure each CPU has executed at least one context switch, as to guarantee that all RCU readers prior to `synchronize_rcu` have finished. Therefore, the implementation has two restrictions: RCU code cannot block and cannot be preempted [McK19c]. After the execution of `synchronize_rcu`, it is safe to clean up any stale data.

The actual implementation of `synchronize_rcu` in the Linux kernel is much more complex, as it has to deal with multiple capabilities expected by the user [McK19c]. Besides, disabling preemption impacts performance. Hence, there are multiple versions of RCU to tackle specific scenarios [McK19b], which allow, for instance, preemption and blocking on RCU critical code.

Figure 3.2 illustrates an example of a node deletion from a linked-list protected by RCU [McK19a]. An RCU writer wants to remove the node `[5, 6, 7]`. It calls `list_del_-rcu` followed by `synchronize_rcu`. These procedures handle the necessary barriers for the developer. Note that existing RCU readers during these operations can still traverse the list either from `[1, 2, 3]` to `[11, 4, 8]`, or `[5, 6, 7]` to `[11, 4, 8]`. After `synchronize_rcu`

has returned, it is safe to clean up the deletion of the node. Hence, the procedure `kfree` is called to claim the memory space.

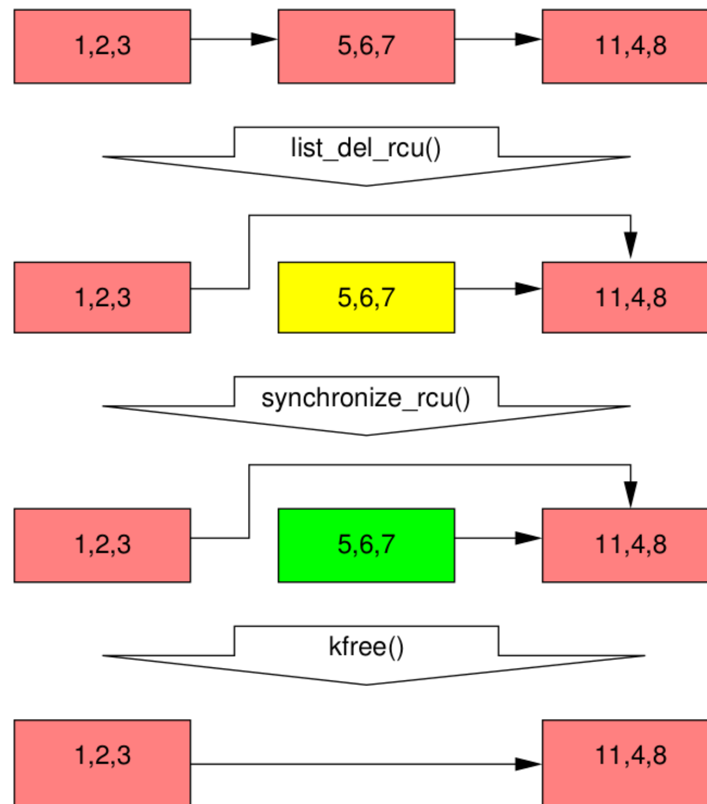


Figure 3.2 – RCU deletion example [McK19a].

Multiple RCU writers may need an external synchronization besides RCU to operate correctly [McK19c], which varies according to the developer implementation. On the user space RCU library [DM19], for instance, a concurrent queue requires a mutex for dequeue operations but does not require it for enqueue operations; it reuses the PThreads' mutex procedures for locking.

3.1.5 Reordering Constraints for PThread-style Locks

Boehm [Boe07] refines the PThread specification to propose a simpler set of clear and uncontroversial rules, allowing the reordering of memory operations for lock synchronization primitives. These rules have a significant performance impact since memory barriers typically limit reorder operations. In addition, the author identifies a class of compiler transformations that can also increase the performance. For these gains, the author proposes a new subset programming language based on C. The objective to propose such language is to verify the correctness of reordering rules under a simplified version of the C language.

The author justifies the proposition of a new language based on the impact of memory barrier operations. Boehm points out the following scenario: the cost of using

lock operations can affect program execution time significantly, even when there is little lock contention. Since the locking cost is often strongly affected by, or even largely determined by, the number of memory barriers, the reduction of memory barrier calls is a worthwhile pursuit.

The central question for the Boehm [Boe07] is thus: under what circumstances can load and store operations be moved into a critical section. His findings suggest that the reordering constraints are not symmetric for lock and unlock operations. Additionally, such behavior was not previously recognized as observed by the PThreads implementation examples provided in this paper.

Table 3.2 complements the information previously shown in Table 2.2 for the use of memory barrier on spinlocks. Boehm demonstrates with these tables that there is much confusion in regards to the correct use of memory barrier with the PThreads standard. Discrepancies in its use occur with the glibc and FreeBSD implementations. In addition, since these mistakes require a specific type of architecture, they may go unnoticed for a long time.

Table 3.2 – Memory barriers used by some PThreads implementations for spinlocks. Adapted from [Boe07].

Environment	lock memory barrier	unlock memory barrier
glibc 2.4 Itanium	full	release
glibc 2.4 x86	full	release
glibc 2.4 ALPHA	acquire	release
glibc 2.4 PowerPC	acquire	release
FreeBSD 6.1 Itanium	acquire	acquire
FreeBSD 6.1 x86	full	full

Boehm defines its unnamed language based on straightforward elements common in a C-like language such as: statements, loops, and variables. We restrict the discussion to data synchronization, which is the topic of this Thesis.

Firstly, Boehm disallows that threads try to relock a lock already owned by that thread; POSIX allows such behavior as undefined behavior, assuming the use of a standard mutex (i.e., created without attributes). Then, Boehm provides simplified lemmas to allow store and load memory operations to move into and out of the critical section, in other words, before and after `pthread_mutex_lock` and `pthread_mutex_unlock` procedures, respectively. There are some restrictions to these movements, mainly I/O and locking operations are not allowed to be moved.

Experimental results are shown in Figure 3.3 for a 2GHz Pentium 4 Xeon. The experimental setup is comprised of a test program that copies 10 million characters, from one file to another, with different types of locks used to control the access to the I/O buffers. Disk access is avoided by using a temporary filesystem residing on memory. The "Default" scheme uses POSIX procedures that are multithreaded-safe (i.e., uses lock internally); "Mutex" and "Spin" are schemes that use POSIX multithreaded-unsafe procedures, but the

author employs a lock and a spinlock around them, respectively; "None" uses a custom-made spinlock implementated by the author. "Lock" and "Unlock" are built on top of "None" and use a full memory barrier for the lock and unlock procedures, respectively; and finally, "Both" is a scheme that uses full memory barrier in the lock and unlock procedures. The results show that the author's implementation is able to reduce almost in half the time spent on the test program compared to "Mutex", which is the worst scenario for this case. The author also notes that spinlocks generally perform better for low contention scenarios.

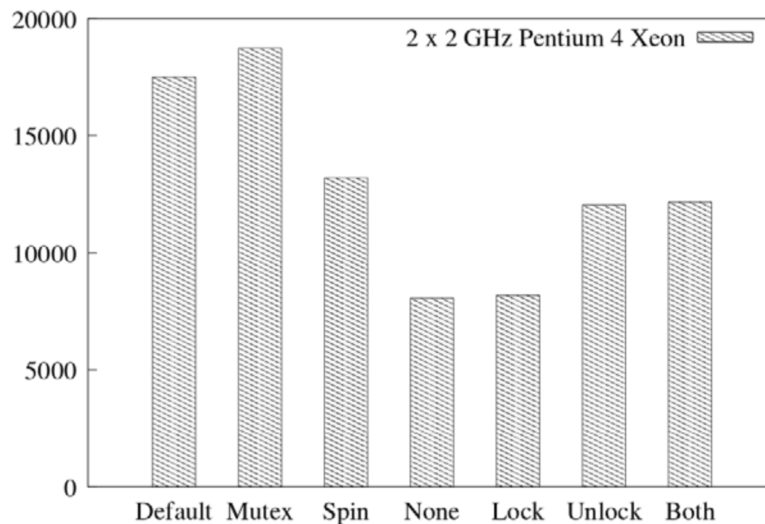


Figure 3.3 – Milliseconds to copy a 10MB file in two threads on a 2GHz Pentium 4 Xeon (lower is better) [Boe07].

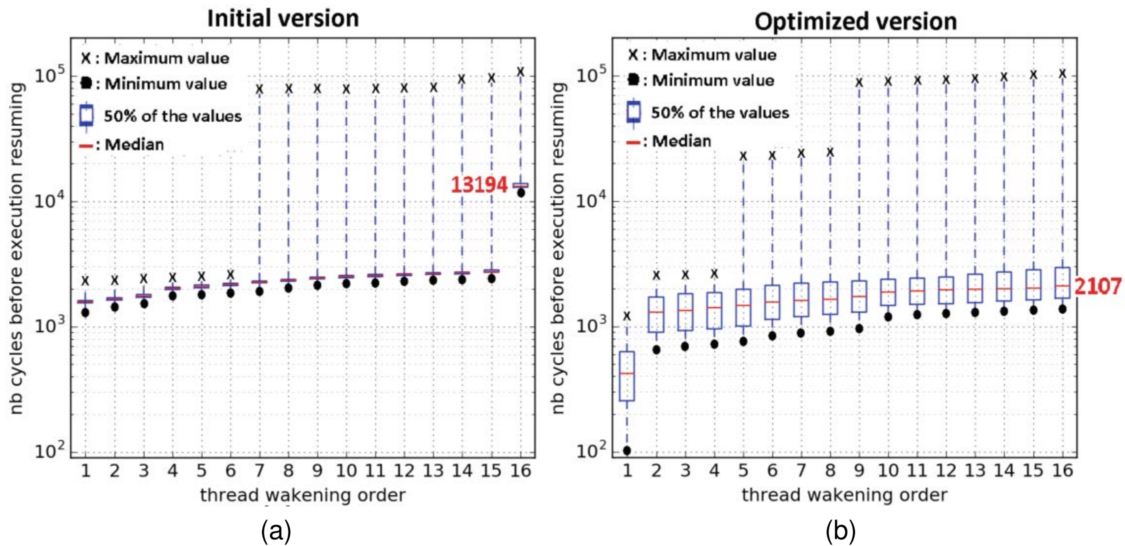
Unfortunately, the author does not provide experimental results on real parallel applications. Hence, the impact of these optimizations is left as an exercise for the reader. In addition, no source code is provided for the author's implementation displayed in Figure 3.3.

3.1.6 Optimization of the GNU OpenMP Synchronization Barrier in MPSoC

France-Pillois et al. [FPMR18] used an instrumented emulation platform to extract precise timing information regarding the use of synchronization barriers of the GNU OpenMP library (i.e., libgomp). They identified that an expansive function was uselessly being called during the barrier waking process. Thus, they propose a software optimization that saves up to 80% of the barrier release phase for a 16-core system. Moreover, as such a change is done at the library level, the optimization is legacy-code compatible.

The evaluation was carried out on the TSAR manycore architecture that supports shared-memory applications. The architecture is organized in four clusters, and each cluster contains four MIPS with a private L1 cache and a shared L2 cache. Figure 3.4 shows the release phase delays by thread arrival. The simulation is a simple for loop executed over 400 times. The X-axis represents the threads in order of release, and the Y-axis represents

the instant the thread leaves the barrier to resume its nominal execution flow. Figure 3.4a illustrates that the original version takes up to 13194 cycles to complete the barrier release process and that a single thread is especially delayed compared to others. Such behavior was the motivator behind the study performed by the authors.



(a) (b)

Figure 3.4 – Delays to complete the barrier release process for a 16-thread application on a 16-core system (a) without and (b) with optimization [FPMR18].

The release phase for the GNU OpenMP library is comprised of two phases: active and passive. When a thread calls the OpenMP barrier, it spins on a memory value that records the number of threads that have arrived on the barrier. This is named the active phase, as it is occupying the core unit. The barrier is only release when that memory value is equal to the user-specified limit. In the example shown in Figure 3.4, the limit is 16. After a specified time, the comparison is stopped, and the thread is put to sleep on a waiting list. Hence, polling is done, and the thread will be woken up by the last arriving thread. This is called the passive phase. The authors noted that the wake-up procedure was being called even in cases where no threads were sleeping. The for-loop used for this work is an example of an application where the threads will normally not be put to sleep, as the application is well balanced. Even when no thread had to be woken up, the time spent in the wake-up procedure was about 12891 cycles, about 97.7% of the whole release procedure for 16 threads. Hence, an optimization was proposed to decrease the overhead of the release procedure, as shown in Figure 3.4b.

Table 3.3 show the gains on the full release procedure for TSAR and Alpha architectures. The gains decrease as the number of CPU increase on TSAR, while the gains remain the same on Alpha regardless of the number of CPU, as in the latter case, the cycle latency is also the same.

Table 3.3 – Gains for the release phase on TSAR and Alpha architectures [FPMR18].

Platform	Threads number	Full release phase delay without optimization	Full release phase delay with optimization	Gain
TSAR	8 on 8 cores	11662 cycles (median)	1481 cycles (median)	87%
	16 on 16 cores	13194 cycles (median)	2522 cycles (median)	81%
	24 on 24 cores	14039 cycles (median)	7975 cycles (median)	43%
Alpha	8 on 8 cores	608 cycles (median)	50 cycles (median)	91%
	16 on 16 cores			
	24 on 24 cores			

The authors provide experimental results on IS, which is a real application from the NAS benchmark. The gain on the total time spent in the release phase is 42.5%, while on the total execution time of the same application was 12.9%.

3.2 Hardware-oriented Solutions

In the previous section, synchronization solutions that do not require any specific hardware to operate besides atomic instructions were presented. This section presents solutions based on novel hardware-assisted synchronization operations. For lock-free algorithms, a severe drawback was discussed (Section 2.2.3): the restriction to atomically change a single memory position. The works of Section 3.2.1 and 3.2.2 tackle precisely such limitation. The work of Section 3.2.3 focuses on a different solution to the mentioned problem: parallel execution of multiple atomic operations. The work from Section 3.2.4 speeds up barrier synchronizations using an independent interconnection. Finally, the work from Section 3.2.5 speeds up data-flow applications for NoC-based architecture designs.

3.2.1 Hardware Transactional Memory (HTM)

Hardware Transactional Memory (HTM) provides an abstraction for running blocks of instructions atomically. It differs from traditional lock-based solutions as the developer needs only to identify which blocks of code must run atomically, and not how concurrent access to shared data must be synchronized. The HTM is responsible for guaranteeing correctness by aborting transactions that conflict with others transactions [DRR14].

Although it is possible to use a software-only transactional memory, the overhead posed by it can be prohibitive [CBM+08]. Fortunately, Intel has provided HTM support since the Haswell architecture, bringing HTM to millions of computer systems [DRR14]. Nevertheless, the Intel implementation has been a victim of numerous issues, prompting the company to disable HTM support for Broadwell CPUs [Was19]. We discuss HTM using the Intel implementation; however, it should be noted that it generally applies to other HTM implementations as well.

Intel provides two interfaces for HTM: Hardware Lock Elision and Restricted Transactional Memory. Hardware Lock Elision is a legacy-compatible instruction set extension that provides hints to the CPU to the start and end regions of the lock elision. For explicit transactions, the developer should use the Restricted Transactional Memory instruction set extension [Kle19a].

Listing 3.2 shows an example of C code using Intel's version of HTM (Restricted Transactional Memory). The code is comprised of a fast path (lines 20-23) when the transaction request has been successful and a slow path (lines 25-30), also called fallback path, when the transaction has failed. Intel recommends that traditional locks be used when the transaction has failed, as shown in Listing 3.2 [Kle19c] [DRR14]. This is called lock-elision

and it is supported by the glibc implementation. The fallback path can also use a retry mechanism with an exponential backoff algorithm. Yet, as it is discussed later, even a single thread execution can be aborted. The code presented in Listing 3.2 is not foolproof as one thread can be using the lock and another thread executing a transaction [DAS19]. This complexity is the reason for incorporating HTM into the synchronization library (such as PThreads), instead of using directly by the developer.

Figure 3.5 displays a set of results from the STAMP benchmark. STAMP is a benchmark suite developed at Stanford for transactional memory research. The results show the execution time for three synchronization techniques: (i) a Single Global Lock (sgl); (ii) software transactional memory (tl2); and (iii) Intel's HTM, called Transactional Synchronization Extensions (TSX). The results are normalized to the single thread execution of sgl. The yada application illustrates that both software and hardware implementations of transactional memory can be slower than sgl for single-threaded execution. Overall, HTM scales better than a coarse-grained lock. One factor that limits the HTM potential for speedup is the abort rate of transactions; even a single thread execution can have aborted transactions. For the STAMP benchmark executing with 8 threads, 7 out of 8 applications had an abort rate of over 70% [YHLR13]. The reason for such high rate abortions is the decision to limit HTM to the L! cache capacity; thus, workloads with large critical sections can be aborted even without concurrency.

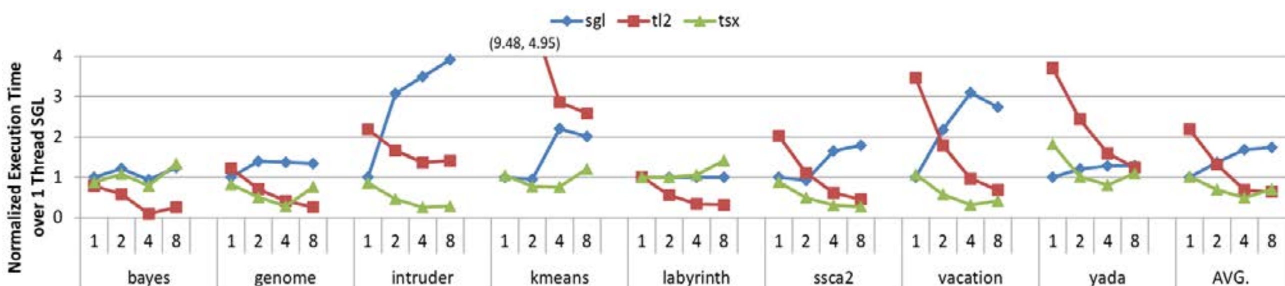


Figure 3.5 – Normalized execution time of eight applications from the STAMP benchmark from 1 up to 8 threads; AVG represents the average execution time of all applications [YHLR13].

Figure 3.6 depicts the scalability of five synchronization schemes on the same application. It shows two interesting scalability issues: (i) the faster synchronization scheme depends on the target number of threads; and (ii) if no application code is changed, as done with the PThreads implementation, the scalability of a synchronization scheme is limited by the choices of the developer [YHLR13]. In other words, a novel synchronization scheme that does not change the application code affects existing code differently [DRR14]. For instance, Intel's HTM is strongly dependent on the access patterns to the L1 cache [DRR14], as it is a critical factor for aborting transactions.

The transactional memory has several compelling research problems that can be improved. They range from the transactional memory design itself, compiler-assisted instrumentation, and HTM tuning mechanisms [DRR14]. Diegues et al. [DRR14] do a

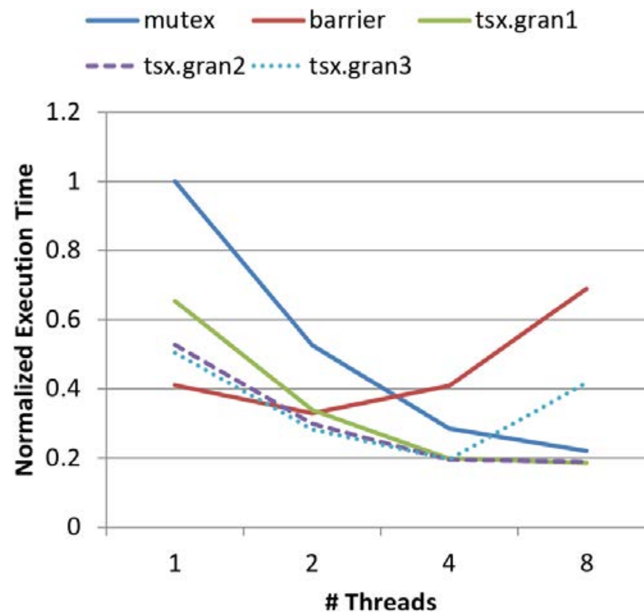


Figure 3.6 – Comparison of five synchronization schemes for PhysicsSolver [YHLR13].

twofold perspective of performance and energy-efficiency for software, hardware, and hybrid transactional memory implementations. Bobba et al. [BMV+08] identify seven performance pathologies in application design that degrades the HTM performance.

3.2.2 A Hardware Implementation of the MCAS Synchronization Primitive

Patel et al. [PKMS17] identify that lock-free algorithms have the potential to be more efficient than its lock-based counterpart is; yet, it is also inherently more difficult to design and debug. Their work focuses on the CAS primitive. As discussed earlier, the CAS primitive can only operate on a single memory location. The design of lock-free algorithms could be significantly eased if that primitive worked on multiple memory locations. Therefore, they propose MCAS, a hardware implementation of a multi-word CAS primitive. The authors depict the simplification with Listings 3.3 and 3.4. The algorithm has already been shown in Listing 2.9, although adapted to use a different data structure. It is intuitively and visually observable that MCAS provides a more straightforward version. From the CAS to MCAS implementation, the code has been reduced from 9 to 3 lines (excluding bracket- and comment-only lines). The MCAS primitive compares the content of `k` variables with `k` memory locations (pairwise), and if all pairs match, then it atomically overwrites the `k` memory locations with `k` new values. Listing 3.4 utilizes a `k` variable of 2.

The MCAS primitive is implemented through two-phase locking. Firstly, cache locking is obtained on all concerned memory locations, and then comparisons are performed.

The memory operations are only executed if the values of all locations are equal to the previously recorded values. A zero-flag is set to 1 for success and 0 for failure. Finally, all cache locking is released.

The ISA is augmented with two instructions: `MTS` and `MCAS`. `MTS` is responsible for setting up the parameters of the MCAS primitive: address of the memory location, recorded old values, and the new values to be written. The `MCAS` instruction, then, executes the operation based on the parameters passed on the `MTS` instruction. In addition, the `MCAS` instruction is also interpreted as a memory barrier operation. Hence, there can only be one active `MCAS` instruction per core.

The hardware required for the MCAS primitive comprises three registers and two tables. The tables are sized to 192×4 and 250×8 bits for up to 32-core architectures and $\mathtt{k} \leq 4$. The tables register the parameters received in the `MTS` instruction, and cache line requests that must be stalled due to cache locking. Hence, the MCAS primitive affects the underlying cache coherence protocol. Hardware synthesis was achieved in 65nm technology and scaled to 14nm operating at 3.4GHz. The area overhead for a 32-core, 400 mm^2 chip area, is 0.0456%.

The Java-based multicore architecture simulator Tejas was used for experimental results on a 32-core system. Some data structures were tested using 32 threads that execute a total of 300 operations each on a shared data structure. They alternate between insertion and deletion of random elements to the data structure. Figure 3.7 summarizes these results. MCAS-OPT is an optimized implementation of MCAS where the instruction `MTS` is executed much earlier than the `MCAS` instruction. The base implementation, MCAS, executes `MTS` exactly before `MCAS`.

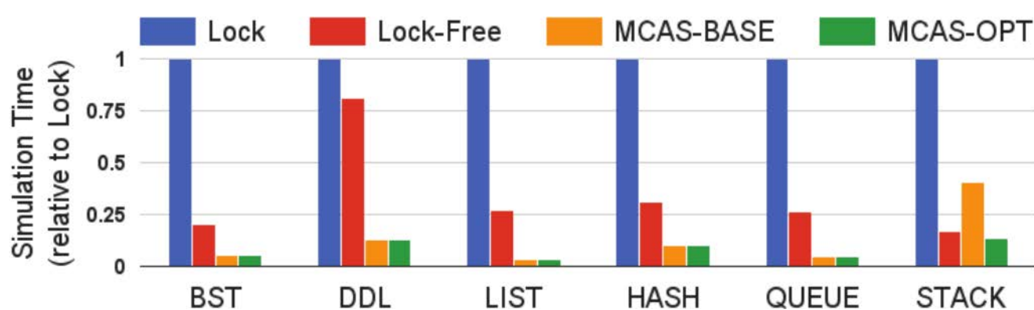


Figure 3.7 – Simulation time comparison [PKMS17].

Both MCAS and MCAS-OPT are 13.8× on average faster than lock-based implementations. The speedup is attributed to the blocking nature of lock-based implementations. Both the MCAS implementations are also faster than the CAS-based lock-free implementation, except for the stack structure where the base implementation of MCAS is slower. The insertion operation for the stack structure is done with a single CAS; hence, for this case, the MCAS implementation resulted in an overhead.

Unfortunately, Patel et al. [PKMS17] did not explore the impact of MCAS on real applications where the impact of many other factors diminishes the gains factors (e.g., the serial portion of the application itself, synchronization decisions). For instance, Abadal et al. [ACAAT16] propose a system with multiple wireless channels to speed up the synchronization process. While it shows significant improvements on micro-benchmarks, when it was tested with the PARSEC benchmark, 9 out of 12 applications showed little improvement in performance (i.e., < 5%). Nonetheless, the solution of Patel et al. has shown a clear advantage over lock-based implementation and a much simpler interface to lock-free applications.

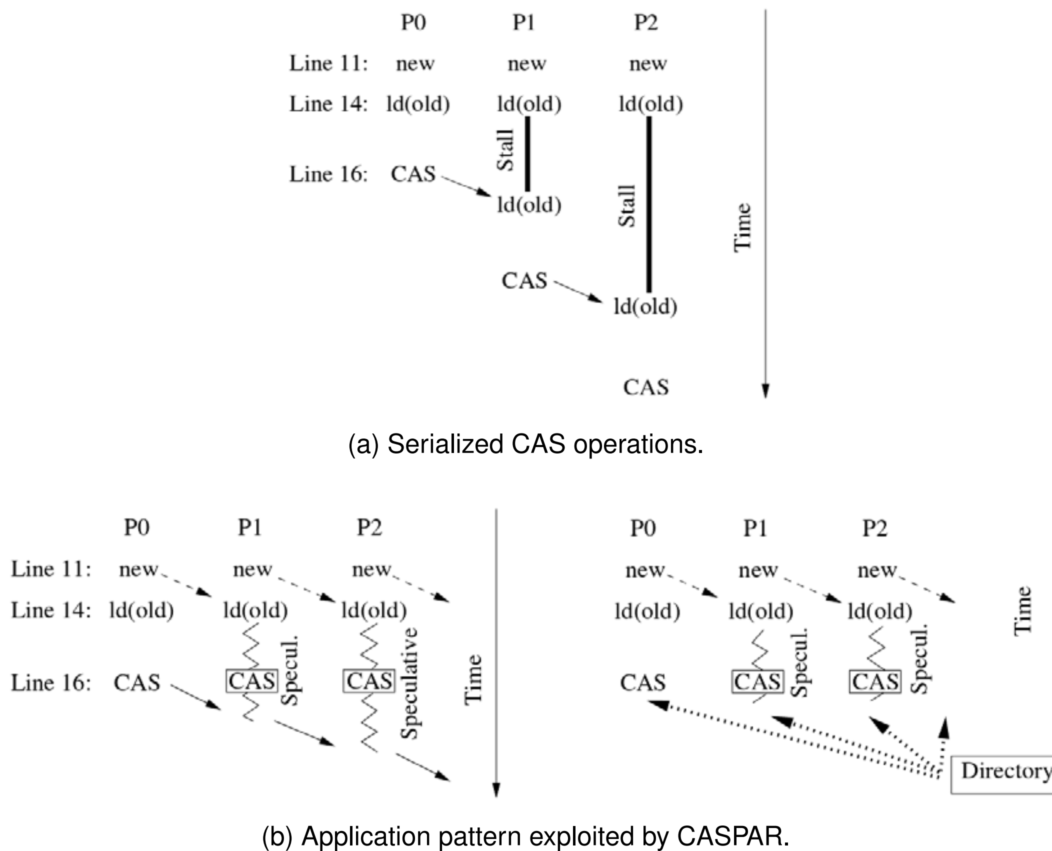
3.2.3 CASPAR: Breaking Serialization in Lock-Free Multicore Synchronization

Gangwani et al. [GMT16] improve the performance of CAS operations by breaking the serialization of multiple CAS calls and executing them in parallel. While Patel et al. [PKMS17] provided a new primitive to write multiple memory positions at the same time, called MCAS, Gangwani et al. propose a novel architecture to parallelize some lock-free parallel applications. Specifically, CASPAR supports applications where the new memory value does not depend on the expected (i.e., old) memory value for the CAS operation call. CASPAR reverts to the serialization of CAS operations for applications that (i) use the expected value to compute the new value or (ii) use mitigation techniques on pointers for ABA handling. No binary code modification is required for using the CASPAR solution.

CASPAR uses a hardware queue to enqueue requests for CAS operations. By itself, the hardware queue still serializes the execution of multiple CAS operations, as they assume exclusive access to a given memory position. An example of serialized CAS execution is shown in Figure 3.8a, where three processors try to write to the same memory position using the CAS primitive. In this example, processor 0 is the fastest one, while the other two processors have to stall their execution waiting for the former processor to finish. This process happens again with processor 2, but now waiting for processor 1.

Moreover, CASPAR needs two additional modules besides the hardware queue: (i) module for identification of contended CAS operations, and (ii) module for parallel execution of multiple CAS operations. All hardware modules proposed by this work are attached to the processor and cache directory.

CASPAR identifies two patterns to exploit for better performance: (i) parallel execution of CAS operations through eager forwarding of the new memory value, and (ii) parallel validation and dequeue of CAS operations. The first idea uses the fact that a queued processor may know early-on the expected memory value that will be set for the shared variable, as this is passed via parameter for CAS operations (the CAS primitive format is as discussed in 2.2.2). Therefore, it eagerly forwards it to its immediate successor in the queue so that the successor processor can overwrite the expected memory value. Hence, a dependency



(a) Serialized CAS operations.

(b) Application pattern exploited by CASPAR.

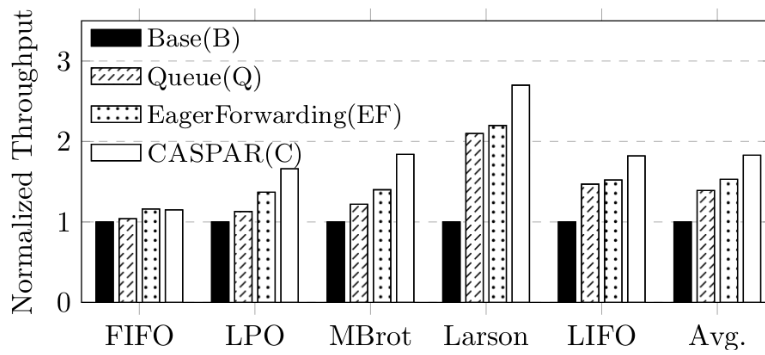
Figure 3.8 – Ideas behind the CASPAR architecture [GMT16].

of queued up CAS operations is created. Since the value passed on is not guaranteed to be written (i.e., CAS can fail), the executions of the successors become speculative. These executions are shown on the left side of Figure 3.8b. The executions go from speculative to committed with the second idea: group validation of multiple CAS operations, and, therefore, group committing and dequeuing of cores. The directory does the verification, and, once confirmed, the group is committed and dequeued in one shot. The group validation is shown on the right side of Figure 3.8b.

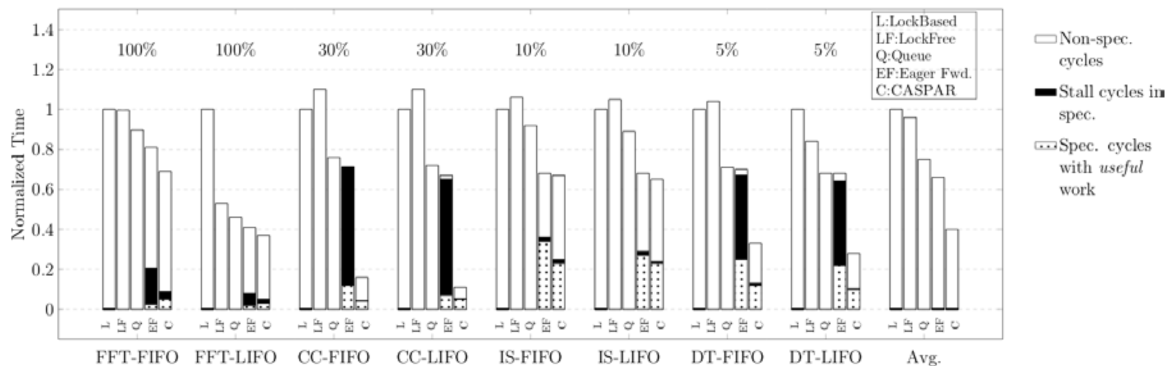
CASPAR is evaluated using a simulated 64-core architecture on the sniper simulator. Figure 3.9 displays the experimental results analyzed in this work.The evaluation uses four computational kernels, one memory allocation kernel, and four applications. In addition, four architecture designs are used for comparison: baseline, a hardware queue for CAS operations, eager forwarding for parallel CAS execution, and the CASPAR design that further the latter design with parallel CAS validation with group commits.

Kernel evaluation, as displayed in Figure 3.9a, is presented through the CAS operation throughput over 5 ms of kernel execution time. On average, eager forwarding and CASPAR improve the throughput by 53% and 83%, respectively, over the baseline design. The gains vary depending on the kernel characteristics.

Figure 3.9b compares the execution time of the applications on different architectures. For this case, the baseline is replaced by two versions: the lock-based and lock-free



(a) Kernel throughput for the different designs (higher is better).



(b) Execution time for applications (lower is better). The number above each application is the fraction of the application time that is simulated.

Figure 3.9 – Experimental results for the CASPAR architecture design [GMT16].

implementations. By rewriting the synchronization in a lock-free manner, the execution time decreases by an average of only 4% – in some cases, the execution time even goes up. The average reduction of execution time is 22%, 12%, and 40% for the hardware queue, eager forwarding, and CASPAR, respectively.

CASPAR introduces two limitations to the architecture design: (i) CASPAR requires that only a single load be exercising its hardware, and (ii) CASPAR forces the processor pipeline to enter in a quiescent (i.e., stalled) state for group committing of CAS operations. Both limit the throughput of the number of instructions per cycle – yet, overall, the experimental results have shown that CASPAR can outperform the serialized scenario.

3.2.4 Design of a Collective Communication Infrastructure for Barrier Synchronization in Cluster-Based Nanoscale MPSoCs

Abellán et al. [AFA+12] identify that barrier synchronization is a key primitive that becomes increasingly challenging as the core count keeps growing. Hardware-accelerated barrier synchronization has been studied at least for the last 20 years [SSP97]. We chose the Abellán et al. work as a representative of this type of research exploration as they use recent technological advances for their study.

A software solution for barrier synchronization relies on each thread communicating to a central place (gather phase) and, when all threads have reached it, the central place communicates back with all threads (release phase). Ideally, the communication back uses a broadcast mechanism to reduce the number of packets. Unfortunately, for NoC-based designs, the broadcast may be unavailable; thus, generating multiple unicast packets. The side-effect of this is the mutual interference between the flows of two traffic, synchronization and data movement, which reduces the overall performance of the system. Their work is focused on standard cell 45nm designs and mainstream industrial tool flow.

They achieve scalability on the MultiProcessor System-on-a-Chip (MPSoC) through core clusterization and replication, where each cluster can potentially operate at an independent frequency. Therefore, the main exploration on barriers of this work is intra-cluster, using asynchronous global links for inter-cluster communication.

Figure 3.10 depicts the topologies proposed to handle barrier synchronization. They are all based on the gather and release phase explained earlier. The Central Barrier (CBarrier) follows a typical Master-Slave structure, where all threads communicate to a central place. Figure 3.11a shows the gather phase for this barrier using C4 as the master. A two-phase gathering procedure is used for the Gline-based Barrier (GBarrier): firstly, all threads are gathered in a horizontal master, where there exists only one per horizontal line. Secondly, the horizontal master communicates with a vertical master. There exists only one vertical master on the cluster. Therefore, all core units are reachable. The two-phase procedure is shown in Figure 3.11b. Finally, Tree-based Barrier (TBarrier) has the lowest number of messages exchanged between master and slaves. This barrier is a simpler version than the GBarrier, yet, it uses a wider line length (2-bit width). The gathering procedure for TBarrier is shown in Figure 3.11c. For the release phase, all topologies follow the same notification flow but in the opposite direction.

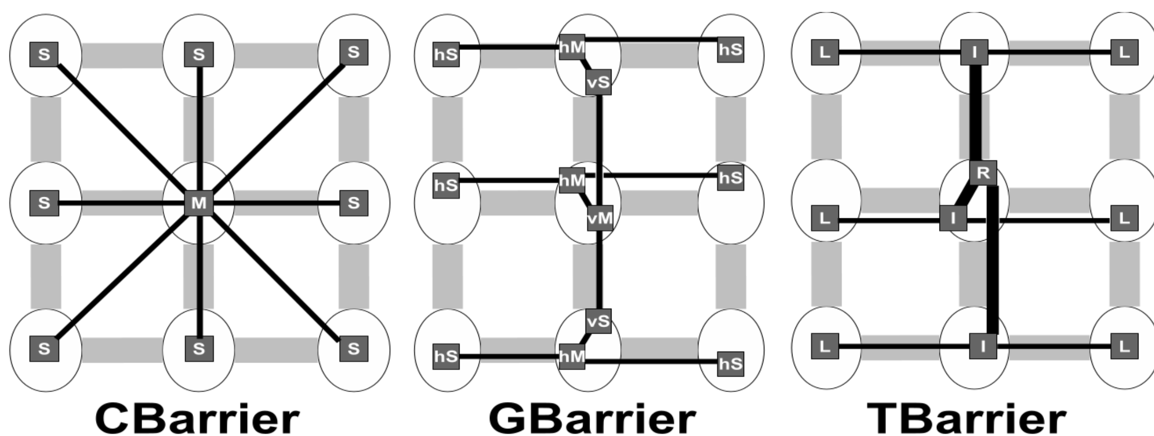


Figure 3.10 – Three barrier synchronization topologies [AFA+12].

The topologies were synthesized with STMicroelectronics 45nm standard cell technology. The results are summarized in Figure 3.12. Figure 3.12a shows that GBarrier has the overall higher frequency, as it has the greatest number of steps to reach synchronization.

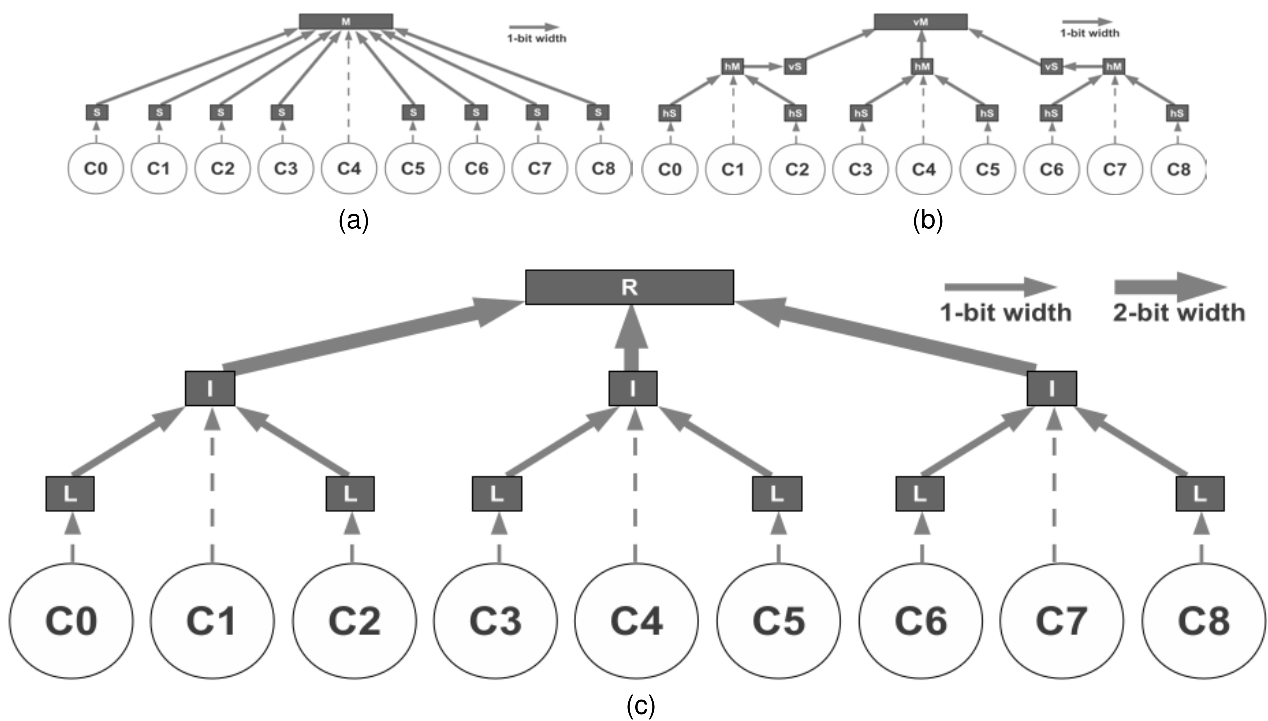


Figure 3.11 – Gather phase for (a) CBarrier, (b) GBarrier, and (c) TBarrier [AFA+12].

Moreover, as the size of the clusters grows, the timing of critical paths for controllers and wires will be longer, which translates into lower achievable frequencies. For all scenarios, CBarrier completes the synchronization with the fewest number of cycles despite its lower frequencies.

Figure 3.12 depicts the area consumption for all topologies running at 600MHz for different-sized clusters. For small clusters, the critical path is defined by the complexity of the barrier controller. For larger clusters, the wire length increase can define the majority of the critical path. In terms of area consumption, the area devoted to wires constitutes the dominant factor for all topologies. CBarrier has the most extended links; hence, it shows the highest overhead, which worsens as the cluster size grows up. For inter-cluster communication, CBarrier can also be used as a flat or a hierarchical design. The flat design has a single master for the entire system, whereas the hierarchical has one master per cluster. For a 64-core system, for instance, there would be 4 clusters with 1 master and 4 masters for flat and hierarchical designs, respectively. The frequency achieved for them was 620MHz and 950MHz for flat and hierarchical designs, respectively. Nonetheless, from the synchronization point-of-view, the flat design is faster, while the hierarchical design almost doubles the number of steps required.

The experimental results were obtained on a full-system simulator, where SystemC models were integrated to simulate the hardware-based barriers, which were annotated with latencies extracted from the synthesis process. The work also discusses the integration of them with the OpenMP software environment. For this, the tree-based barrier was chosen

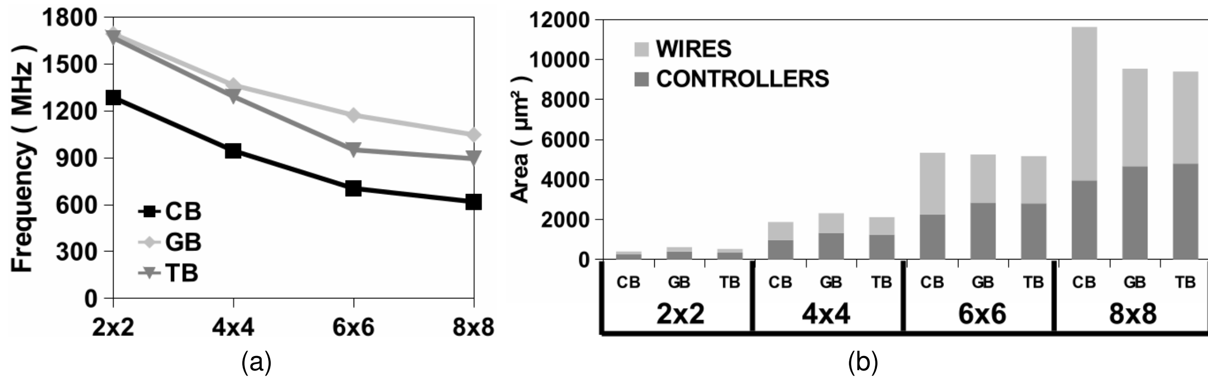


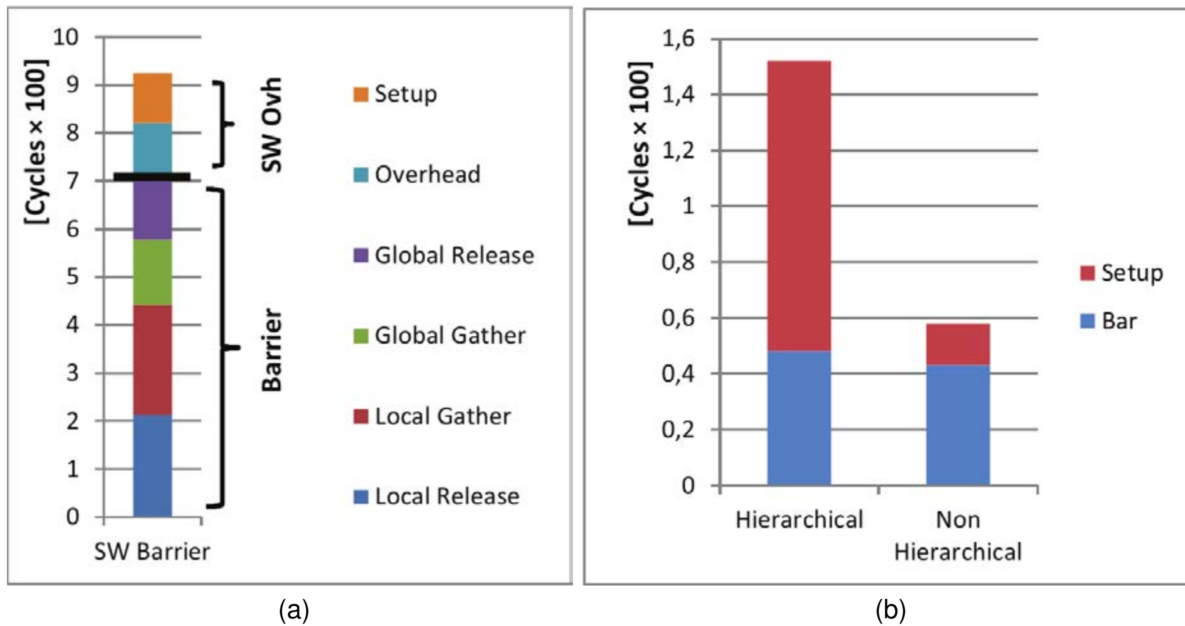
Figure 3.12 – (a) Optimal frequencies and (b) area overhead running at 600MHz for all analyzed topologies [AFA+12].

for both software and hardware simulations. Hence, there is a master core per cluster and a single global master. The software implementation was done by the authors as well using private variables as flags for the gather and release phases. The hardware barriers support both flat and hierarchical designs.

The hardware barriers also support synchronization on a smaller set of cores instead of all of them, but a setup phase is necessary to program the controllers appropriately. For the flat design, it can be easily achieved using a single write to a memory-mapped register of the global master. For the hierarchical design and software implementation, the setup is more complex, as it is necessary to compute the number of clusters and threads involved. Either way, the master thread executes the setup stage at the parallel region creation. Thus, the necessary code is inserted into the `parallel_start` procedure. Besides the register for the setup stage, two more registers are employed: `bar_reg_in` for a core to participate in the gather phase and `bar_reg_out` for waking up threads waiting for the barrier event (completion of the release phase). Hence, the first and second registers are exclusively written by the core and the controller, respectively.

Figure 3.13 depicts the overhead for barrier synchronization in software and hardware. The breakdown of the software implementation, Figure 3.13a, displays that it requires approximately 700 cycles to execute the gather and release phases. Additionally, it has an overhead of approximately one hundred cycles for executing OpenMP procedures and initializing the barrier itself. Overall, synchronizing 64 cores from OpenMP costs slightly more than 900 cycles. Figure 3.13b reports the cost for two hardware implementations. It is interesting to note that the barrier synchronization itself is not very different for both cases, yet, the setup phase is, which is expected as the setup phase is more complex for a hierarchical design. Overall, the flat design has a faster execution time than its software counterpart, and the hierarchical design can be faster if the number and location of threads do not change, as otherwise, the setup phase has to be recomputed.

Unfortunately, as was the case with Patel et al. [PKMS17] work, no results were shown for real applications. For Abellán et al. [AFA+12] work, it would be especially interesting



(a) (b)

Figure 3.13 – The barrier cost of (a) SW and (b) HW implementation in cycles [AFA+12].

to demonstrate the impact on real applications as they have added support for their solution in OpenMP. However, they implemented a synthetic benchmark where the granularity of work between barriers varies from 10 to 10000 cycles. The results of this benchmark are presented in Figure 3.14. For extremely small workloads ($<$ 100 cycles), the barrier overhead dominates the execution time. If a target 5% of overhead is desired, then the hardware implementation reaches it at a granularity of a thousand cycles, while the same point is reached at ten thousand cycles for the software implementation. Finally, it is worth noting that from the software perspective, the latency difference between the two implementation of hardware is negligible since the overhead from the software stack tends to hide it.

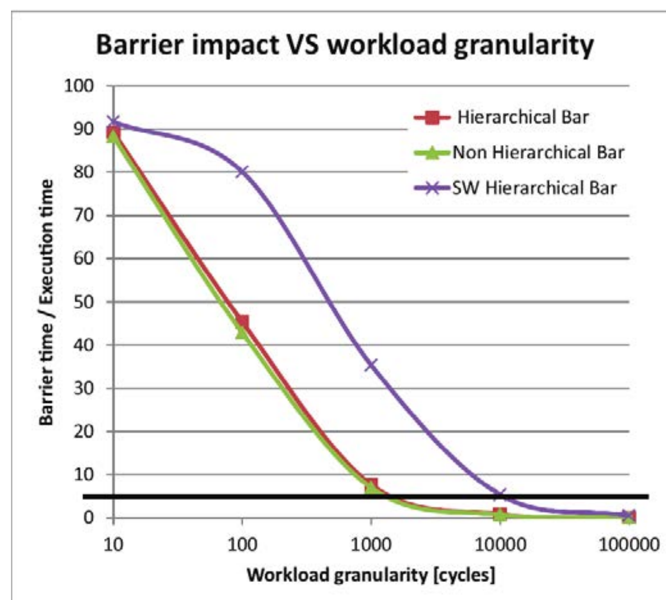


Figure 3.14 – Barrier overhead for varied-sized parallel workload [AFA+12].

3.2.5 Notifying Memories: a case-study on Data-Flow Application with NoC Interfaces Implementation

Martin et al. [MRSD16] identify that NoC-based systems increase the communication latency significantly for data-flow applications compared to traditional bus-based systems. They introduce the Notifying Memories concept to reduce this overhead by eliminating useless memory requests for these applications. The synchronization primitive explored in this work is the spinlock – hence, threads busy-wait on events instead of sleeping. Additionally, this work does not employ caches.

The data-flow software model offers a well-defined manner to deal with software complexity and scalability. Data-flow actors must check firing rules related to input data and output buffer space. An example of data-flow code is presented in Listing 3.5. An action is fired (i.e., executed) when a set of conditions are satisfied. This so-called firing rule usually consists of checking the number of tokens available in the input and output First-In-First-Out (FIFO). If that is not the case, the process has to be re-executed.

However, the continuous testing of firing rules results in many memory request packets. Worse, most of them may be useless, as the memory has not been changed. In the worst case, the software implementation can produce six requests for a single failed firing action. Therefore, this work proposes to address this problem with a novel approach: transform memories into masters able to initiate transfers by means of notifications when data is ready; thus, getting rid of useless memory request packets. This concept is called Notifying Memories. It provides memories with notification and processors with listening mechanisms, which are conceptually similar to the observer design pattern.

Experimental results were conducted targeting the MPEG-4 decoder for different video sequences. Table 3.4 summarizes the percentage of unsuccessful firing attempts and their reasons.

Table 3.4 – Unsuccessful firing rules attempts and its reasons [MRSD16].

Video		Useless	Empty	Full
Sequence	**Format**	**attempt**	**input FIFO**	**output FIFO**
Akiyo	CIF	42.7%	63.7%	36.3%
Parkjoy	720p	21.3%	90.8%	9.2%
Foreman	CIF	34.8%	90.7%	9.3%
Coastguard	CIF	27.8%	98.4%	1.6%
Stefan	CIF	25.9%	83.3%	16.7%
Bridge far	QCIF	23.8%	38.4%	61.6%
Ice	4CIF	45.6%	70.4%	29.6%

There are two possible reasons why no action can be performed: (i) one of the input FIFOs does not contain enough tokens; or (ii) one of the output FIFOs does not contain enough space. Note that applications described in C, as done in Listing 3.5, do not test the second condition if the first one has failed. The results show that at least 20% of attempts are unsuccessful and go up to 45%. This observation motives the integration of mechanisms able to monitor the FIFO status and to emit notifications.

The implementation of the Notifying Memories is shown in Figure 3.15 for a target architecture composed of 13 cores and 15 memory modules distributed in a mesh topology. NIs are enhanced with two new modules: listener and notifier. The former is responsible for receiving notifications on FIFO changes made from the latter. The solution is agnostic to the implementation of the interconnect, cores, and memory modules. Both notifier and listener are highlighted in orange in Figure 3.15.

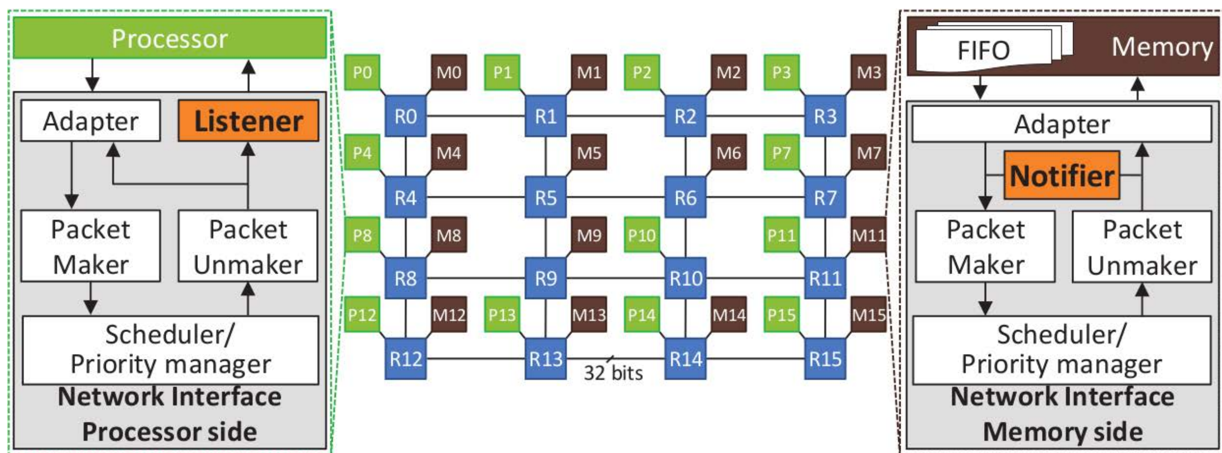


Figure 3.15 – Target architecture with Notifying Memories [MRSD16].

The notifier comprises a couple of logic components and a set of registers. It has three phases of operation: configuration, checking, and notification phases. Firstly, a manager core is responsible for informing all notifiers of the mapping data. Once configured, the notifier checks for packets that modify the FIFO status. It uses a comparator to check if the new FIFO status respects the restrictions provided by the application. For instance, in Listing 3.5, if a new packet related to `fifo_in_1` has been received, it checks the condition `nb_of_tokens(fifo_in_1) >= 64`. If the restriction is satisfied, a flag is set on a bank of the notifying registers. Finally, the notifier loops around the bits of the bank of notifying registers and generates the notifying packets consecutively for flagged bits.

The listener is a more straightforward module also comprised of logic components and a set of registers. It operates only in two phases: configuration and execution phases. Similarly to the notifier, it receives the application mapping from the start and configures its internal registers. In the execution phase, the listener receives notifications from the notifier and provides this data to the local core.

Both modules were synthesized on a design where each memory core has access to a notifier, and each core has access to a listener. Hence, the total number of notifiers and listeners are 15 and 12, respectively. Each one of them comprises 145 registers, for 145 firing rules. The synthesis was achieved with a 65nm CMOS process technology operating at 500MHz. The proposed solution has an area overhead of 12.4% compared to a reference NoC. It also increases the power consumption to a value of 16.3%. Yet, overall, the system can save power by decreasing the number of packets on the NoC, as shown in the experimental results.

The MPEG4-SP decoder with a diverse set of videos was employed for the experimental results. A model of the application was simulated into a SystemC NoC simulator. Also, the actors were mapped manually to minimize the number of hops of communication. The results are summarized in Table 3.5.

Table 3.5 – Notification memory gain for decoding 10 frames of five video sequences [MRSD16].

Video		**Throughput**	**Latency**	**Injection rate**	**Switch conflicts**	**Flits number**
Sequence	**Format**					
Bridgefar	QCIF	+15.53%	-73,96%	-45,80%	-71,38%	-54,22%
bus	CIF	+2.84%	-73,79%	-53,40%	-72,90%	-54,73%
grandma	QCIF	+16.79%	-68,96%	-60,78%	-85,50%	-67,36%
foreman	CIF	+14.26%	-78,41%	-46,81%	-72,86%	-54,39%
ice	4CIF	+15.41%	-78,44%	-50,53%	-75,33%	-58,16%

The average results confirm the efficiency of Notifying Memories leading to reductions of 78% for latency, 60% for injection rate, 67% for transported flits, while improving throughput by up to 16%, approximately. The reduction can be seen by using, for instance, the ice video sequence. Packets in the NoC are organized into data and control categories. The former holds tokens or requests for reading tokens information, and the latter holds mapping information, setting and reading the FIFO structure, and notification signals. The categories also apply to flits. The ice sequence demanded 19 times more control packets and 10 more control flits for the reference system than the proposed system. The values for the ice sequence are shown in Figure 3.16.

Figure 3.16 – Classification of packets transported after 10 decoded frames of ice [MRSD16].

Listing 3.2 – Example of HTM-enabled parallel code.

```
1  #include <immintrin.h>
2  #include <pthread.h>
3
4  struct data {
5          unsigned int     status;
6          pthread_mutex_t *lock;
7          void             *data;
8  };
9
10 void
11 data_init(struct data *dt);
12
13 void
14 func(void)
15 {
16         struct data dt;
17
18         data_init(&dt);
19         dt.status = _xbegin();
20         if (dt.status == _XBEGIN_SUCCESS) {
21                 /* critical section start */
22                 /* critical section end */
23                 _xend();
24         }
25         else {
26                 /* fallback path */
27                 pthread_mutex_lock(dt.lock);
28                 /* critical section start */
29                 /* critical section end */
30                 pthread_mutex_unlock(dt.lock);
31         }
32 }
```

Listing 3.3 – Lock-free CAS algorithm [PKMS17].

```
1  while (true) {
2          last = tail; // 'tail' is the queue's tail
3          next = last->next;
4          if (last == tail)
5                  if (next == NULL)
6                          if (CAS(&(last->next), next, newNode) == next) {
7                                  CAS(&tail, last , newNode);
8                                  return;
9                          }
10                 else
11                         CAS(&tail, last , next);
12 }
```

Listing 3.4 – Lock-free MCAS algorithm [PKMS17].

```
1  do {
2          last = tail;
3          next = last->next;
4          // first argument (=2) is the arity
5          result = MCAS(2, &(last->next), &tail, next, last, newNode, newNode);
6  } while (result == false);
```

Listing 3.5 – Example of Dataflow application flow. Adapted from [MRSD16].

```
#include <stdlib.h>

struct df_fifo {
        size_t          df_writer, df_reader;
        size_t          df_nb_readers;
        void           *df_data;
        size_t          df_data_len;
};

struct df_fifo *fifo_in_1, *fifo_in_2;

#define FIFO_OUT_SZ            256
struct df_fifo *fifo_out;

/* returns the number of tokens on the FIFO */
size_t
nb_tokens(const struct df_fifo *fifo);

void
fire(void)
{
again:
        if (nb_tokens(fifo_in_1) >= 64 && nb_tokens(fifo_in_2) >= 1 &&
            (FIFO_OUT_SZ - nb_tokens(fifo_out)) > 64) {
                /* execute firing action */
        }
        else
                goto again;
}
```

3.3 Discussion

Data synchronization is an essential component to many parallel applications. The strategy of how to employ it affects the parallelization potential of these applications directly [YHLR13]. Here, we explored a diverse set of strategies for data synchronization that presented distinctive tradeoffs in complexity and parallelization. Generally, complexity and parallelism are proportional as parallelization introduces non-determinism into the system, and it must be dealt with additional code to avoid deadlocks, livelocks, and error-prone scenarios [BAAS09]. In other words, non-determinism increases performance at the cost of code complexity.

Table 3.6 shows a comparison of the reviewed work summarized in 7 topics. The first topic is the name of the solution. The second topic is the orientation of the solution: software-based, hardware-based, or a mixed solution. In essence, every solution is hardware-based, as they require basic hardware to operate. Thus, we distinguish hardware-based from software-based for cases where specific hardware has to be employed, and it is not available in industry architecture specifications. If, on top of that, the software must be changed in some way (i.e., application or library changes) the solution is called a mixed solution. For instance, MCAS is an example of a mixed solution as: (i) it requires a hardware module not currently available[3]; and (ii) it forces the developer to transform the application code (i.e., multiple CAS calls are reduced to a single MCAS). Conversely, Notifying Memories is a hardware-oriented solution, as neither the application nor the synchronization library is altered.

The third topic enumerates any additional requirements for a solution besides providing data synchronization. Latency is an essential requirement for all solutions analyzed. Also, most of the solutions allow developers to employ any desired application model. PThreads, for instance, exposes the control of parallelism at its lowest level. As such, it offers maximum flexibility [Rei07]. OpenMP, TBB, and Notifying Memories provide specialized solutions for one application model: fork-join, task-based, and data-flow, respectively. Finally, most of the hardware-enabled solution also limit their area consumption.

The fourth topic shows that only three of the analyzed solutions also target legacy code: France-Pillois et al. work, CASPAR and Notifying Memories. On the one hand, the France-Pillois et al. work is an optimization of the OpenMP library, which is implemented entirely in software. CASPAR and Notifying Memories, on the other hand, are implemented entirely in hardware. While Notifying Memories do not require application modification, they do require a setup phase at boot time that must be provided by the developer. Conversely, our solution does not require any setup phase. HTM is also legacy code compatible if it

[3]MCAS can be simulated in a transaction on HTM-enabled hardware; yet, it may not perform the same as the hardware implementation. For example, a transaction can abort in situations that MCAS would not.

Table 3.6 – Related work summary.

Solution	Orientation	Requirements	Legacy code compatible	Uses PThreads	Target data synchronization	Experimental results
PThreads	Software	Latency	No	Yes	Barrier, Condition, Mutex	Many real applications
OpenMP	Software	Latency and application model	No	Yes (libgomp)	Atomic, Barrier, Mutex	Many real applications
TBB	Software	Latency and application model	No	Yes (Linux)	Atomic, Condition, Mutex	Many real applications
RCU	Software	Latency	No	May use	RWMutex	Linux kernel
Boehm	Software	Latency and correctness	Maybe	Yes	Mutex	Synthetic microbenchmark
France-Pillois et al.	Software	Latency	Yes	Indirectly (OpenMP)	Barrier	IS and microbenchmarks
HTM	Mixed	Latency	Maybe	May use (recommended)	Mutex	Indirectly (PThreads)
MCAS	Mixed	Latency and area	No	No	Atomic	Synthetic microbenchmarks
CASPAR	Hardware	Latency	Yes	No	Atomic	Applications and microbenchmarks
Hardware-based barrier	Mixed	Latency and area	No (Not addressed)	Indirectly (OpenMP)	Barrier	Synthetic microbenchmarks
Notifying Memories	Hardware	Latency, application model, area	Yes	May use (spinlock)	Spinlock	MPEG-4 decoder
Subutai	**Mixed**	**Latency and area**	**Yes**	**Yes**	**Barrier, Condition, Mutex**	**PARSEC and microbenchmarks**

is restricted to the PThreads library. For all other cases, the burden of transforming the application code is left for the developer. The compatibility of these solutions with Subutai will be discussed in Section 3.3.2.

The fifth topic shows that PThread can be employed in 9 of the 11 analyzed works. It can be used directly by the developer in PThreads, HTM, TBB, RCU, and Notifying Memories. For OpenMP, Boehm, France-Pillois et al. work and Hardware-based barrier, the use is indirect: the OpenMP implementation library can use it, as done, for instance, on the libgomp. MCAS and CASPAR are used for lock-free algorithms; hence, they avoid lock-based synchronization libraries such as PThreads.

The sixth topic is the target data synchronization primitive natively supported by the solution. Thus, we exclusively show the primitives that do not require code implementation by the developer. Subutai differs from other optimization solution as it is the only one that targets multiple synchronization primitives (barriers, conditions, and mutex). The other optimization solutions always target a single synchronization primitive, i.e., locking (RCU, HTM, and Boehm), barriers (France-Pillois et al. work and Hardware-based barrier), spinlocks (Notifying Memories), and atomic operations (MCAS and CASPAR). PThreads, OpenMP and TBB also support a variety of synchronization primitives; however, they are intended to provide a generic API for general-purpose use, instead of optimizing an existing implementation.

The seventh topic is the experimental results for the analyzed solutions, respectively. All solutions, besides Notifying Memories, target the shared-memory paradigm. This is expected as the shared-memory paradigm shares the address space for all threads; thus requiring some form of access policy to control it. The use of a synchronization solution

defines the access policy. 8 out of 11 solutions uses at least one real application. Boehm, MCAS and Hardware-based barrier do not provide such results.

Table 3.7 – Related work key contribution.

Solution	Key contribution
PThreads	Cross operating system support for multithreaded application via standardized POSIX interface
OpenMP	API specification for portable multithreaded application using the fork-join programming model
TBB	Library for parallel programming that provides a higher level of abstraction compared to PThreads and OpenMP
RCU	Block-free read access to shared data even when the data is being updated
Boehm	Relaxing PThreads reordering constraints for a subset of the C language
France-Pillois et al.	Optimize the release procedure for the GNU OpenMP library
HTM	Abstraction that provides the transaction concept for running atomically blocks of code
MCAS	CAS procedure that operate over a range of memory positions at the same time
CASPAR	Execution of multiple CAS operations in parallel using specialized hardware
Hardware-based barrier	Multiple hardware solutions for hardware-based barrier synchronization
Notifying Memories	Hardware-accelerated notification for firing rules of data-flow applications
Subutai	**Hardware-accelerated PThreads synchronization primitives for legacy-compatible parallel applications**

Subutai is presented in the last line of Table 3.6. As it requires hardware modifications and changes a synchronization library, it is a mixed solution. As common to other hardware-based solutions, it limits latency and area consumption. Subutai is compatible with any parallel application already employing PThreads library, as the software changes are done entirely in the library. We chose to be compatible with PThreads, as will be discussed in the next subsection; hence, it is compatible with the PThreads interface. Finally, Subutai targets shared-memory paradigm applications, and this work uses the PARSEC benchmark for experimental evaluation.

Table 3.7 depicts the key contribution of each work for the data synchronization area of research. The contribution is the most important aspect of the work captured in a single phrase.

Four works (PThreads, OpenMP, TBB and HTM) are generic API specifications for cross-platform use. All other works are optimization on existing APIs, except for RCU,

as it creates a rwlock capable of reading and writing at the same time. Boehm optimizes memory barriers for PThreads lock and unlock procedures. The France-Pillois et al. work and Hardware-based barrier optimize the use of barriers on 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 application and synchronization scenario: data-flow and spinlocks, respectively. Finally, our solution accelerates PThreads data synchronization primitives through hardware execution while maintaining legacy-code compatibility.

3.3.1 The Choice of PThreads

From the multiple possibilities of legacy compatible interfaces, we chose the PThreads interface. Subutai, which will be described in the next chapter, transforms software events (e.g., locking, condition wait) to hardware events (e.g., packets). As such, we can target any number of available library interfaces. The PThreads interface was chosen for two reasons: (i) it is widely employed as a *de facto* standard to parallel application implementation; and (ii) as shown in Table 3.6, it is used internally as the base of other synchronization solutions. Therefore, PThreads provides a broad range of applicability to Subutai.

3.3.2 Subutai Compatibility with Other Legacy-code Compatible Solutions

The following solutions are legacy-code compatible besides Subutai: Boehm, France-Pillois et al., CASPAR, Notifying Memories, and HTM. The works of Boehm and France-Pillois et al. are entirely done at the software level and do not apply directly 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 spinlock usage is done through PThreads (i.e., `pthread_spin_lock`). Unfortunately, that is not the case with the paper presented. Besides, Notifying Memories only targets the data-flow application model, while we support any model that uses the shared-memory paradigm. Finally, HTM can cooperate with our solution and will be detailed in Section 3.3.2.1.

3.3.2.1 HTM

One of the HTM operation modes is functionally similar to Subutai: implementation restricted to the PThreads library. In this case, HTM is also legacy-code compatible. As such, HTM proves that our solution is feasible.

An essential feature of Subutai is its compatibility with other solutions employing PThreads. Hence, HTM and Subutai are not mutually exclusive. In fact, Subutai can work cooperatively with HTM. As discussed in Section 3.2.1, the recommended way to deal with the fallback path of a transaction is to use a traditional lock from, for instance, PThreads. In addition, a transaction may have to check if the said lock is owned currently by any other thread even if it is executing on the fast path (i.e., inside a transaction) to avoid race conditions. Subutai can accelerate both these scenarios when the lock uses the PThreads interface.

Listings 3.6 and 3.7 provide an example of a transaction using a PThreads lock. Initialization of the PThreads lock and error-checking has been omitted to simplify the example design for both Listings. Listing 3.6 shows a naive, but intuitive, implementation of a transaction to update a shared variable. The shared variable `shared_var` and associated lock `lock` are declared in lines 5 and 6, respectively. The transaction is executed in lines 12 and 13 to update the shared variable and finish the transaction, respectively. If the transaction fails, the fallback path is used (lines 15, 16, and 17). As recommended, a lock is used in this case. Unfortunately, the implementation is not correct: the lock is not providing mutual exclusion – if a transaction is started while another thread is in the fallback path, an increment to the shared variable may be lost [Kle19b].

The race condition is solved with Listing 3.7. In both paths, transaction and fallback, the lock is checked (lines 13, 18, and 20). The transaction only needs to check the status of the lock, while the fallback path needs to own it.

A restriction of Listing 3.7 is the access to the internal structure of `pthread_mutex_t`. A user application should only use opaque pointers for PThreads. It is valid, though, to access the internal structure in the library itself, which is the case for HTM and this listing [Kle19b]. The internal structure of `pthread_mutex_t` is presented in the next chapter (Listing 4.1).

In sum, both paths of HTM deal with a PThreads lock. These accesses can be handled and accelerated by Subutai. The implication is that HTM is complementary to our work.

Listing 3.6 – Intuitive, but incorrect, implementation for an HTM-enabled application [Kle19b].

```
#include <stdlib.h>
#include <immintrin.h>
#include <pthread.h>

size_t                  shared_var = 0;
pthread_mutex_t         lock;

void
func(void)
{
        if (_xbegin() == _XBEGIN_START) {
                shared_var++;
                _xend();
        } else {
                pthread_mutex_lock(&lock);
                shared_var++;    /* read, modify, write */
                pthread_mutex_unlock(&lock);
        }
}
```

Listing 3.7 – A correct implementation for an HTM-enabled library. Based on [Kle19b].

```
#include <stdlib.h>
#include <immintrin.h>
#include <pthread.h>

size_t                  shared_var = 0;
pthread_mutex_t         lock;

void
func(void)
{
        if (_xbegin() == _XBEGIN_START) {
                /* is any owner present? */
                if (lock.__data.__owner) /* access to internal structure */
                        _xabort(0xff);   /* lock is busy */
                shared_var++;
                _xend();
        } else {
                pthread_mutex_lock(&lock);
                shared_var++;    /* read, modify, write */
                pthread_mutex_unlock(&lock);
        }
}
```

4. SUBUTAI SOLUTION

"Paul has since convinced some of us [kernel developers] that compiler writers are pure evil and out to get us."

I have seen the glint in their eyes when they discuss optimization techniques that you would not want your children to know about!

Peter Zijlstra and Paul McKenney: page-table walkers vs memory order

Subutai is a synchronization solution for legacy and novel parallel applications. Subutai is comprised of a software/hardware co-design to perform fast synchronization operations. Figure 4.1 highlights the components of Subutai for a general-purpose computing stack.

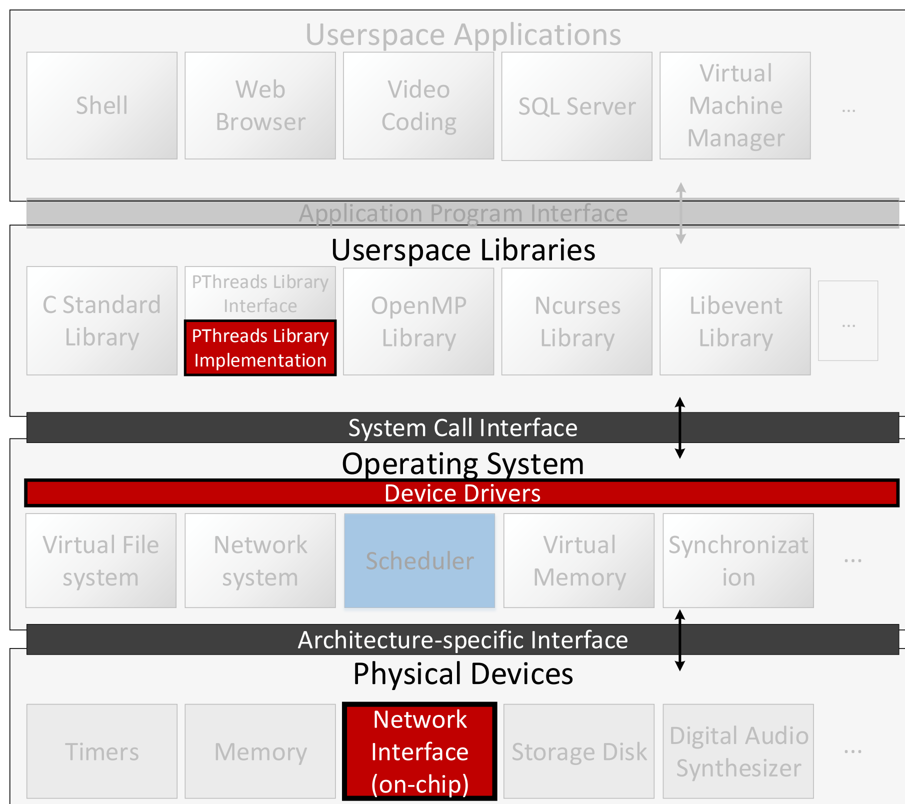


Figure 4.1 – The Subutai solution; Subutai components are highlighted in red.

Subutai is comprised of three elements: a user space library, a kernel space driver, and a hardware module. The user space library mimics an existing synchronization solution intended for parallel applications. Hence, our library has the same procedure signatures as the existing library; yet, it has different procedures implementation. The ability to mimic existing

synchronization libraries is the key feature of Subutai: the acceleration of legacy parallel applications. Subutai also supports novel applications that employ the same library. On the data synchronization discussion of Chapter 2, we observed that shifting an application from one synchronization solution to another requires refactoring the source code. Unfortunately, the refactor is not as simple as changing the names of procedure calls: for OpenMP, for instance, a specific model of parallel execution is enforced. Therefore, the developer may be forced to redesign the entire parallel algorithm. The process of application modification is costly in terms of development time and investment, and software already is the highest investment cost of new products (Figure 1.6).

For this work, the PThreads standard was chosen as the user space library to be replaced. As discussed previously in Section 3.3.1, the PThreads standard can be used as the synchronization mechanism or the underlying structure for other solutions; hence, it has the highest potential impact on legacy software.

Besides the user space library, each core has a new hardware module responsible for accelerating synchronization operations. This component, called Subutai-HW, is a state machine coupled with a small dedicated memory. Subutai-HW and the target architecture are described in Section 4.2 and Section 4.1 respectively.

Similar to the other synchronization libraries, once the user calls a procedure, the library employs services from the kernel through a system call. System calls provide the link between the hardware and software parts of the Subutai solution. Thus, the hardware protocol is abstracted from the user space library. Section 4.3 details the software part of our solution.

4.1 Target Architecture

Figure 4.2 shows a schematic of the target architecture. Each core communicates with instruction and data caches and a local NI. An instance of the OS is created for each core as well. The router for interprocessor communication uses a standard design with buffers, a crossbar switch, and a switch allocator.

Historically, performance gains in user applications have been obtained through the advances in hardware engineering, which required little or no change to the application code. Unfortunately, the dwindling of Moore's Law and the realization that clock frequency cannot be scaled indefinitely because of power constraints have resulted in a shift to parallelism on CPU design. Modern multiprocessors now consist of double digits of processing core units [HdM16] [EBSA+11]. Therefore, we target a manycore architecture composed of 64 processing cores.

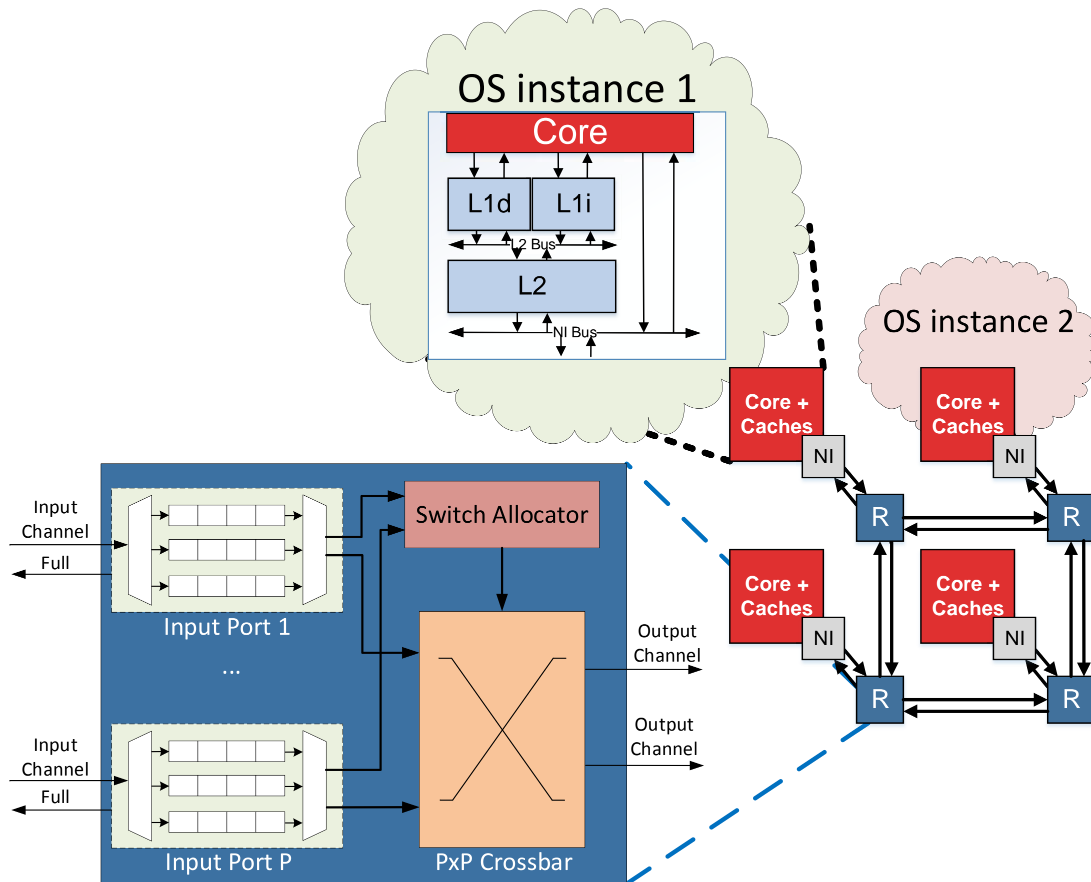


Figure 4.2 – Schematic of the target architecture for Subutai; the target architecture is comprised of 64 cores.

The Level 1 cache is private and split into instruction and data caches. The Level 2 cache is shared among all cores, and banks are distributed on the system. We explore synchronization solutions for Symmetric Multiprocessing (SMP) because it facilitates the development of parallel applications as the developer does not need to concern itself with data placement [PH13]. Hence, cache coherence is required and used.

We employ a decentralized approach to the OS where each core has its self-governing OS. When information is required to be shared at the OS level, we use replication instead of sharing as to decrease contention. For dozens or more cores, message passing can be much faster than memory sharing [BBD+09]. The decentralized OS design enables the scheduler to be decentralized as well. A decentralized scheduler can provide a faster thread switching, which is important for multithreaded parallel applications.

The interprocessor communication system uses a packet-based Network-on-Chip (NoC), which provides a more efficient on-chip communication when compared to traditional solutions as a shared bus for double digits multiprocessing systems [BM02]. Physically, distributing router units reduces the wire delays and the capacitance of the interconnection. Architecturally, decentralizing the interconnect fabric enables reliable systems building through independent operations.

4.2 Subutai Hardware (Subutai-HW)

Subutai-HW extends the NI architecture for handling synchronization primitives. Figure 4.3 shows the schematic representation of Subutai-HW and its location on the target architecture. The main components of Subutai-HW 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 hardware architecture by RTL simulation and synthesis. In addition, we also developed an analytical model to demonstrate its operation latencies.

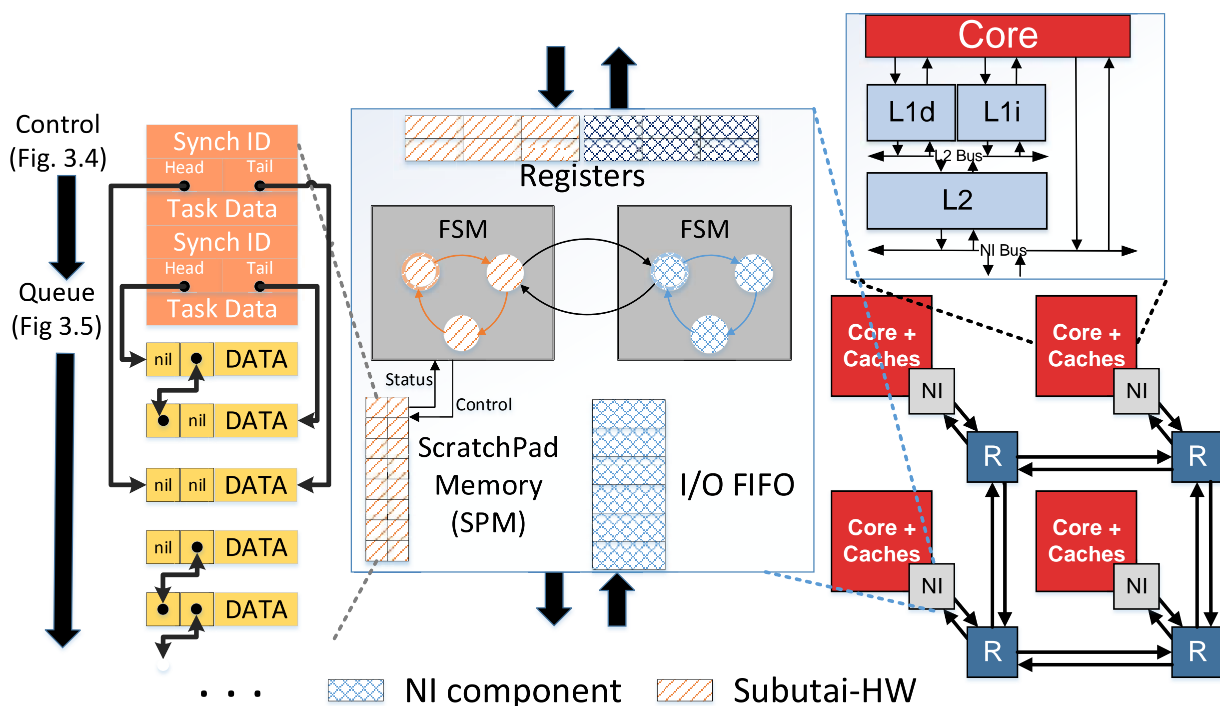


Figure 4.3 – Schematic representation of Subutai-HW and the NI. The hardware elements required by Subutai-HW implementation are highlighted in orange.

Subutai-HW employs double-linked queues to record events, as shown on the left-hand side of Figure 4.3. As an alternative to a garbage collector, the double-linked queues allow Subutai-HW to consume memory on demand. Besides, condition variables are dealt more efficiently with such structure, as it avoids the thundering herd problem [Lin19a]. The queue manipulation is based on the futex implementation of the Linux kernel, which is explained in Section 4.3.

Subutai-HW operates using two record information structures. The first one, shown in Figure 4.4, records the synchronization primitives' metadata. The first 32-bit field is the only one known by software and is employed as a unique Identification (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 is used as

a pointer to itself; we 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 (Figure 4.5). Finally, the last 32-bit field records values used for some of the primitives. The first 16-bit 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 furthest 16-bit is applied only for the barrier primitive to record the maximum number of threads allowed in a barrier.

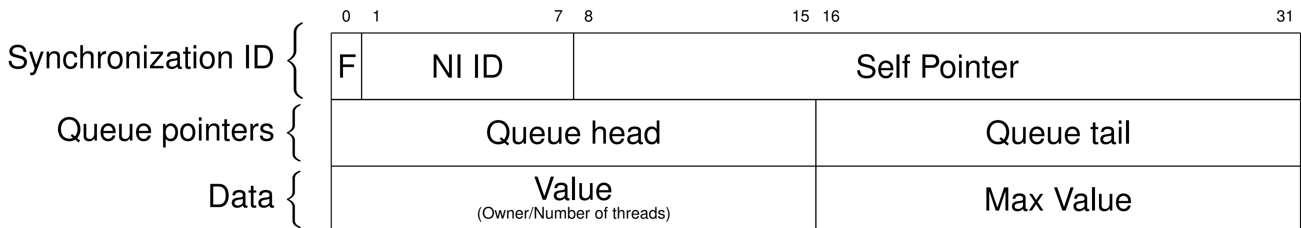


Figure 4.4 – Subutai-HW control structure.

Figure 4.5 shows an entry to the double-linked queue composed of six fields. The first bit is employed to allocate/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 17th 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.

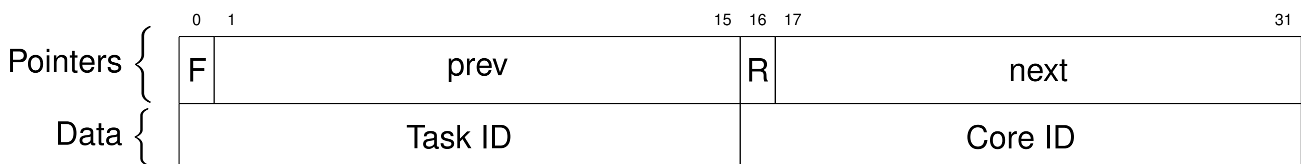


Figure 4.5 – Subutai-HW queue structure.

The bare minimum memory requirement for the SPM is one control entry and 63 queue entries, regarding a target 64 core architecture. Since we have to record up to $p - 1$ cores, the minimum SPM size is $\frac{1 \times 96 + 63 \times 64}{8}$ = 516 bytes. Note that Subutai-HW is incorporated into every NI; consequently, we handle up to 64 primitives even with minimum sizing. For our target architecture, we use an SPM of 1 KiB (4 control entries and 122 queue entries) that handle up to 256 primitives in hardware. 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 ($< 2 \times 63$) – since the worst-case scenario is 63 queues per entry, as explained earlier[1]. Thus, a static solution would be either too limited or a waste of memory resources.

[1]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.

The essential queue procedures and their latencies are shown in Table 4.1. These procedures are the foundation for the synchronization operations; in general, either some operation will insert or remove an entry from the queue. Thus, the queue procedure latencies will have a significant impact on the latencies of the states that represent the synchronization operations.

Table 4.1 – Latency of essential queue procedures. m = memory latency.

Queue procedure	Best response time	Worst response time
pop_free_queue	$2m$	$4m$
pop_synch_queue	$3m$	$7m$
push_synch_queue	pop_free_queue+$3m = 5m$	pop_free_queue+$6m = 10m$
cat_queue	$3m$	$7m$

The first procedure of Table 4.1, pop_free_queue, is for obtaining a free entry. A Subutai-HW register controls the head and tail of the queue that controls all free entries named free queue. For the fastest scenario, two operations are required: (i) fetch the prev field from memory for this entry; and (ii) write nil to the prev and next fields with a single memory operation. Then, a check is made with the fetched prev field: if it is empty, it means the free queue is empty, and no more operations are required. Otherwise, two more operations are required: (i) fetch the pointers for the previous entry; and (ii) write nil to the next field for such entry, thus, marking it the new tail of the queue. For both cases (i.e., empty free queue or not), the Subutai-HW register is updated with the new tail information.

The second procedure (pop_synch_queue) dequeues an entry from the queue of a synchronization operation. There are two differences with this procedure compared to pop_free_queue: (i) the head and tail of the queue are kept on memory instead of a register; and (ii) entries are removed from the head instead of the tail. Hence, more memory operations are required. In addition, the dequeued entry is added to the free queue. The push_synch_queue is the opposite of pop_synch_queue: it enqueues an entry to the tail of the synchronization queue. It is even more expansive since it needs first to obtain an entry from the free queue. It does this by relying on the pop_free_queue procedure.

The last queue operation is cat_queue, which is responsible for concatenating two synchronization queues. This procedure is used exclusively for the condition synchronization. The first queue is added to the tail of the second queue. Only three operations are required for the best response scenario: (i) fetch pointers for the second queue; (ii) if the queue is empty, rewrite them with the first queue head and tail pointers in one memory operation; and (iii) write nil to the head and tail pointers of the first queue. Otherwise, seven memory operations are required.

Two queue procedures (pop_free_queue and pop_synch_queue) change the Subutai-HW register that controls the free queue on their last operation. We did not count this as an

additional latency cycle as we made sure that no operation that directly follows this procedure will access the register.

Table 4.2 shows the latencies of the states as dependent on the Subutai-HW cycle c, SPM 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 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 packets before we have finalized processing the requests. Additionally, some states (e.g., Deallocation) do not need to generate packets at all.

Table 4.2 – Latencies of Subutai-HW 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	$4m + 1c$	$(n \times 1m) + 3m + 1c$	$(n \times 1m) + 1m + 1c$
Deallocation	$3m$	$3m$	None
Mutex Lock	$2m + 1c$	$11m$	$2m + 1c$
Mutex Unlock	$2m$	$10m + 1c$	$2m + 1c$
Barrier Wait	$7m$	$(1m + 1c) + \rho \times (11m + 3c)$	$(1m + 1c) + (12m + 4c) + (23m + 7c) \ldots = (1m + 1c) + \rho \times (11m + 3c)$
Condition Wait	$5m + 1c$ + Mutex Unlock	$10m + 1c$ + Mutex Unlock	None
Condition Broadcast	$1m$	$18m + 1c$	$11m + 1c$
Condition Signal	$1m$	$29m + 2c$	$11m + 1c$

To illustrate the best and worst response times of Table 4.2, we describe the Mutex Lock state, which is responsible for modeling the `pthread_mutex_lock` operation. The fastest scenario, whose latency is $2m + 1c$, happens when the mutex is unlocked. It requires two memory operations: (i) fetch the control structure (field "Value" from Figure 4.4) 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 the state deals with the queue. 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; then, it writes the requesting thread information into it and the tail information in the primitive metadata by calling the `push_synch_queue` (6 more memory operations), performing 11 memory operations in total. The latency for the other states follows a similar procedure.

Table 4.3 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 uniform access of 2 cycles, 4 control structures, and 122 queue entries. We also considered a 4 ns for an *FSMentry* (3 cycles for 3 flits of 32 bits and 1 cycle to decide the next state) and 1 ns for an *FSMexit* (1 cycle to set a flag) to reach any state.

Table 4.3 – Latency of Subutai-HW states with parameters $c = 1ns$, $m = 2ns$, $n = 4$, $\rho = 63$, $FSMentry = 4ns$, and $FSMexit = 1ns$.

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 nanoseconds. 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 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: $FSMentry + FSMexit + 1m + 1c + \rho \times (11m + 3c)$.

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, the assumption 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 one thread is truly released since a mutex is associated with it. Therefore, the broadcast state avoids the scenario previously described for the barrier state – only the owner of the mutex will be released.

In addition to the FSM, Subutai-HW also includes six 32-bit and three 1-bit registers; three are used for the packet fields (Figure 4.6), and six more to (i) handle the free queue entry list; (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 in the two control structures (Figure 4.4 and Figure 4.5) to handle any request. Area consumption will be presented and discussed in Section 6.3.1.

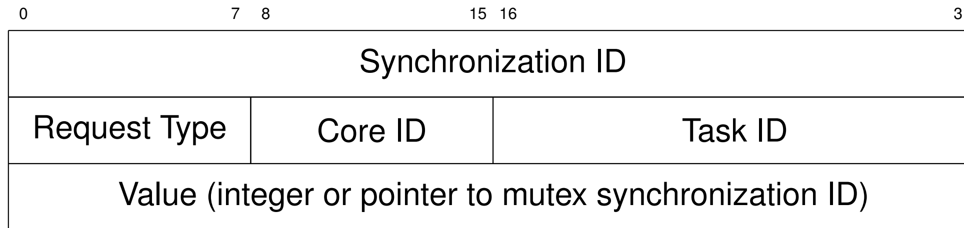


Figure 4.6 – Subutai's packet format.

4.2.1 Subutai-HW RTL Implementation and Verification

Subutai-HW was first constructed as a pseudo-code implementation. The implementation is available as Appendix A. From the beginning, we designed the hardware to handle double-linked queues and to access a private memory area.

The double-linked queue manipulation is the most complex structure used in our hardware design. The following operations are possible with them: (i) They can be traversed from head to tail and tail to head; (ii) two queues can be merged without the need to traverse any of them; this makes them very fast for merging operations; and (iii) an element can be removed regardless of its position on the queue. Due to its complexity, we developed a C tester to verify that our implementation (Appendix A) meets the double-linked queue specification. We use as the baseline implementation of a double-linked queue the `TAILQ_-` macros provided by `queue.h`. This header is found on both Linux and BSD systems, and it has been in use since 1994.

Some of the procedures of Subutai-HW are essentially wrappers to the queue operations that ensure the queue maintains the expected interface by Subutai-HW (e.g., control bits from Figure 4.5). Consequently, the following procedures have also been verified: `pop_free_queue`, `push_synch_queue`, `push_synch_queue_checked`, and `cat_queue`.

Then, we developed the RTL version of the same hardware, using the VHDL description language, and verified it using the testbench methodology [RPS00]. For improving readability and decreasing complexity, most of the operations needed by the Subutai-HW FSM have been developed as procedures [Gai04]. The following packages have been designed:

1. **Constants**: define (i) multiple constant values used for Subutai-HW (e.g., memory size area, nil pointer); (ii) basic procedures to manipulate pointers (e.g., clear free bit); and (iii) transformation procedures from datatypes to string as to facilitate debugging.

2. **Memory operations**: define procedures to read or write memory position on the SPM.

3. **Queue operations**: define essential procedures to enqueue or dequeue elements from a given queue (e.g., `pop_free_queue`, `push_synch_queue`).

There is a dependency of these packages in the order they are presented; thus, the memory operations package requires the constant package. The queue operations package requires both memory operations and constant packages.

The FSM of Subutai uses all these packages to achieve its functionality. It is responsible for (i) sending and receiving packets, (ii) controlling access to the internal structure of the hardware; and (iii) providing the interface of Subutai-HW for the system. All synchronization operations supported by the hardware have been verified.

4.3 Subutai Software

Software-wise, Subutai reimplements an existing parallel library and a kernel driver. We chose to reimplement the PThreads library as it can be employed by itself and as a backbone for other synchronization solutions. The kernel driver is a typical driver that communicates with a peripheral through Input/Output (I/O) operations and Interrupt Requests (IRQs).

From all the capabilities of the PThreads library, we reimplemented the default mutex, barrier, and condition variables. PThreads allows optional attributes to be defined through procedure calls. The current Subutai Software version does not support this, although the support is possible. Conversely, Subutai supports thread join, create, and exit operations as they are essential to the library.

The family of mutex procedures illustrates our implementation; the functionality of these procedures has been described in Section 3.1.1. We use the GNU LibC as the software reference as it is widely employed in Linux distributions (the reference GNU LibC version used in this work is 2.26).

4.3.1 User space PThreads Library

The structure of the PThreads mutex, called `pthread_mutex_t`, is defined on files `thread-shared-types.h` and `pthreadtypes.h` and shown in Listing 4.1. The size of the structure is architecture-dependent; for an x86-64 machine, running on 64-bit mode, its size is 40 bytes. Lines 2-23 of Listing 4.1 shows the variables employed for mutex handling – line 2 is the actual lock, comprised of a single integer; line 3 is used for recursive locks; line 4 is the thread identification and so on. Line 2 is the basic requirement for the mutex operations – all other variables are used for optional attributes and debugging [FRK02]. David Wragg [Wra19] shows a PThreads implementation, called skinny-mutex, comprised of this single field.

Listing 4.1 – Definition of `pthread_mutex_t`.

```
1  struct __pthread_mutex_s {
2          int __lock __LOCK_ALIGNMENT;
3          unsigned int __count;
4          int __owner;
5  #if __WORDSIZE == 64
6          unsigned int __nusers;
7  #endif
8          /* KIND must stay at this position in the structure to maintain
9             binary compatibility with static initializers.  */
10         int __kind;
11         __PTHREAD_COMPAT_PADDING_MID
12 #if __WORDSIZE == 64
13         __PTHREAD_SPINS_DATA;
14         __pthread_list_t __list;
15 # define __PTHREAD_MUTEX_HAVE_PREV      1
16 #else
17         unsigned int __nusers;
18         __extension__ union {
19                 __PTHREAD_SPINS_DATA;
20                 __pthread_slist_t __list;
21         };
22 #endif
23         __PTHREAD_COMPAT_PADDING_END
24 };
25
26 typedef union {
27         struct __pthread_mutex_s __data;
28         char __size[__SIZEOF_PTHREAD_MUTEX_T];
29         long int __align;
30 } pthread_mutex_t;
```

It is important to note that developers utilizing the PThreads library only operate on opaque pointers. Hence, the actual implementation of the structure is abstracted away. The use of opaque pointers makes PThreads malleable to different implementations.

A request for locking is received on the procedure called `pthread_mutex_lock`. After permission and attribute checking is done, another procedure is called to actually lock the mutex. This second procedure is architecture-dependent. Listing 4.2 shows the SPARC implementation of `__lll_lock_wait` (i.e., the procedure responsible for locking). The pointer to the `futex` variable is the lock shown in line 2 of Listing 4.1, and the `private` variable is the attribute that decides if other processes can access the mutex. Two atomic compare-and-exchange operations are executed on lines 6 and 10 – this is an attempt to lock the mutex; if

the first one is successful, line 8 is skipped. The code is certainly all but obvious, but it solves livelocks conditions explained by Drepper [Dre19].

Listing 4.2 – SPARC locking procedure.

```
1  void
2  __lll_lock_wait(int *futex, int private)
3  {
4          int oldval;
5          do {
6                  oldval = atomic_compare_and_exchange_val_24_acq(futex, 2, 1);
7                  if (oldval != 0)
8                          lll_futex_wait(futex, 2, private);
9          }
10         while (atomic_compare_and_exchange_val_24_acq(futex, 2, 0) != 0);
11 }
```

4.3.2 Kernel Space Futex

Fast user space Mutexes (futex) is a lightweight kernel-assisted locking primitive for user space applications. It provides a fast solution for uncontended lock acquisition and release operations. The mutex state is stored in user space (an integer value). No system call overhead is needed when the mutex is uncontended; atomic operations are enough. For low contention locks, the system call overhead can be significant [FRK02]; in the contended case, the kernel is invoked to perform sleep and wake procedures [Har19], which is the behavior shown in Listing 4.2 – only when the lock is contended, the system call is called (line 8).

Internally, futex uses wait queues to record threads waiting for a lock event. Although futex is built for locking operations, it can be used as a backbone for other synchronization primitives as conditions and barriers [Dre19]. This is precisely the case for GNU LibC; BSD systems have a similar system call called `_umtx_op` [Fre19b]. Listing 4.3 shows a futex queue entry for Linux Kernel version 5.1.9. Line 2 is a priority double-linked list. Line 4 identifies the thread waiting for a futex event. Lines 5 and 6 are the lock and key used for a hash table, respectively. The other variables are optional attributes.

Figure 4.7 presents the relationship between user space and kernel space. The user calls a system call with its mutex state (`uaddr`), and the kernel creates a `futex_q` structure and computes a futex key based on the mutex state. Then, it uses the key to store the `futex_q` on a specific bucket of the hash table. Consequently, there is one `futex_q` for each

Listing 4.3 – Kernel space futex queue (kernel/futex.c).

```
1  struct futex_q {
2          struct plist_node list;
3
4          struct task_struct *task;
5          spinlock_t *lock_ptr;
6          union futex_key key;
7          struct futex_pi_state *pi_state;
8          struct rt_mutex_waiter *rt_waiter;
9          union futex_key *requeue_pi_key;
10         u32 bitset;
11 } __randomize_layout;
```

task waiting for an event and possibly many futex_q per futex. Besides, the same bucket can be shared by different futexes, as shown in Figure 4.7 [Har19].

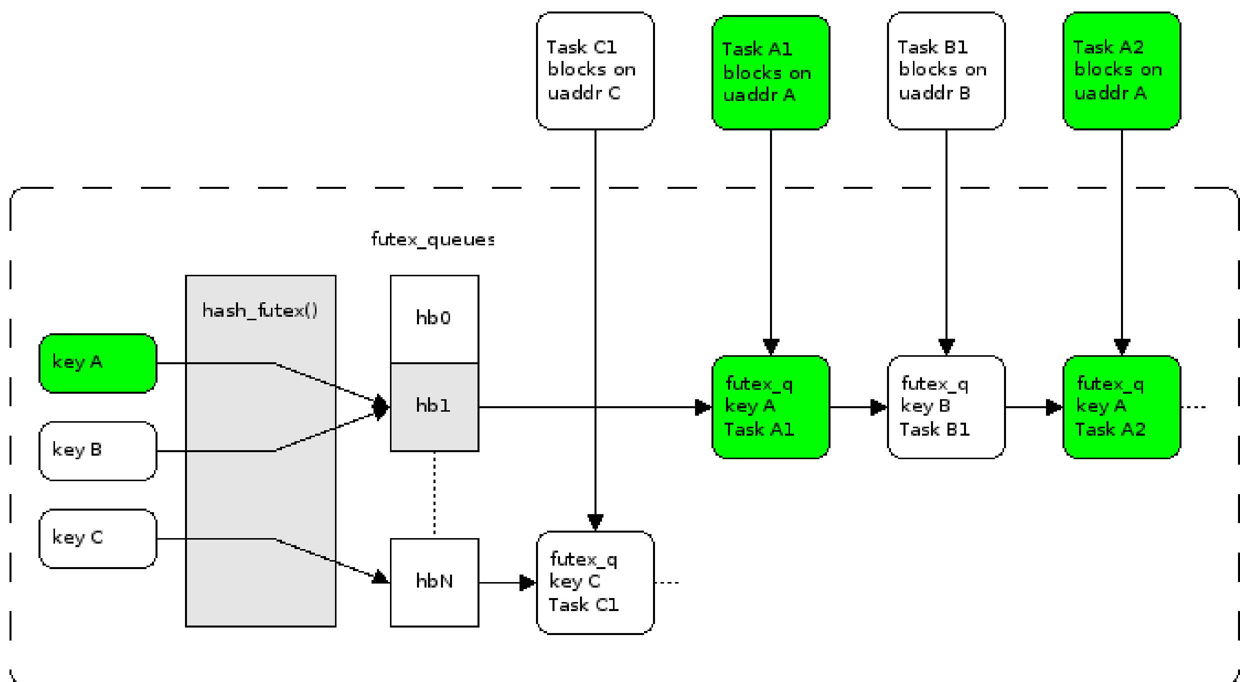


Figure 4.7 – Futex kernel implementation [Har19].

4.3.3 Subutai Implementation

The new software implementation is much simpler than the glibc implementation as the processing is offloaded to Subutai-HW. Consequently, the synchronization library is transformed to act as the link between the user application and the hardware. The

pthread_mutex_t structure that records the mutex's metadata is reduced to a single value: the synchronization ID. This value is the only information that cannot be computed solely on software to generate packets; thus, it is recorded in the structure shown in Listing 4.4. The implementation represents a reduction from 40 bytes (Listing 4.1) to 4 bytes. Additionally, the same reduction can be applied to conditions and barriers, as they are handled in Subutai-HW as well. The cache space gains are summarized in Table 4.4.

Listing 4.4 – The pthread_mutex_t employed by the Subutai Solution.

```
1 #include <stdint.h>
2
3 #define __PTHREAD_NI_DST(val)   (((val) >> 24) & 0x7F)
4 typedef struct {
5         uint32_t        __synch_id;
6 } pthread_mutex_t;
```

Table 4.4 – Cache space reduction of synchronization primitives.

Synchronization Primitive	GLibC x86-64 (bytes)	Subutai (bytes)	Reduction (Percentage)
mutex	40	4	90%
barrier	32	4	87.5%
condition	48	4	91.7%

We have achieved a reduction of 90% of cache usage, approximately, for synchronization primitives. Note that information has changed location from the cache to the SPM controlled by Subutai-HW. Therefore, valuable cache space is freed up to the application and the OS to use. Also, the SPM is not a shared memory, which further reduces resource utilization, such as the interconnect for cache coherence communication. The current version of Subutai is limited to the standard attributes of these primitives, but future releases that support additional features should incorporate these features into the hardware-side. Therefore, the reduction of cache usage still would hold.

Once a synchronization procedure is called, the library provides the link to Subutai-HW. For simplicity, the OS does not have access directly to Subutai-HW: the link is provided through the NI. Hence, both OS and library can reuse existing procedures for NI communication. Listing 4.5 shows a simplified implementation of a mutex procedure for a Linux kernel driver. We use the Linux kernel API as an example because it is one of the most well-known interfaces. A kernel API differs considerably among kernels.

Line 12 of Listing 4.5 generates a Subutai-HW request from the user-supplied synchronization ID and request type. The request type is inferred by the procedure call (e.g.,

Listing 4.5 – Simplified Subutai driver implementation.

```
1  #include <linux/compiler.h>
2  #include <linux/net_device.h>
3  #include <linux/skbuff.h>
4
5  int
6  ni_mutex_lock(struct net_device *dev, void __user *u_data)
7  {
8          int     ret;
9          struct  sk_buff  *skb;
10         struct  ni_priv  *priv = dev->priv;
11
12         skb = __ni_gen_pkt(priv, u_data, HW_REQ_MUTEX_LOCK);
13         __ni_add_skb(priv, skb);
14         ret = ni_hw_tx(skb, dev);
15         if (ret < 0)
16                 goto end;
17
18         /* put itself on a wait queue */
19         ret = __ni_sleep_on(skb, priv);
20         /**
21           * wakes up once response packet has arrived or interrupted by
22           * another signal
23          **/
24 end:
25         __ni_rm_skb(priv, skb);
26         return (ret);
27 }
```

`ni_mutex_lock` for mutex locking, `ni_barr_wait` for waiting on a barrier). Additionally, the NI address is derived from the synchronization ID (line 3 of Listing 4.4) and the `ni_priv` internal structure. The generated packet is stored in a `sk_buff` structure and recorded internally to be freed at a later point (lines 13 and 25). The actual transmission is done on line 14 and reuses the NI transmission procedure. Lines 15 and 16 check and interrupt the procedure execution if an error has occurred. Line 19 makes the current thread sleep waiting for the mutex to be owned by it. Finally, the thread is woken up when such an event occurs and returns to the library on line 26.

Initial experiments using benchmarks showed that line 19 of Listing 4.5 was problematic, as the thread sleeps unconditionally for the response packet. However, sleeping/waking up threads are expansive operations. If the mutex is unlocked, most of the latency is consumed by the sleep/wake procedure. Hence, we need a more efficient mechanism to use the hardware-accelerated operations.

Thus, we employ a predictor to infer probabilistically the mutex's state employing the 2-bit saturation counter used for branch predictor as it is straightforward and can be extended to handle more bits for accuracy. Figure 4.8 depicts the state machine of the 2-bit predictor. For branch prediction, every time a branch happens, the state machine is consulted. If the state is either in strongly or weakly not taken, the core assumes the branch will not happen; the reverse happens for either strongly or weakly taken states. After the branch is evaluated, the procedure works as follows: branches evaluated as not taken decrement the state toward strongly not taken; and branches evaluated as taken increment the state toward strongly taken. Therefore, the predictor is continually updated with new branch information.

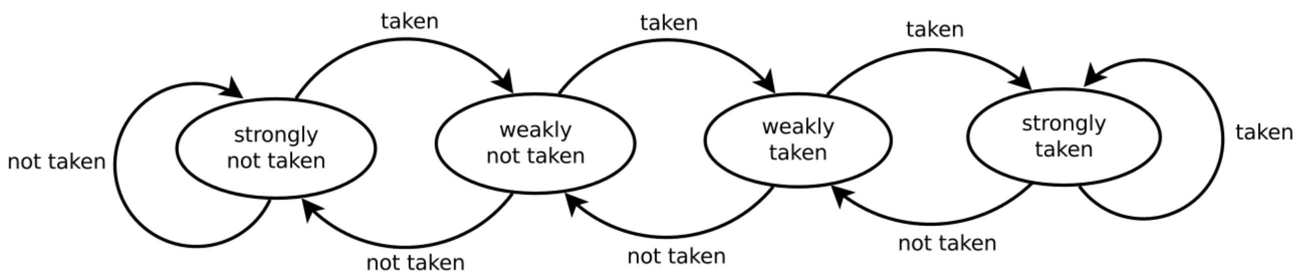


Figure 4.8 – 2-bit saturating counter for branch prediction [Dia19].

Our scenario uses the predictor to decide if the driver sleeps unconditionally (not taken states) or not (taken states). When the driver does not sleep unconditionally, a delay is added to wait for the response packet. In other words, the core spins for a configured amount of time and only sleeps if no response packet has arrived. Linux has an example of such API called `ndelay` that delays execution for at least the number of nanoseconds provided by its parameter. Linux calibrates the number of loops required to delay using BogoMips [Lov10] for each core.

In sum, Figure 4.9 depicts the communication flow from the user application to Subutai-HW and vice-versa. First, the application makes a PThreads interface request; the Subutai-enhanced PThreads library identifies the synchronization ID for this primitive and passes it on to the driver, along with the interface request (e.g., mutex lock). Then, the driver writes to data and control registers of the NI to send a packet and to flag a new request, respectively. Then, the driver 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 Figure 4.9. One when there is no response packet and no backward procedure; thus, the driver returns immediately after writing to control registers. The other one happens when the driver accesses the local Subutai-HW. The same procedure is followed, but without injecting packets into the NoC.

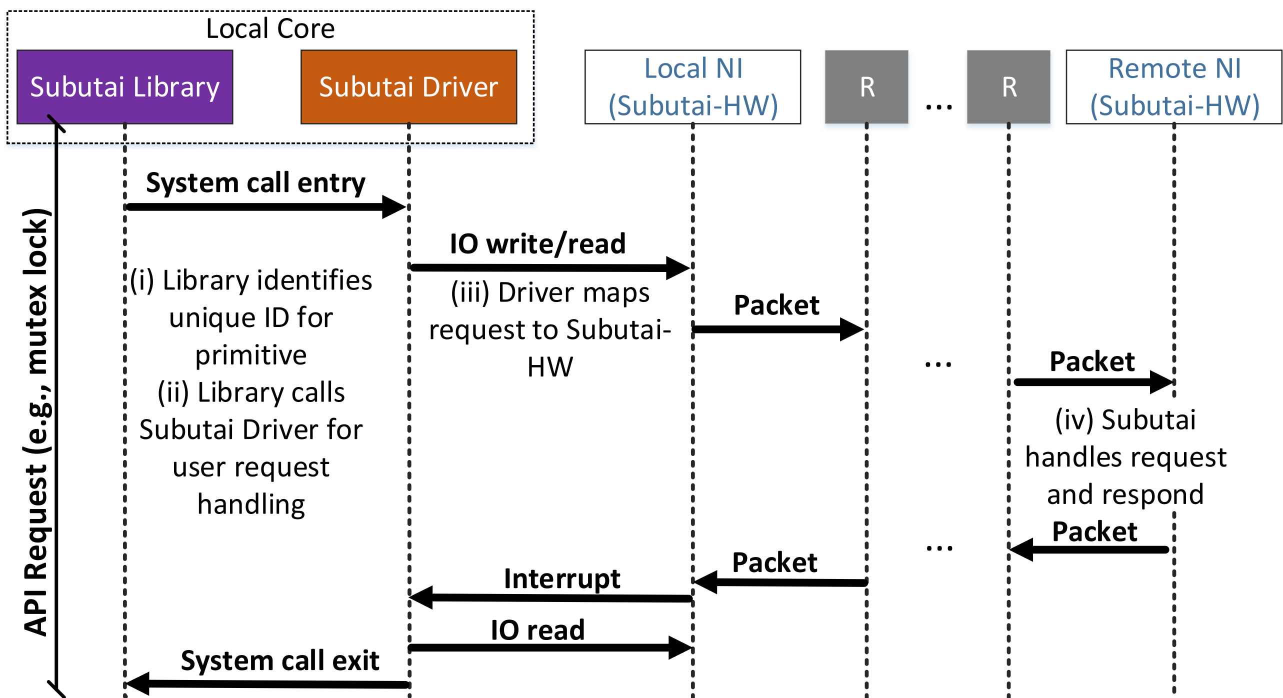


Figure 4.9 – Communication flow of Subutai.

5. SUBUTAI EXTENSIONS

Trabalhe com o que você gosta e nunca mais goste de nada.

Fernando Grando on life lessons

This chapter proposes extensions built on top of the essential components of Subutai, namely the user space library, Subutai-SW, and Subutai-HW. We propose two extensions to Subutai for accelerating some scenarios while increasing the cost of adoption. The extensions diverge from the essential components of Subutai, as Subutai will continue to work with the absence of the former, while it will not work with the absence of the latter. Subutai extensions work functionally as filesystem extended attributes. Ext4 [Ext19], for instance, provides extended attributes to increase the capability of the filesystem, in terms of size capability, security, and other attributes. For this work, we focused on performance benefit (i.e., a decrease of execution runtime) for Subutai extensions.

Figure 5.1 highlights in blue the components of the system that need modification for the Subutai extensions. We propose two Subutai extensions: one for the schedule and another for the PThreads library.

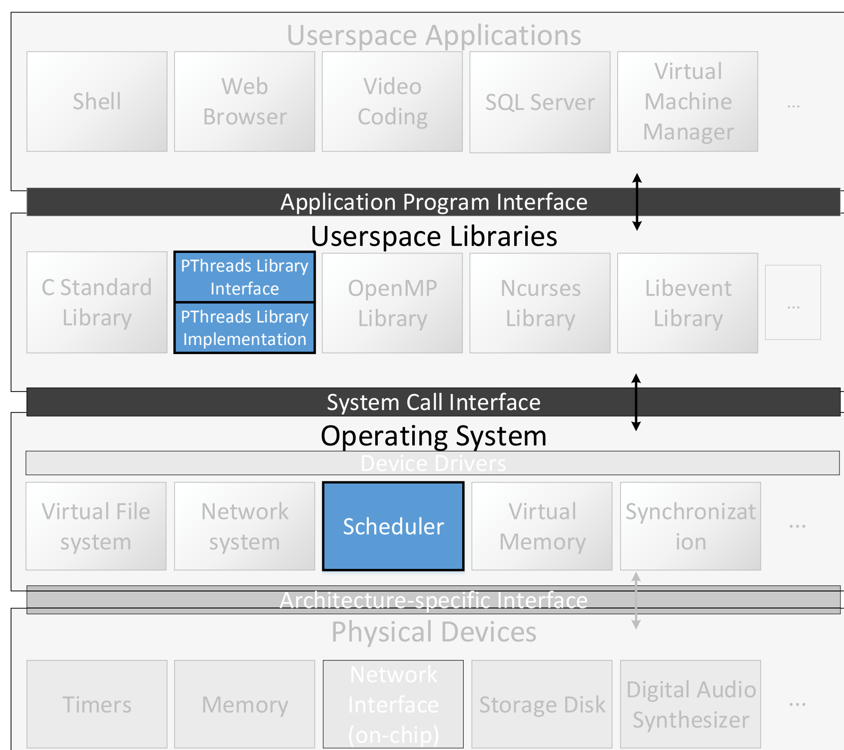


Figure 5.1 – The Subutai extensions are highlighted in blue.

Firstly, we propose a scheduling policy for accelerating the execution of multiple parallel applications running concurrently. The policy can be applied to varied scheduler

techniques and will be described in Section 5.1. Secondly, we propose to reimplement the PThreads conditions, called neocondition, for avoiding the use of mutual exclusion policy. However, such reimplementation needs to change the interface of PThreads; thus, it is restricted to source-code compatibility only (instead of binary compatibility as the rest of Subutai). Since neocondition is an extension of our solution, this does not change the binary-compatibility of Subutai. The neocondition synchronization is described in Section 5.2.3.

5.1 Critical-Section Aware (CSA) Scheduling Policy

5.1.1 Motivation

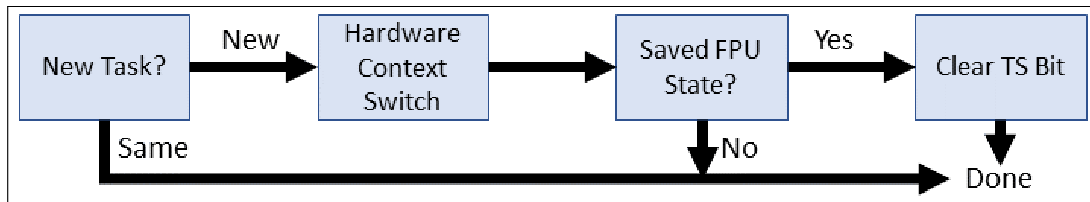
Until the rise of multicore architectures, OS designers considered scheduling to be a solved problem [LLF+16]. However, architectural and application changes pressured the scheduler to work with modern hardware, such as non-uniform access to memory, cache coherency, and diverse set of application models. The increase of scheduling complexity can be visually observable and is shown in Figure 5.2[1]. The figure is restricted to the context switch process provided by the scheduler. Initially, context switching was done at the hardware-level, and the only software optimization present was related to the Floating-Point Unit (FPU) state. More than twenty years later, the context switching is mostly done at the software-level and includes many features besides optimizations: security concerns (stack canary, retpoline, I/O permissions), debugging (debug registers), virtualization (hypervisor, Xen), and thread handling.

Lozi et al. [LLF+16] show that unbalance scheduler work distribution can significantly degradation overall system performance. They identify a series of bugs on the scheduling of Non-Uniform Memory Access (NUMA) machines. Resolving such bugs, they achieved a speedup ranging from 4$\times$ up to 137.59$\times$ running the NAS applications benchmark.

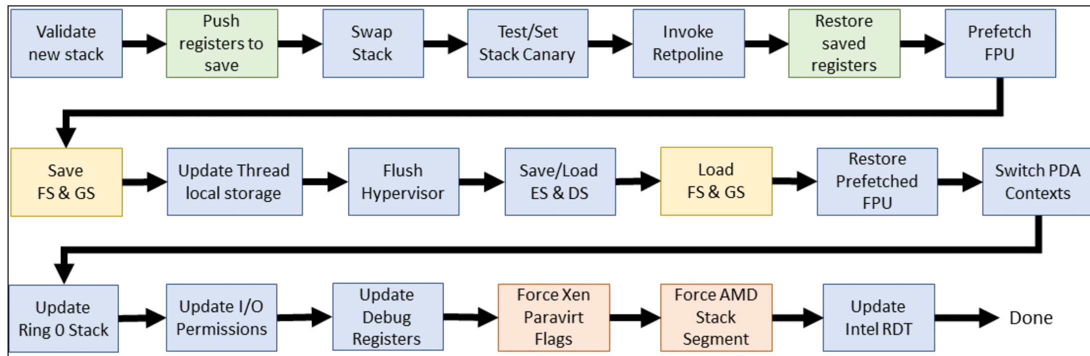
Our motivation is thus twofold: (i) the Subutai solution speeds up individual applications by accelerating their synchronization primitives usage. As we target legacy code, we are unable to change the use of such primitives. Therefore, we target the scheduler policies as it does not require the modification of the application code. Ergo, we intend to further speed up applications by aggregating multiple parallel applications with a critical section-aware policy; (ii) as will be shown in Section 5.1.3, certain scheduling policies increase the critical section of parallel applications – a major factor in the scalability of such applications.

We do not propose a new scheduler design; instead, we provide a policy and its performance impact on parallel applications for (i) ignoring and (ii) accelerating the

[1] Linux 0.11 was exclusively written for x86, while Linux 4.14 supports multiple architectures. Thus, the comparison is made for x86 only.



(a) Linux 0.11 core scheduler procedure (1991).



(b) Linux 4.14.67 core scheduler procedure for x86 (2018).

Figure 5.2 – Core scheduler procedure `switch_to` steps for two Linux kernel versions [Mai19].

critical sections of parallel code. Therefore, scheduler designs can be adapted to use this information. POSIX allows the application itself to determine its choice of scheduler policy through procedures calls [IEE16]; however, the system may deny this request.

Unfortunately, running multiple applications will inherently make every application slower (i.e., increased execution runtime), as before they had the exclusive right of the core[2], and now they must contend this resource. Nevertheless, the scheduling impact on execution runtime can be mitigated by the policies employed on the scheduler.

For parallel applications, a fair distribution of scheduler timeslots may be problematic. Parallel applications can be roughly divided into two execution modes: sequential and parallel. Every parallel application includes at least a small sequential part for initialization, such as thread creation and parsing of application parameters. Generally, the actual work of the application is parallelized. Yet, mutual exclusion data access is another sequential execution that is commonly used between 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), prohibiting the parallel execution of other threads. Consequently, delaying the critical section execution should be avoided to decrease the overall sequential time of an application.

[2]Excluding the idle thread.

5.1.2 Baseline Scheduler Design

We assume the round-robin (RR) algorithm as the baseline scheduler design. RR assigns timeslots for each process in equal portion and in circular order, without giving priority over any process. In addition, the RR scheduler avoids starvation by running the application set in a deterministic order. Thus, RR gives a fair[3] share of CPU time and produces low response time [EEG18]. Furthermore, we assume two restrictions: (i) every thread is considered as a process; and (ii) only one thread of each application is present in a given CPU. The latter limitations ensure that an application cannot receive improper higher priority by increasing the number of threads.

5.1.3 Application Example

To demonstrate the impact of scheduling on parallel applications we provide Figures 5.3 and 5.4. Those figures show the comparison of the critical section and sleep time spent on three application sets, respectively.

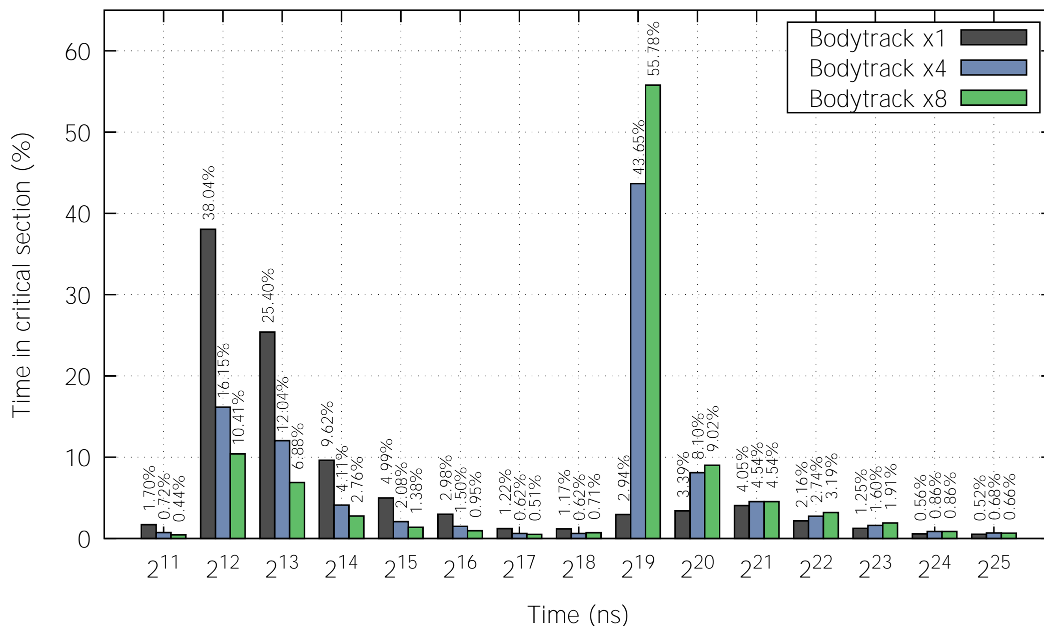


Figure 5.3 – Overall time spent in the critical sections for all the work-related mutexes of Bodytrack on a RR scheduler.

We compare a single application, Bodytrack, while it runs alone, with three others, and with seven other instances of Bodytrack (named, respectively, Bodytrack x1, Bodytrack

[3]In the next section, we will describe a metric capable of quantitatively compare the schedule fairness for a set of processes.

x4, and Bodytrack x8). The Y-axis is the percentage of overall spent time on a given time interval (X-axis). For Figure 5.3, the percentage refers to critical section latency; Figure 5.4 is the time spent waiting for a mutex to be available. For both figures, the X-axis comprises interval values in the form of [*RangeInit*, *RangeEnd*), where *RangeInit* is the $X-1$ value and *RangeEnd* is the X value for any given X value; for instance, the X value equals to 2^{12} is the time spent on a critical section for the interval $[2^{11}, 2^{12})$ ns.

As we compare the same application on these three scenarios, the number of times the application accesses the critical section is approximately the same[4]. On the other hand, the times spent per access (Figure 5.3) and waiting for a mutex (Figure 5.4) are not the same, as the scheduler can interrupt the application execution. Figure 5.3 shows that as the number of applications increases, the time spent per access also tends to increase since the scheduler does not differentiate execution on a sequential or parallel code. As was discussed previously, the sequential code should be run as fast as possible.

For some specific values, either the critical section latency or the sleep time may be higher percentage-wise for fewer instances of Bodytrack than with more instances Bodytrack (For instance, the X value equals to 2^{13} on both figures). Therefore, Figure 5.5 presents the overall time spent in critical sections for the three scenarios explored here. As expected, the sum of critical section latencies increases as more applications compete for core usage. Ergo, the time spent waiting on a mutex also increases with more applications.

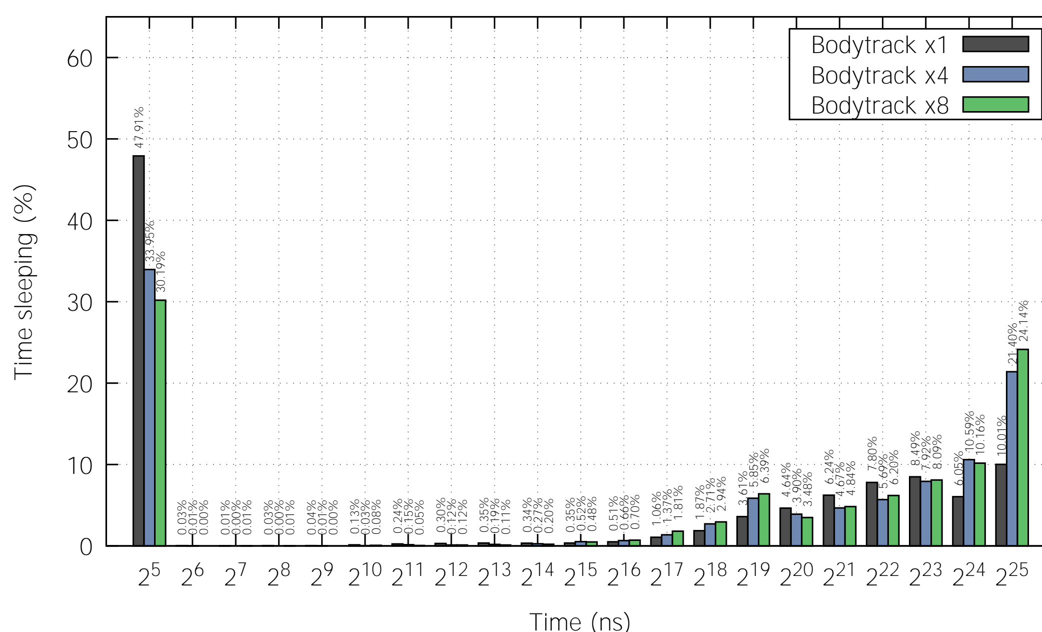


Figure 5.4 – Overall time spent in mutex queues for all the work-related mutexes of Bodytrack on a RR scheduler.

[4]Accesses to condition and, therefore, to the mutex associated with it, may vary according to the situation the threads were scheduled.

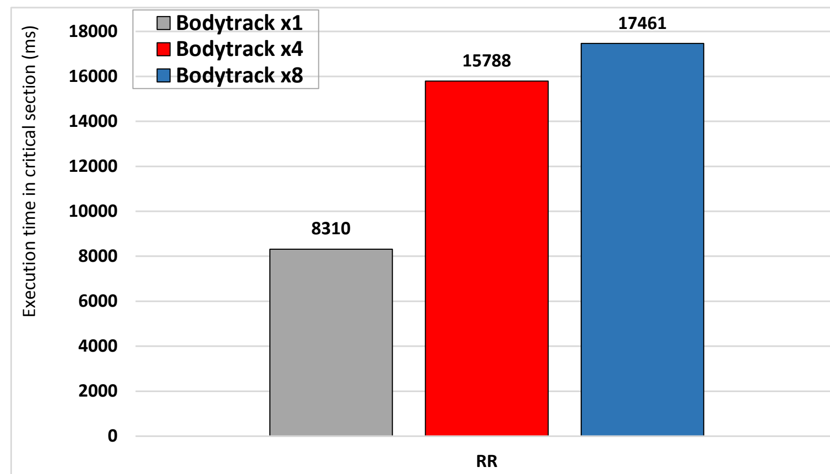


Figure 5.5 – Total execution time spent in critical sections for all the work-related mutexes of Bodytrack on a RR scheduler.

5.1.4 Design and Implementation Choices

Schedulers can be developed to prioritize some aspects of an application (e.g., CPU-bound, deadline, number of threads, and energy consumption). One type of such schedulers is the fair scheduler. A scheduler can be defined as 'fair' if equal-priority applications suffer the same slowdown due to the sharing of the system resources. The unfairness metric can be used to evaluate the fairness of the scheduler. The lower-is-better metric is defined as follows [GGSPM18]:

$$Unfairness = \frac{MAX(Slowdown_1, ... Slowdown_n)}{MIN(Slowdown_1, ... , Slowdown_n)} \quad (5.1)$$

Where n is the number of applications in the workload and $Slowdown_i = \frac{ET_{schedi}}{ET_{alonei}}$. ET_{schedi} denotes the execution time of application i under a given scheduler, and ET_{alonei} is the execution time of application i when running alone on the system. As discussed in Section 5.1.2, we employ RR as the baseline scheduler, although more complex policies can also be applied. The values obtained with the unfairness metric are discussed in Chapter 6.

The behavior of the application example of Section 5.1.3 shows that ignoring the nature of the critical sections of a parallel application results in an improper performance decline due to the sequential execution increase (i.e., critical section). Thus, we introduce the Critical-Section Aware (CSA) policy into the scheduler policies for executing critical section code as fast as possible.

The CSA policy works as follows – Every time a given thread has CSA enabled and is currently inside a critical section (i.e., holding a mutex), this thread has priority over the execution of all others that are not in the same scenario. In the case another thread also has CSA enabled and it is inside another critical section, a RR policy is applied to switch

between the two until either one finishes. Finally, if there are no threads that meet those requirements, a RR policy is applied to switch between the entire application set. A time limit is implemented in CSA to avoid deadlock and decrease the overall impact on the other threads that are executing on the scheduler concurrently. The limit is defined as:

$$CSALimit = (ThrReady + ThrRun - 1) * (2 * TS) \tag{5.2}$$

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 timeslot selected for the RR policy, generally in miiliseconds. For instance, the time limit of a thread that gains CSA priority, among 8 threads on the *ThrReady* and *ThrRun* states with a *TS* of 1ms, is 15ms.

This limit was chosen 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 $(ThrReady + ThrRun - 1) * TS$. Therefore, the schedule maintains its fairness characteristic as it will rollback to RR policy if the critical section would be too onerous.

Livelocking can be avoided by 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 [NO16].

When executing the same application set as Figure 5.5, but with the CSA policy enabled, the critical section execution time is kept as close as possible to the single application execution, as depicted in Figure 5.6.

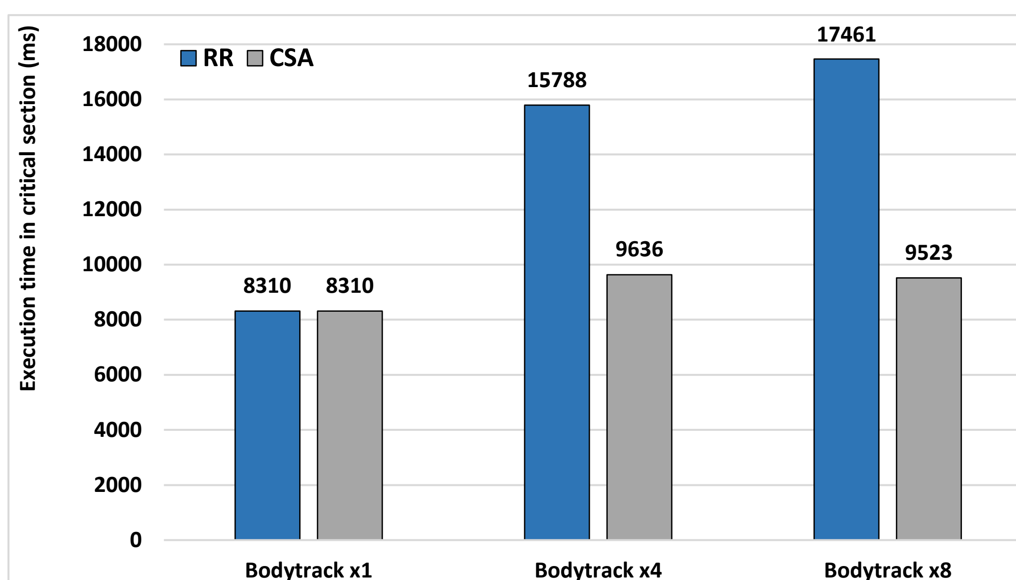


Figure 5.6 – Overall time spent in critical sections for all the work-related mutexes of Bodytrack on a RR and CSA-enabled scheduler.

Due to the restriction of *CSALimit* for fairness, as described in Equation 5.2, only a subset of critical sections is accelerated. This limit is the reason the critical section time

is lower with 8× applications than with 4×. Table 5.1 shows the impact of *CSALimit* on the Bodytrack application set. Approximately 10% and 8% of the total critical sections had CSA disabled as their time surpassed the *CSALimit* time for Bodytrack×4 and ×8, respectively. Even though we are analyzing the same Bodytrack, while running in a set of 4 and 8 applications, there are some discrepancies on the total number of requests for scheduling due to the use of synchronization primitives.

Table 5.1 – 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

5.2 Neocondition

5.2.1 Motivation

POSIX defines the condition synchronization as thus [IEE16]:

> *"A synchronization object which allows a thread to suspend execution, repeatedly, until some associated predicate becomes true. A thread whose execution is suspended on a condition variable is said to be blocked on the condition variable."*

The POSIX definition creates an association between a synchronization variable and a user-defined predicate. For example, predicates can be created for a FIFO to wait for the full and empty states. Thus, two condition variables would be created.

Listing 5.1 shows an example of a multi-threaded application that uses two conditions: one where a single element has been added to the queue; and another, where the queue has been entirely filled. These conditions are called `fi_cond_one` and `fi_cond_all`, respectively. For this example, we assume there are multiple consumers for a single producer. In other words, multiple threads may call either `fifo_wait_for_one` or `fifo_wait_for_full`, yet only one thread calls `fifo_inc_len`. Therefore, access to the shared variable `fi_len` can be done using atomic operations instead of mutual exclusion, which simplifies the code. Besides, such an access pattern to the queue is typical in master-slave applications.

Listing 5.1 – Example of two conditions and their associated predicates for a FIFO. Initialization and error-checking are not shown for simplification purposes.

```
1  #include <stdlib.h>
2  #include <pthread.h>
3
4  #define ACCESS_ONCE(x)           (*(volatile typeof(x) *)&(x))
5  struct fifo {
6          char    *fi_data;
7          sig_atomic_t    *fi_len;
8          size_t  *fi_max_len;
9          pthread_cond_t  *fi_cond_full, *fi_cond_one;
10         pthread_mutex_t *fi_mutex_full, *fi_mutex_one;
11 };
12
13 void
14 fifo_wait_for_one(struct fifo *fi)
15 {
16         pthread_mutex_lock(fi->fi_mutex_one);
17         while (ACCESS_ONCE(*fi->fi_len) < 1)
18                 pthread_cond_wait(fi->fi_cond_one, fi->fi_mutex_one);
19         pthread_mutex_unlock(fi->fi_mutex_one);
20 }
21 void
22 fifo_wait_for_full(struct fifo *fi)
23 {
24         pthread_mutex_lock(fi->fi_mutex_full);
25         while (ACCESS_ONCE(*fi->fi_len) < (sig_atomic_t)*fi->fifo_max_len)
26                 pthread_cond_wait(fi->fi_cond_full, fi->fi_mutex_full);
27         pthread_mutex_unlock(fi->fi_mutex_full);
28 }
29 void
30 fifo_inc_len(struct fifo *fi)
31 {
32         ACCESS_ONCE(*fi->fi_len) += 1;
33         if (ACCESS_ONCE(*fi->fi_len) == (sig_atomic_t)*fi->fi_max_len)
34                 pthread_cond_broadcast(fi->fi_cond_full);
35         else
36                 pthread_cond_signal(fi->fi_cond_one);
37 }
```

The example provided by Listing 5.1 allows us to explore two interesting characteristics of conditions in the next sections. Firstly, the example shows the use of locking for waiting on a condition, yet no locking is done for the notification (i.e., `pthread_cond_broadcast` and `pthread_cond_notify`). This will be explored in Section 5.2.1.1. Secondly, we discuss in Section 5.2.1.2 the requirement of a repeatedly looping (i.e., `while` statement) for the condition due to the absence of calls reciprocity.

5.2.1.1 Mutexes and Condition Variables

POSIX defines that waiting on a condition variable must be done while holding a lock; otherwise, the application triggers undefined behavior [IEE16]. Listing 5.1 visually shows on lines 19-22 and 28-31 the expected use, and that the lock is received via a parameter on the function call. In addition, signaling a condition variable does not require locking (lines 39 and 41), and no such variable is received on the signaling function call. Optionally, the application may also choose to lock the signaling thread. This may be required for the scheduling to have predictable behavior because the lock must belong to the signaling thread.

The rationale for employing locking associated with condition variables is that it facilitates real-time implementations as the association can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. Additionally, the association avoids extra context switches and provides more deterministic lock acquisition.

The standard also defines two premises for the association of mutexes and condition variables: (i) mutexes are expected to be locked only for a few instructions; the premise is enforced by the requirement of increasing parallelism, and, thus, the avoidance of long serial regions of code; and (ii) waiting on a condition variable should be a relatively rare situation. A given thread needs to wait for access to the condition variable if another thread is currently testing and calling the waiting call. Yet, by the first premise, the use of the lock should be minimized. The standard estimates that [IEE16]:

> *"The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex"*

The rationale for the association of mutexes and condition variables is sound. However, we believe it is possible to disassociate them and increase the performance of parallel applications. The reasons are as follows. The ease of implementing real-time behavior is interesting, but it may not be necessary for a given set of applications. Therefore, these applications must to pay the cost of locking while they do not use their benefit.

It should be noted that POSIX does not guarantee any releasing order for waiting threads, assuming the default, attribute-less, condition variable, and its associated mutex. Ergo, we propose that the associate be kept for cases where the user has real-time constraints. Otherwise, a more lightweight option, lockless, may be employed.

POSIX also states that locking is expected to be limited to a few instructions. While it is true that developers try to minimize critical sections, due to its performance-degrading effect, there are some considerations to be made. As discussed previously, the signaling thread may or may not use the lock associated with the condition. Therefore, access to condition data has to be made concurrently to support a lockless signaling thread. Generally, concurrent access to shared data will be slower than exclusive access to the same, as

provided, for instance, by mutual exclusion[5]. The concurrent access is slower as it requires atomic operations, memory barriers, and retries (topics explored in Chapter 2).

Another performance-degrading aspect is the recent attacks on microprocessors [LSG+18] [KGG+18] [WVBM+18], especially targeting the x86 architecture. Meltdown forced the use of kernel page-table isolation techniques, which increases the system call overhead [LSG+18]. Spectre and Foreshadow [KGG+18] [WVBM+18] limit the speculation window of the processing unit, which produces slower execution. Experimental results show that performance has been degraded by up to 14% (19% if disabling HyperThreading) for Intel chips [Lar19b] [Lar19a]. These attacks forced the system to be overall slower, yet, the impact on critical sections may be more significant due to its sequential nature on parallel execution. Thus, we believe a lightweight lockless solution may increase the condition performance.

5.2.1.2 Spurious Wakeups with Condition Signaling

One surprising aspect of the `pthread_cond_signal` procedure is the lack of guaranteed reciprocity with `pthread_cond_wait` on the POSIX standard. This leads to spurious wakeups that will be explained with our motivational example.

We propose the following modifications for Listing 5.1: (i) remove calls for `pthread_cond_broadcast`, the associated `fifo_wait_for_full`, and the associated variables (`fi_cond_full`, `fi_mutex_full`); and (ii) assume the developer guarantees that a `pthread_cond_signal` call with be executed only when the FIFO has at least one element; given those conditions, a developer may be tempted to propose the following change on the example: change lines 20 and 29 from a `while` to an `if` statement. Not only would this simplify the code, but also avoid an unnecessary comparison for every `pthread_cond_wait` call. The reason for the statement change is that it is assured by item (i) that only a single thread will be woken up at a given time, and it is guaranteed that the test on lines 20 and 29 would always be true after the call to `pthread_cond_wait` by item (ii). Ergo, it is superfluous to recheck the condition test. This assumption has been made, for instance, for the bodytrack benchmark provided by PARSEC. Although reasonable, unfortunately, the assumption is not valid according to the standard as the latter does not guarantee the expected behavior of `pthread_cond_signal`.

A common misconception [IBM19] [Ora19] [App19][6] of the `pthread_cond_signal` procedure is that it unblocks one thread waiting for the associated condition. In reality, the procedure unblocks at least one waiting thread, and the number of unblocked threads is not known to the caller. Listing 5.2 is the example provided by the standard as the expected implementation of condition procedures, showing that a single `pthread_cond_signal` can wake up two other threads: lines 5 and 16 present the reason. The sequential value

[5] Here, we are comparing only the latency to access to the shared data, and not the full process (i.e., lock acquiring, data access, and lock releasing). For the latter case, concurrent access may be faster.

[6] [App19] is based on the 1996 POSIX standard where the spurious wakeup is not explicitly mentioned.

`cond->value` is used internally to represent that a new signal has arrived, yet, this value is not exclusively attached to any specific thread; hence, multiple threads can perceive the change of the value of `cond->value` and wake up.

Listing 5.2 – Multiple awakenings by condition signal. Numbered comments refer to the order of events. [IEE16].

```
1  pthread_cond_wait(mutex, cond):
2          value = cond->value; /* 1 */
3          pthread_mutex_unlock(mutex); /* 2 */
4          pthread_mutex_lock(cond->mutex); /* 10 */
5          if (value == cond->value) { /* 11 */
6                  me->next_cond = cond->waiter;
7                  cond->waiter = me;
8                  pthread_mutex_unlock(cond->mutex);
9                  unable_to_run(me);
10         } else
11                 pthread_mutex_unlock(cond->mutex); /* 12 */
12         pthread_mutex_lock(mutex); /* 13 */
13
14 pthread_cond_signal(cond):
15         pthread_mutex_lock(cond->mutex); /* 3 */
16         cond->value++; /* 4 */
17         if (cond->waiter) { /* 5 */
18                 sleeper = cond->waiter; /* 6 */
19                 cond->waiter = sleeper->next_cond; /* 7 */
20                 able_to_run(sleeper); /* 8 */
21         }
22         pthread_mutex_unlock(cond->mutex); /* 9 */
```

The standard states that [IEE16]:

"The effect is that more than one thread can return from its call to `pthread_cond_wait`() or `pthread_cond_timedwait`() as a result of one call to `pthread_cond_signal`(). This effect is called "spurious wakeup"."

In other words, the developer cannot assure that the number of calls of `pthread_cond_signal` will be the same as `pthread_cond_wait`. Ergo, it must always use a `while` statement to tolerate spurious wakeups. The standard also mentions that [IEE16]: (i) the spurious wakeup could be resolved, but the event occurs only rarely, and the correction would reduce the degree of concurrency of this operation; and (ii) forcing the use of the `while` loop is considered an added benefit, as it makes the application more robust.

The discussion on this section has been on the standard guarantees; however, the developer does not interface with the standard directly; he interacts with the pthread library

implementation of it. Thus, the implementation may provide different behavior. NPTL [Lin19b] and libthr [Fre19a] are two of the most common implementations found, respectively, on the glibc and the FreeBSD implementation of the C standard library. Both use spin-wait locks[7] to protect shared data of the condition variable and may avoid spurious wakeups. Nonetheless, both provide the same definition of the POSIX standard on their manual page (i.e., `pthread_cond_signal` wakes at least one thread). Therefore, even if they avoid spurious wakeup, they encourage the standardized usage of `pthread_cond_signal`. Besides, relying on the implementation of the standard leads to non-portable code.

5.2.2 Condition Usage Examples from The PARSEC Benchmark

This section demonstrates some of the usages of condition variables on the PARSEC benchmark. The discussion of these applications, and PARSEC itself will be presented in Chapter 6. Here we limit the discussion to condition variables only. No application on the PARSEC benchmark uses the associated mutex for real-time purposes. Nonetheless, we selected three applications to provide examples of condition variable usage.

5.2.2.1 Bodytrack

Bodytrack is implemented with a single master and multiple worker threads to execute its work. The master sends commands to the worker threads and they execute them. Bodytrack uses the single condition `workAvailable` for threads to sleep waiting for new commands. Listing 5.3 shows the condition usage by using two procedures: `RecvCmd` and `SendInternalCmd`. A worker thread that arrives at the `RecvCmd` procedure will check if any new work is available (i.e., `cmd != THREADS_IDLE`); if that is not the case, it will sleep on the condition. This procedure correctly uses the `while` loop to tolerate spurious wakeups[8]. The `SendInternalCmd` is only called by the master, and it provides new work and wakes up any sleeping threads. The `workDispatch` lock seems to be used for (i) the condition and (ii) protecting the shared `cmd` variable. However, its use for (ii) is superfluous; `cmd` does not need protection for its current use and may be an artificial of an earlier version. Besides the `SendInternalCmd` procedure, the `AckCmd` procedure also writes to cmd. For both cases, no lock is needed for protecting since (i) `SendInternalCmd` is used on one thread only; (ii) the `pthread_barrier_wait` guarantees that only one thread will receive a positive value, thus, `cmd` will be written only by this single thread; and (iii) `SendInternalCmd` only finished after the barrier `poolReadyBarrier` is finished, which only happens after `AckCmd` has already written

[7] A spin-wait lock spins for a given amount of time and, if it fails to acquire the lock, enters the waiting queue and sleeps; otherwise, no sleep is required, and a context switch is avoided.

[8] In the previous section, we mentioned Bodytrack incorrectly uses an `if` statement for the condition variable. This happens in another set of procedures called `Run` and `GetNextImageSet` for asynchronous IO processing.

to the `cmd` variable. Thus, we postulate that `workDispatch` is a lock exclusively used for the condition variable.

Listing 5.3 – Bodytrack's condition variable usage. Adapted from C++ to C.

```
1  unsigned short int
2  RecvCmd(void)
3  {
4          unsigned short int _cmd;
5
6          pthread_mutex_lock(workDispatch);
7          while (cmd == THREADS_IDLE)
8                  pthread_cond_wait(workAvailable, workDispatch);
9          _cmd = cmd;
10         pthread_mutex_unlock(workDispatch);
11         return (_cmd);
12 }
13
14 void
15 SendInternalCmd(unsigned short int _cmd)
16 {
17
18         pthread_mutex_lock(workDispatch);
19         cmd = _cmd; /* send command */
20         pthread_cond_broadcast(workAvailable);
21         pthread_mutex_unlock(workDispatch);
22
23         /* wait until all work is done and pool is ready */
24         pthread_barrier_wait(poolReadyBarrier);
25 }
26
27 void
28 AckCmd(void)
29 {
30         int master;
31
32         master = pthread_barrier_wait(workDoneBarrier);
33         if (master) {
34                 pthread_mutex_lock(workDispatch);
35                 cmd = THREADS_IDLE;
36                 pthread_mutex_unlock(workDispatch);
37         }
38         pthread_barrier_wait(poolReadyBarrier);
39 }
```

5.2.2.2 Streamcluster

Listing 5.4 – Streamcluster's condition variable usage. Adapted from C++ to C.

```
void
pspeedy(Points *points, float z, long *kcenter, int pid, pthread_barrier_t *barr)
{
        pthread_barrier_wait(barr);
        static bool open = false;
        static int i;
        (...)
        pthread_barrier_wait(barr);
        if (pid != 0) {
                /* we arent the master thr. We wait until a center is opened */
                while (1) {
                        pthread_mutex_lock(mutex);
                        while (!open)
                                pthread_cond_wait(cond);
                        pthread_mutex_unlock(mutex);
                        (...)
                        pthread_barrier_wait(barr);
                        pthread_barrier_wait(barr);
                }
        } else {
                /* I am the master thread. I decide whether to open a center
                   and notify others if so. */
                for (i = 1; points->num; i++) {
                        (...)
                        if (to_open) {
                                (*kcenter)++;
                                pthread_mutex_lock(mutex);
                                pthread_cond_broadcast(cond);
                                pthread_mutex_unlock(mutex);
                                (...)
                                pthread_barrier_wait(barrier);
                                open = false;
                                pthread_barrier_wait(barrier);
                        }
                }
                pthread_mutex_lock(mutex);
                open = true;
                pthread_cond_broadcast(cond);
                pthread_mutex_unlock(mutex);
        }
        (...)
}
```

Streamcluster follows the same strategy of using one master and multiple worker threads. It relies heavily on the use of barriers for synchronization; the condition variable is used to wait for a new center opened by the master. Listing 5.4 depicts the streamcluster code. When the center has been opened, the variable `open` goes to true. Once again, as the variable is only written by the master, no lock is required. Besides, the variable is written without the lock on line 35. Therefore, the mutex is exclusively used due to the requirement of the condition variable.

5.2.2.3 Ferret

Ferret uses a pipeline model with six stages, where each stage has its thread pool for working. The condition variables have two objectives: (i) notify any sleeping thread that a new element has been added to the queue; and (ii) notify all sleeping threads that the queue has been terminated. For this application, the lock `que->mutex` also has two objectives: (i) associates with the condition variable; and (ii) protects the queue shared data, specifically `que->tail` and `que->end_count`. Therefore, this lock is essential for the application workflow.

Listing 5.5 – Ferret's condition variable usage.

```
1  int dequeue(struct queue *que, void **to_buf)
2  {
3          pthread_mutex_lock(&que->mutex);
4          while (que->tail == que->head && (que->end_count) < que->prod_threads)
5                  pthread_cond_wait(&que->empty);
6  
7          /* check if queue has been terminated */
8          if (que->tail == que->head && (que->end_count) == que->prod_threads) {
9                  pthread_cond_broadcast(&que->empty);
10                 pthread_mutex_unlock(&que->mutex);
11                 return (-1);
12         }
13         *to_buf = que->data[que->tail];
14         que->tail++;
15         if (que->tail == que->size)
16                 que->tail = 0;
17         pthread_cond_signal(&que->empty);
18         pthread_mutex_unlock(&que->mutex);
19         return (0);
20 }
```

5.2.3 Design and Implementation Choices

We propose a novel design for the condition synchronization called neocondition. It provides the same functionality while removing the requirement of an associated mutex. As was discussed previously in Section 5.2.1.1, we believe the removal of the lock dependency may provide performance benefits for parallel applications. Developers that require the use of the mutex for protecting shared data or the priority transfer from the condition to the mutex may continue to use the PThreads condition with Subutai.

Neocondition is a synchronization variable that demands changes in three aspects of the system: the parallel application, the synchronization library, and, in the case of Subutai, the Subutai-HW. As the Subutai-HW will handle the waiting queue of threads, the kernel can work the same as it does with PThreads condition.

5.2.3.1 Application Changes

The parallel application needs to be modified to reap the benefits of neocondition. A parser can be used to identify and replace the locations of possible use automatically. The parser would keep track of the condition variable usages and any data that is accessed in the mutex-unlock code section. Then, it would check if that data requires protection by locking, mainly by observing how that data is written. In Section 5.2.2, we presented three usages of the PThreads condition. The first one, Bodytrack, may require user-intervention as it may confuse a parser due to its unnecessary use of the lock. The second one, Streamcluster, can be fully automated. The last one, Ferret, cannot benefit from neocondition and would be refused for the parser. The replacement of code can be made with a macro, as this would allow easy exchange of PThread condition and neocondition without incurring any performance overhead. Listing 5.6 demonstrates the common API among PThreads condition and neocondition. The developer can force the use of PThreads condition with the `force_condition` parameter. Note that as this value will be constant and directly written by the developer, the compiler can optimize from the `if/else` conditional execution to a direct call to the respective case. A more complex macro can be done to provide execution between the lock-unlock code section; however, this would be needed most likely because the application should not use neocondition. Thus, the application should avoid the common API and use PThreads conditions.

We highlight the fact that although the application must be changed to reap the benefits of neocondition, its workflow stays the same. The developer has minimal, if any, effort required to refactor the code.

Listing 5.6 – Common API among PThreads condition and neocondition.

```
#define COMPAT_COND_WAIT_IF(exp, mutex, cond) do {                  \
        pthread_mutex_lock(mutex);                                  \
        while (exp)                                                 \
                pthread_cond_wait(cond);                            \
        pthread_mutex_unlock(mutex);                                \
} while (0)

#if defined(USE_NEOCONDITION)
#define COND_WAIT_IF(exp, mutex, cond, force_condition) do {    \
        if (force_condition)                                        \
                COMPAT_COND_WAIT_IF(exp, mutex, cond);              \
        else {                                                      \
                neocondition_seqnum(cond);                          \
                while (exp)                                         \
                        neocondition_wait(cond);                    \
        }                                                           \
} while (0)
#else
#define COND_WAIT_IF(exp, mutex, cond, unused)                      \
        COMPAT_COND_WAIT_IF(exp, mutex, cond)
#endif
```

5.2.3.2 Synchronization Library Changes

For the synchronization library, we propose a lockless implementation to avoid incurring sequential code. In Section 5.2.1.1, two of the most important implementations of PThreads were discussed, and they both used locks for controlling access to shared data. In addition, in Section 5.2.1.2, we have demonstrated that the standard also assumes the use of locking for controlling shared-data. Nonetheless, with the majority of the operation done at the hardware-level for Subutai, the lockless implementation can be made easier. The OS primitives and the Subutai-driver still require the use of locking as they are done entirely at the software-level; however, this is already the implementation used for the other synchronization primitives.

Internally, neocondition works as thus. As discussed by the standard (Section 5.2.1.2), conditions can be represented as values – therefore, neocondition uses a stateful sequential variable to control notifications. This is called the sequential number of neocondition. The sequential number is an increasing value; every time a notification is received, the sequential number is increased on one value unit. It should be noted that the sequential number has no relation to the condition test; though, this is the same behavior with PThreads condition – as shown in Listing 5.6, both of them do not have access to the condition test, they are done externally on lines 3 and 14.

Listing 5.6 shows that before every call to `neocondition_wait` there is a call to `neocondition_seqnum`. The latter procedure records internally the last known sequential number observed by this thread. Then, from the call to `neocondition_wait` until the thread sleeping, different components of the system can check if the sequence number has changed. Similarly to the glibc and the futex implementation of Linux kernel, we check twice for sequence number changes: (i) in the synchronization library, after `neocondition_wait` is called but before the kernel space is invoked; and (ii) in the kernel, after the task is put to sleep but before the thread goes to sleep. The last check is a common kernel technique to avoid losing wake-up calls [CRKH05].

PThreads already ensure that the developer uses opaque pointers for the condition variable; in other words, the developer does not know the contents of the condition datatype `pthread_cond_t`. Thus, our addition of the sequence number does not impact the application directly.

5.2.3.3 Subutai-HW Changes

Neocondition reuses the basic processing of PThreads barrier. Neocondition does not reuse the condition processing since the former requires the management of two queues (condition and mutex), while neocondition and barrier require only one queue. The distinction between barrier and neocondition is that the former checks for a specific number of threads while the latter checks for a new value. For conditions, the data field of the control structure (Figure 4.4) is a single pointer to the mutex synchronization. Neocondition reuses the data field as a single 32-bit integer for the sequential number.

Table 5.2 presents the complexity of the required four new states for neocondition handling. They are used for: (i) retrieving the current sequential number on a given neocondition variable (named neocondition seqnum); (ii) sleeping on a neocondition variable (named neocondition wait); and (iii) notifying one and all threads waiting on a neocondition (named neocondition signal and broadcast, respectively). This Table, as done with Table 4.2, does not include the entry and exit time of the states: both times are the same as shown in Table 4.3.

Neocondition seqnum is a read on a memory position of the SPM. Thus, it takes one memory operation and one cycle to request the creation of a new packet. Neocondition has no mutex attached to it; thus, no mutex unlocking operation is required for waking up threads. Neocondition wait either (i) is avoided if the sequence number has changed since the call to neocondition seqnum state, or (ii) the thread sleeps on the neocondition. They are represented by the best and worst response time of the neocondition wait state, respectively. Case (i) is straightforward: it reads the sequence number (latency = $1m$) and compares it to the number on the packet (latency = $1c$). If it does not match, a new sequence number has been generated since the last read; therefore, a packet is sent to the requestor (latency = $1c$). Otherwise, case (ii), the thread is queued up – this is the same process as described in

Table 5.2 – Complexity of neocondition states. c = cycle latency, m = memory latency, n = number of synchronization variables handled by Subutai-HW, ϱ = number of threads on a neocondition.

State	**Best response time**	**Worst response time**	**Packet Injection**
Neocondition seqnum	$1m + 1c$	$1m + 1c$	$1m + 1c$
Neocondition wait	$1m + 2c$	$11m$	$1m + 2c$
Neocondition signal	$2m + 1c$	$10m + 1c$	$10m + 1c$
Neocondition broadcast	$2m + 1c$	$2m + 1c + \varrho \times (10m + 1c)$	$(13m + 1c) + (23m + 2c) + (33m + 3c) \ldots$

Section 4.2 requiring 11 memory operations (i.e., allocate a queue entry and enqueue it on the neocondition queue).

Signaling a neocondition event for at least[9] one thread is achieved on the state Neocondition signal. For this state, the sequential number is updated (latency = $2m$)[10] and the queue is checked for sleeping threads (latency = $1c$). If there are no sleeping threads, the state is finished; otherwise, one sleeping thread is woken up, and a packet is sent to it (latency = $8m$). Broadcasting a neocondition event has the same behavior of signaling, but instead of waking one thread, it will wake up all sleeping threads. Therefore, this case is similar to the releasing phase of barriers. The threads will be periodically released every time a queue entry is consumed. Reading each queue entry and preparing the packet takes 10 memory operations and one Subutai-HW cycle.

5.2.4 The Positive and Negative Attributes of neocondition

In the last three Sections, the design of neocondition was discussed. Now, we provide a comparison of the characteristics of neocondition against PThreads conditions from the developer's point-of-view. Table 5.3 depicts the positive and negative attributes of neocondition compared to PThreads condition. Each set of positive and negative attribute on a line are related and will be explained shortly.

The first of the set of attributes is the compatibility of neocondition with existing parallel applications. These applications can be patched to use the novel API of neocondition

[9] Neocondition has the same limitation as PThreads condition for signaling: more than one thread can read the updated sequential number. Refer to Section 5.2.1.2 for the discussion of this scenario.

[10] If we assumed only one thread would signal events, then the sequential number could be read from the packet, and a memory read would be avoided; however, the application can use multiple threads for signaling; thus, we first read and then write the memory position.

Table 5.3 – Positive and negative attributes of neocondition compared to PThreads conditions.

Positive attributes	**Negative attributes**
Legacy source-code compatible	Not fully compatible with existing parallel applications
Two locks removed	Neocondition is susceptible to the thundering herd phenomenon
Condition deals with two queues; Neocondition, one	Neocondition may be problematic for real-time implementations

while retaining their existing parallel workflow. Maintaining the existing workflow is a key feature of neocondition, as redesigning this aspect is onerous (redesigning parallel application is discussed in Chapters 1 and 2). The use of neocondition may even be possible for cases where only the binary is available, by employing binary substitutions; yet, this has not been validated in this Thesis. However, even though neocondition is source-code compatible, it is not compatible with every parallel application. As discussed in Section 5.2.3.1, applications that use the associated lock for shared-data access (e.g., Ferret) cannot be converted to neocondition, as it would make the application susceptible to race conditions on their shared data.

The second set of attributes is related to the removal of locks. We propose the removal of (i) the associated lock of conditions, and (ii) the general use of locks for the neocondition handling in the library. The standard does not force the use of a lock for the internal processing of conditions; yet, this is commonly found, as shown in Section 5.2.1.2. Thus, locking would be avoided in two of three places: only kernel space would use locking. Unfortunately, the removal of the associated lock makes neocondition susceptible to the thundering herd phenomenon [Lin19a]. The essence of the thundering herd is thus. Given a set of $1 \ldots n$ threads, they each arrive at the neocondition in an increasing time order T; thus, $Tn - 1$ arrives before than Tn. In addition, the time difference of the first to the n thread can be in the order of seconds. Eventually, all n threads will be released by a given thread, either by signaling or broadcasting. If the thread uses broadcast, then all n threads will be released, roughly at the same time, and execute the code next to the neocondition wait call. If the proceeding code is a request for locking (i.e., `pthread_mutex_lock`), then all n threads will try to acquire the lock, and $n - 1$ calls will be pointless because all except one of the threads will be able to acquire it. Thus, $n - 1$ will go to sleep again, this time waiting on a lock. In other words, computational resources are wasted in this scenario. The association of a lock with the condition variable avoids the scenario, as even with broadcasting, only one thread will truly be awoken (i.e., all other threads are transferred directly to the mutex waiting queue). This is the second scenario that neocondition is not recommended, but, for this case, it is merely a performance issue, while the other, mentioned in the previous paragraph, has a concurrency correctness violation.

Finally, the last set of attributes is related to the queue usage of the neocondition and PThreads condition. Neocondition only handles one queue, while PThreads condition handles two (i.e., mutex and condition queue). On the one hand, there are two reasons for handling two queues: (i) the two queues avoid the thundering herd phenomenon; and (ii) facilitates the development of real-time implementations. On the other hand, neocondition aims to increase performance by avoiding mutual exclusion use.

The experimental results of neocondition will be described in Section 6.3.2.3.

6. EXPERIMENTAL RESULTS

> It is all too easy to denigrate Dijkstra from the viewpoint of the year 2012, more than 40 years after the fact.
>
> If you still feel the need to denigrate Dijkstra, my advice is to publish something, wait 40 years, and then see how *your* words stood the test of time.
>
> Paul McKenney on Dijkstra's synchronization solution for the Dinning Philosopher's problem

This chapter presents the experimental results conducted for Subutai. Similar to other mixed and hardware solutions, we used an architecture simulator to provide a quantitative evaluation. We employed the PARSEC benchmark as our target parallel application set, as it offers a wide range of distinct application domain and parallelization granularity [BKSL08]. We also developed a micro-benchmark to demonstrate key aspects of our solution.

This chapter is organized as follows. Section 6.1 details the setup environment for the quantitative evaluation. Section 6.2 introduces the PARSEC benchmark and details the parallelization model used in some of the applications. Finally, Section 6.3 presents and discusses the experimental results.

6.1 Experimental Setup

A full system simulator is an architecture simulator capable of executing software stacks from real systems (user and kernel code) without any modification [EAW10]. Such a tool can create virtual platform designs capable of gathering experimental data with workloads compatible with the running software. Gem5 is one simulator based on discrete event simulation, which is the result of the combined effort of a myriad of industrial and academic institutions such as AMD, ARM, University of Michigan and University of Texas. Gem5 aims to be a community-driven tool focused on object-oriented design for architecture modeling [BBB+11]. The accuracy of Gem5 has been a topic of interest of many researchers [BGOS12] [ECC14] [GPD+14]. Overall, Gem5 has been found to have discrepancies within the acceptable range. Some of them have been alleviated with latency model tuning [GPD+14].

However, the simulation of computer architectures requires tremendous computational effort since it comprises any number of processors, memories, and I/O devices. Thus, accurate low-level descriptions of hardware-level simulation, such as RTL, and detailed

hardware simulation model, increase the time for design exploration making prohibitive the entire system simulation [BGOS12] [GPD+14]. A single detailed simulation on Gem5 for PARSEC applications goes from 7.5 hours (Blackscholes application) up to 108 hours (x264 application) on a multicore system comprised of 8 cores [Cat15]. Our target architecture, comprised of 64 cores, makes the simulation significantly slower[1], making it prohibitive for benchmarking.

Therefore, we employ Gem5 to collect essential information to create a dynamic trace, which can be simulated significantly faster. The same approach has been used extensively with Gem5 [BGO+15] [NSM+15] [NBSG17]. In fact, our trace approach is very similar to Butko et al. [BGO+15]; the traces identify the functions that do and do not depend on synchronization primitives. Accordingly, we can simulate all synchronization primitives and understand their impact on the rest of the code. Figure 6.1 shows the methodology flow of our experimental setup in four steps.

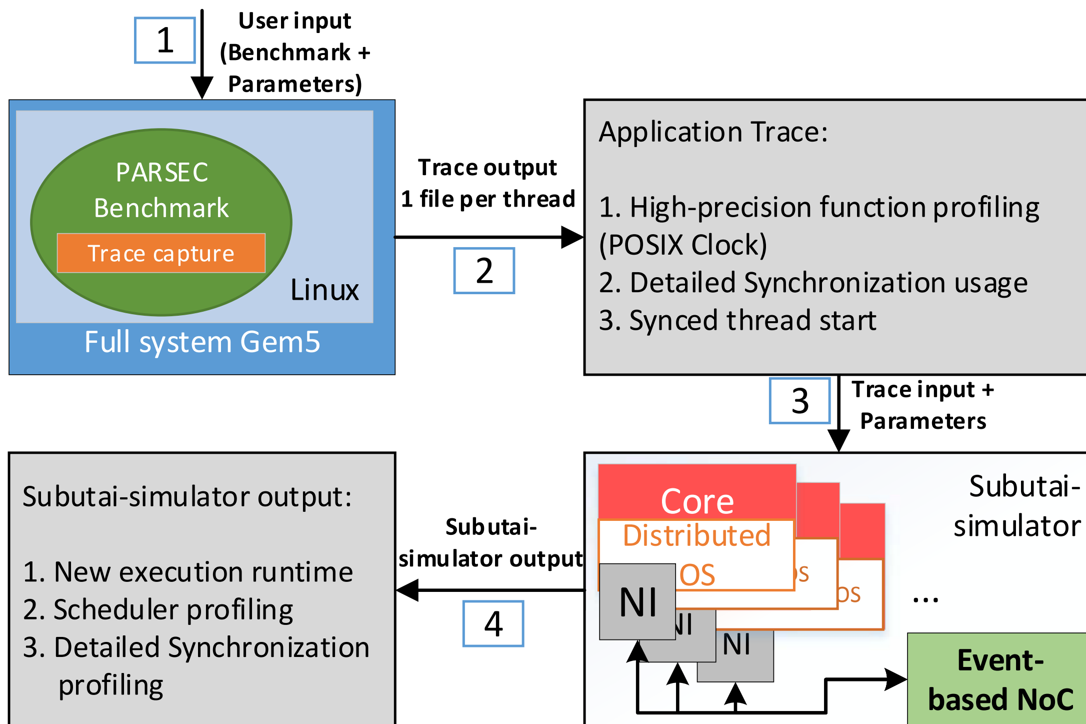


Figure 6.1 – Experimental setup for Subutai evaluation.

The first step of our setup is the application execution using Gem5 in full-system simulation mode to provide an accurate characterization of the application. We run the entire PARSEC application utilizing the `simmedium` input size.

The second step is the application execution trace, from start to finish, as the sequential portions of code can hinder the real speedup of any parallel application [SR15] (c.f. Figure 1.1). The application trace provides the execution times between synchronization calls and the number and execution time of each synchronization call. The trace is further

[1]Gem5 is a single thread application; thus every additional core that has to be simulated will increase the simulation time.

annotated with every synchronization primitive's metadata – 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 manycore simulator called Subutai-simulator (modeled in SystemC) that reads the traces to generate tasks in the OS. Then, these tasks mimic the execution of the application threads, according to the execution times from the traces, and execute the synchronization functions. We reproduce NoC communication, queues and hardware latencies in our SystemC environment. The NoC was set up for 32 bits links, no virtual channel and an I/O buffer of 16×32 bits per each router port.

Initially, we intended to use an existing cycle-based NoC simulator for interconnect simulation. While providing the most accurate results, cycle-based simulation is extremely slow. As we target real applications, our simulated time is in the order of magnitude of seconds, which is 10^9 nanoseconds. Assuming 64 routers executing at 1GHz (i.e., 1ns clock cycle), for a 1 second simulated time, there are $64 \times 1 \times 10^9 = 6.4 \times 10^{10}$ events generated regardless if any processing is required (e.g., new data). Besides, it is necessary to have traffic injectors to use the NoC, making simulation even slower. In other words, cycle-based simulation makes the user always simulate the worst-case scenario. Therefore, we adapted an existing NoC implementation to be event-based; in this way, the cost of simulation is proportional to the user's demand.

Table 6.1 shows the simulation time for some NoC simulators executing 1 second of simulated time without injecting packets (i.e., injection rate of 0%) on an 8×8 NoC (i.e., 64 cores). The test was executed on two Intel-based machines. The following simulators were tested: (i) ShoC [CCD+15] – a cycle-based NoC simulator with flit precision; (ii) Noxim [CMM+16] – also a cycle-based NoC simulator with flit precision aimed at wired and wireless networks; (iii) Noxim-XT [MLBR17] – an extension to Noxim for bit-accurate power estimation; and (iv) Capgras – the event-based simulator for Subutai-simulator.

Table 6.1 – Simulation time for some NoC simulators executing 1 second of simulated time without injecting packets.

NoC Simulator	**Injected packets**	**Simulated time**	**Simulation time (Xeon W3520 @2.67GHz)**	**Simulation time (Xeon E5-2660 v3 @3.3GHz)**
ShoC	0	1 second	3 078m49s	2 532m37s
Noxim	33[1]	1 second	477m38s	341m29s
Noxim-XT	36[1]	1 second	1 296m45s	1 402m50s
Capgras	189[2]	1 second	0m 7s	0m 7s

1. Noxim does not allow 0% injection rate; hence, we employed an injection rate of 1×10^{-9}%.
2. 63 packets from Thread_0 to all other cores to create additional 63 threads. Then, 126 packets from and to Thread_0 for join operation (i.e, `pthread_join`).

ShoC is the most demanding simulator because it uses independent injectors for each router; consequently, it has 64 traffic injectors operating at a cycle-accurate level as well. Noxim, on the other hand, has only one global injector, which makes it faster compared to

ShoC. Noxim-XT increases the computational cost of Noxim by inserting monitors for energy evaluation. Finally, Capgras is the solution proposed by us since the computational resources are proportional to the NoC usage. Capgras also employs independent injectors that only operate when new data is available. Note that this test uses a sleeping application that does not generate any events, thus, we just present the overhead of the simulator. Therefore, an event-based NoC is not the main bottleneck of the simulator (i.e., avoids spurious event every 1ns). Currently, the main bottleneck of the simulator is the scheduler tick.

Finally, the OS latencies were extracted from FreeRTOS [Ama19]. The processing cores are clocked at 1GHz 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 environment are clustered in the fourth step of Figure 6.1 and discussed in Section 6.3.

6.2 PARSEC – Benchmark Suite for MultiProcessing

Benchmarking is the quantitative foundation for computer architecture research [BKSL08]. Without a program selection that provides a representative load of the target application space, performance results can be skewed and invalidate conclusions drawn from it. A well-known fact of multiprocessing is the disruptive change of programming models for programs to benefit from their full potential. The use of older High-Performance Computing (HPC) workloads does not fit this scenario since it is based on smaller suites and sequential applications. This shortcoming is the target intended to be answered by the Princeton Application Repository for Shared-Memory Computers (PARSEC) suite [BKSL08]. Intel and Princeton University created the first version of PARSEC; the latest version available of PARSEC is 3.0 [Pri19]. It is a highly used benchmark with more than 55 papers in International Symposium on Computer Architecture from 2010 to 2014 [SR15].

The five objectives proposed by PARSEC are described as following:

1. **Multi-threaded Applications** – Shared-memory multiprocessor is one of the most employed architecture today for HPC. The trend for future architectures is to deliver performance improvements through increasing core counts on multiprocessing. Therefore, applications that require processing power must use a parallel model of execution.

2. **Emerging Workloads** – The increase of processing power enables new classes of applications whose computational requirements were beyond the capabilities of earlier generations of processors; hence, the benchmark suite should represent this trend.

3. **Diverse** – A benchmark suite must be broad in its representative load of applications, which includes both interactive applications like computer games, offline applications

like data mining, and programs with different parallelization models. While a real representative suite is impossible to create for all cases, reasonable effort should be applied to maximize the diversity of the program selection.

4. **Employ State-of-the-Art Techniques** – A benchmark suite must be up-to-date with current practice in parallel application techniques.
5. **Support Research** – A benchmark suite intended for research has additional requirements that go beyond the ones used for benchmarking real machines alone. Representative input sets with different proprieties should be provided.

PARSEC fulfills these objectives by providing rich, parallelized, state-of-the-art applications with diverse areas of research. The areas contemplated are computer vision, media processing, computational finance, enterprise servers, and animation physics. Table 6.2 summarizes the key characteristics of PARSEC benchmarks.

Table 6.2 – Qualitative summary of key characteristics of PARSEC benchmarks [BKSL08].

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

PARSEC provides three categories of input sets for each benchmark. The `test` and `simdev` are tiny input sets intended for testing and development, and should not be used for scientific studies. The input sets `simsmall`, `simmedium`, and `simlarge` are intended for simulators and are progressively larger (i.e., larger inputs contain more working sets and parallelism). They represent approximately the runtime execution of 1, 5, and 15 seconds, respectively. Finally, the `native` input set is the most interesting one because it is a real program input. However, its runtime execution is about 15 minutes, which is prohibitive for a full system simulator. Table 6.3 details the types of instructions and synchronization primitives employed on all benchmark applications under an 8-core system with the input set `simlarge`.

Southern et al. [SR15] evaluated the scalability of PARSEC and found that none of the application achieved the perfect speedup. Figure 6.2 summarizes the results obtained by them. The highest core-count machine used was a quad-socket system with 12-cores per socket, totaling 48 cores. For the entire application execution, the best speedup achieved was for the swaptions application – speed up of 30$\times$ and 25$\times$ for the `simlarge` and `simmedium` input sizes, respectively. However, for Bodytrack, for instance, the best speedup was 9$\times$ and

Table 6.3 – Breakdown of finer details of the benchmark applications for input size `simlarge` on an 8-core system [BKSL08].

Program	Problem Size	Instructions (Billions)				Synchronization Primitives		
		Total	FLOPS	Reads	Writes	Locks	Barriers	Conditions
`blackscholes`	65,536 options	2.67	1.14	0.68	0.19	0	8	0
`bodytrack`	4 frames, 4,000 particles	14.03	4.22	3.63	0.95	114,621	619	2,042
`canneal`	400,000 elements	7.33	0.48	1.94	0.89	34	0	0
`dedup`	184 MB data	37.1	0	11.71	3.13	158,979	0	1,619
`facesim`	1 frame, 372,126 tetrahedra	29.90	9.10	10.05	4.29	14,541	0	3,137
`ferret`	256 queries, 34,973 images	23.97	4.51	7.49	1.18	345,778	0	1255
`fluidanimate`	5 frames, 300,000 particles	14.06	2.49	4.80	1.15	17,771,909	0	0
`freqmine`	990,000 transactions	33.45	0.00	11.31	5.24	990,025	0	0
`streamcluster`	16,384 points per block, 1 block	22.12	11.6	9.42	0.06	191	129,600	127
`swaptions`	64 swaptions, 20,000 simulations	14.11	2.62	5.08	1.16	23	0	0
`vips`	1 image, 2662 × 5500 pixels	31.21	4.79	6.71	1.63	33,586	0	6,361
`x264`	128 frames, 640 × 360 pixels	32.43	8.76	9.01	3.11	16,767	0	1,056

6× for the same set of inputs, respectively. The geometrical mean for the speedup of the entire application set was 5× and 4× for the same set of inputs, respectively.

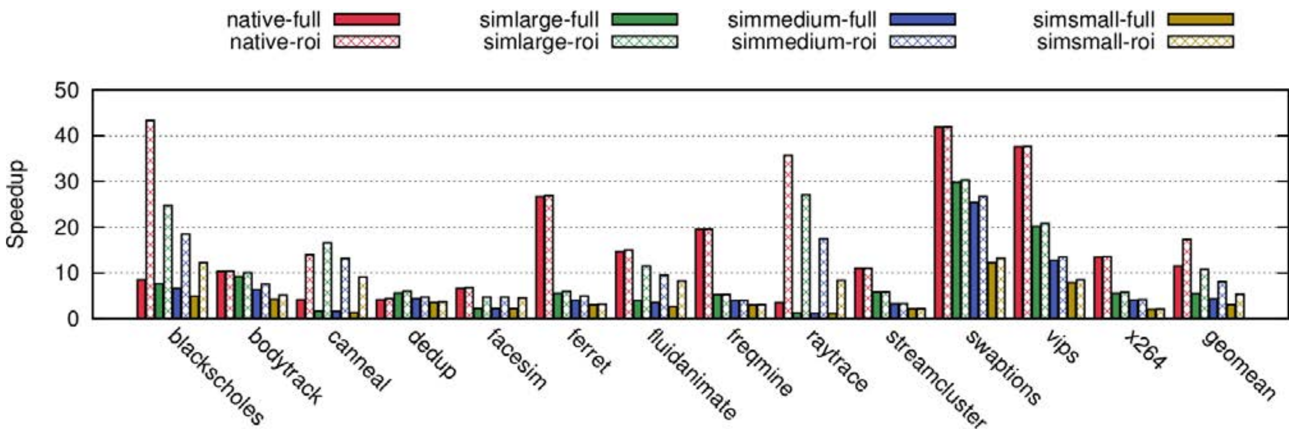


Figure 6.2 – Maximum speedup measured for a 48-core system for each benchmark, region, and input set combination. Full and ROI inputs represent the entire application and parallel portion, respectively [SR15].

We experiment on four PARSEC applications: Bodytrack, Streamcluster, Facesim, and x264. Bodytrack and Streamclusters are the only two applications that use the three types of synchronization primitives; then, we chose Facesim and x264 for their application domain and parallelization techniques (Figure 6.3).

Table 6.4 depicts the number of synchronization primitive calls for our application set while using 16, 32, and 64 threads. Summing up all values, Streamcluster is the application most dependent on synchronization operations, followed by Facesim, Bodytrack, and finally x264. Streamcluster and Facesim reach hundreds of thousands of operations, while Bodytrack and x264 reach tens of thousands. For each type of operation, we have the following order:

Streamcluster has the most calls for barriers operations, and Facesim has the most calls for conditions and mutex operations.

Table 6.4 – Number of events of synchronization primitives during the execution of PARSEC applications.

Application	Type	Events per number of threads		
		16	32	64
Bodytrack	Barrier[1]	2 101	4 293	13 416
	Condition	447	750	1 529
	Mutex	9 000	10 472	8 677
Streamcluster	Barrier[1]	208 048	364 480	728 960
	Condition	381	802	1 274
	Mutex	510	1 054	2 142
Facesim	Barrier	0	0	0
	Condition	18 850	36 834	72 998
	Mutex	55 735	198 070	411 148
x264	Barrier	0	0	0
	Condition	86	310	354
	Mutex	4 154	4 340	4 344

(1) Every packet is counted as an independent event. Therefore, for a 64 barrier, for instance, 64 events are generated for waiting on a barrier, and another 64 events are generated for releasing them, as we do not support broadcast on the interconnect fabric.

Table 6.5 shows the number of synchronization primitives utilized for the core workflow of each application in our set. Facesim and x264 use approximately a hundred of primitives, while Bodytrack and Streamcluster are limited to half a dozen. Only the Facesim application changes the number of primitives according to the number of threads instantiated. Besides, no barrier variables are employed in Facesim and x264; thus, it justifies the absence of barrier calls in Table 6.4.

Table 6.5 – Number of synchronization primitives for PARSEC (`simmedium` input). n = number of threads.

Application	Mutex	Condition	Barrier
Bodytrack	3	1	4
Streamcluster	1	1	1
Facesim	$n + 1$	2	0
x264	95	95	0

The next sections describe the high-level implementation details of our application set. Their description is based on Bienia et al. work [BKSL08] and the implementation provided by PARSEC. We target an agnostic solution in terms of the application domain; hence, we only focus on the communication model. Bodytrack and Streamcluster use barrier-based synchronization for synchronizing threads on a predetermined code point. Thus, they are both highly susceptible to delays, as all threads are blocked until the slowest one reaches

the predetermined point. Facesim and x264 do not use such an approach; thus, they are less susceptible to delays (i.e., slowing down one thread does not affect all others directly, as it happens with the former two applications).

6.2.1 Bodytrack Communication Model

Bodytrack is a computer-vision application that tracks a 3D pose of a mark-less body. Figure 6.3 shows the result of Bodytrack: a processed image frame. It uses 6 mutexes, 4 barriers, 3 conditions, and employs 3 types of threads to sort inputs of T threads. First, a single 'master' thread (T_0) is responsible for creating synchronization primitives, creating T_{t-1} threads, and sending computation requests for them. Then, the threads $T_1, \ldots, T_{t-2}$ do the actual computation through the requests from T_0. Finally, the last thread (T_{t-1}) performs asynchronous I/O operations (e.g., loading images from disk to memory).

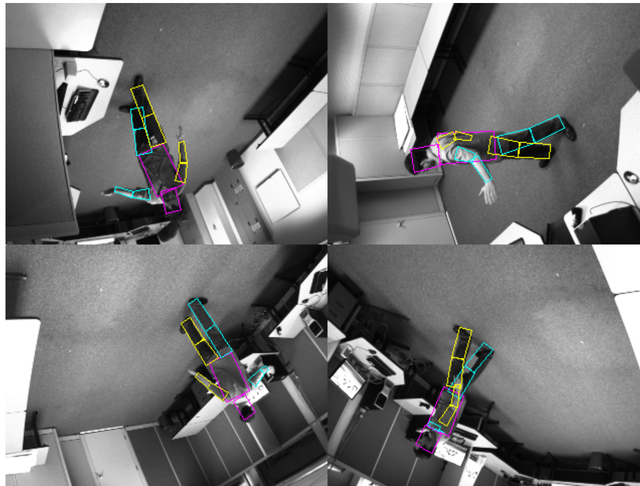


Figure 6.3 – Bodytrack's output.

Figure 6.4 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 for the number of worker threads available; when it is done, it sends a condition broadcast for all worker threads. Meanwhile, the worker threads are checking if their work is already available: if it is not, it waits on the condition variable; if it is, it skips the condition and goes to the next phase. The next phase for the worker threads is the computational part. It uses mutexes to access shared data.

Meanwhile, T_0 waits for all worker threads to finish; in this case, it uses a barrier condition. The barrier guarantees that all worker threads are ready to handle the next work request. As the worker threads finish their work, they join the barrier as well. Only when all threads have joined the barrier, they are released to execute the next phase. The next phase

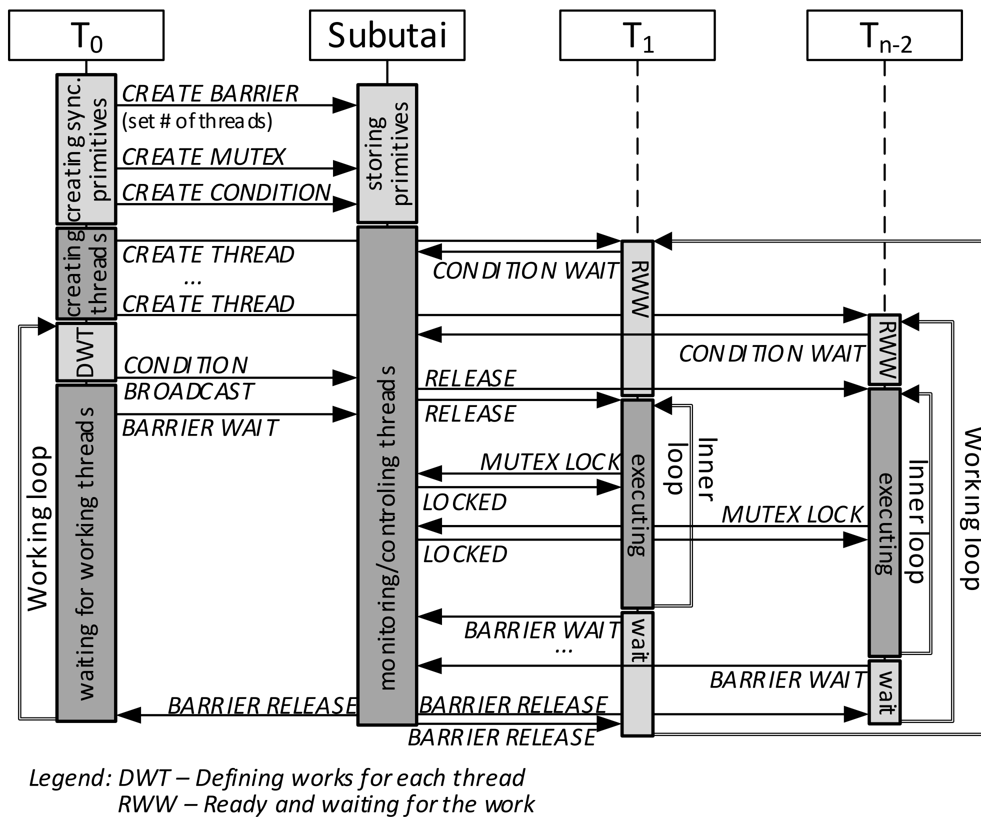


Figure 6.4 – Bodytrack's core workflow.

loops back to the generation of more work for the worker threads in case of T_0, and waiting for said generation for all worker threads. This loop is executed until no more work is available.

The workflow plotted in Figure 6.4 omits three aspects of Bodytrack's work: (i) after the worker threads have received a request through the condition, they acknowledge it through the use of another barrier (not shown in the Figure), and the associated mutex of the condition; (ii) the thread responsible for asynchronous I/O (T_{t-1}), because it communicates only with T_0 and produces very few events compared to the core workflow presented in the Figure; (iii) the process of application termination, because it does not use data synchronization.

6.2.2 Streamcluster Communication Model

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. Streamcluster has a much simpler communication model than Bodytrack, using a single instance of mutex, barrier, and condition. Table 6.4 shows that Streamcluster is much more dependent on barrier synchronization than Bodytrack. Once again, a 'master' thread (T_0) creates multiple threads for computation. These threads perform the actual computation; however, they do not need to receive commands from T_0. The worker thread T_1 is also a

master thread when it is required to open a new center for clustering. Otherwise, the worker threads synchronize through barriers only.

6.2.3 Facesim Communication Model

Facesim simulates motions of a human face for visualization purposes. It receives as input a face model and a time sequence of muscle activations. Facesim employs the fork-join model of parallel processing; however, instead of using a barrier, it uses two conditions and a mutex to synchronization all threads. In addition, faster threads can steal work from slower threads so Facesim can work in parallel in situations that other applications cannot (e.g., Bodytrack and Streamcluster as both do not have work-stealing capabilities). The mutexes that are generated according to the number of threads (Table 6.5) are employed for work-stealing. Therefore, the parallel work is statically partitioned, and it is replicated instead of shared when data spans more than one partition.

6.2.4 x264 Communication Model

x264 is a lossy video encoder for high-quality streams. It receives a compressed or uncompressed video stream and encodes it using the H.264 standard. This application does not synchronize all threads as done by the other applications on the set; it 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 threads requested. The total number of stages created is $1 + 2 \times \mathit{videoframes}$. Besides, all mutexes variables are associated with condition variables. The condition variables are used to inform the threads of the encoding progress and to make sure that no data is accessed while it is not yet available.

6.3 Experimental Results

This section describes and discusses the experimental results, which are divided into four sections: (i) Subutai's area consumption and state-of-the-art comparison; (ii) single parallel application execution from PARSEC; (iii) multiple parallel application execution from the same benchmark; and (iv) micro-benchmark. The real applications depict the behavior of Subutai in our target application paradigm – the shared-memory paradigm. The micro-benchmark is employed to demonstrate some fundamental aspects of Subutai.

6.3.1 Subutai's Area Consumption and State-of-the-art Comparison

Subutai-HW comprises 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 an 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. Additionally, the memory requirement is negligible when compared to a typical processor cache (less than 10%, if the cache size is 16 KiB). Table 6.6 summarizes the synthesis results showing that our solution increases by 46% the basic NI area, including the local SPM. However, the overhead is amortized when the whole chip area is considered. Using Patel et al. [PKMS17] chip area of 400mm^2, the percentage of total area consumption of Subutai-HW is $\frac{64 \times 0.00632821}{400} = 0.0010\%$, while the enhanced NI is $\frac{64 \times 0.01976744}{400} = 0.0032\%$ for 64 cores.

The synthesis of Subutai-HW was achieved using Synopsis DC with an STMicroelectronics Silicon on Isolator (SOI) 28 nm technology and 1 GHz clock frequency. The SPM was synthesized with the same constraints and technology using Cut Explorer.

Table 6.6 – Synthesis results for Subutai-HW and SPM using 28 nm SOI.

Components	Area (μm^2)	Technology	Overhead
Basic NI	13 539.23	28 nm	–
Subutai FSM	2626.21	28 nm	19 %
SPM	3702	28 nm	27 %
Basic NI + Subutai-HW (FSM + SPM)	**19867.44**	**28** nm	**46** %

We compare our solution to those related works that provide enough data about the absolute area consumption (i.e., not in percentages) and technology used. Table 6.7 depicts the area consumption of five hardware-based solutions.

As the solutions target different architecture designs, we divided the total area consumed by the number of cores estimated in the system (i.e., area per core), as this technique provides a fairer analysis. In this case, Subutai is second-to-last in terms of area consumed per core in the system. Also, Subutai and HTM have an additional area requirement per core; i.e., HTM needs to change the first level cache 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. [AFA+12] has the overall least consumption as it is mainly comprised of wires and controllers. The last line of Table 6.7 shows the estimation of area consumption for a 400mm^2 chip [PKMS17] for the same set of related work. Subutai only consumes approximately 0.01% of the total chip. Once again, it is third-to-last in overall area consumption.

Table 6.7 – State-of-the-art area consumption.

	HTM [SDS08]	MCAS [PKMS17]	Abellán et al. [AFA+12]	Notifying Memories [MRSD16]	**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 %	**0.010** %

6.3.2 PARSEC Experimental Results

The PARSEC benchmark requires a user-defined value to determine the number of threads to spawn by each benchmark. The value defined may not be the same value used for the total number of threads, as these benchmarks may create additional threads [SR15] [Cat15]. Bodytrack, for instance, produces two extra threads for the value received by the user. Hence, to use 64 threads, the user has to request only 62 threads. This work always uses the total number of threads created.

6.3.2.1 Speedup of a Single Parallel Application

We have simulated the four PARSEC applications that comprise our application set (i.e., Bodytrack, Streamcluster, x264, and Facesim) in three application configurations (16, 32, and 64 threads) to visualize their behavior into our target architecture. For three out of four applications, they restrict their core usage to the number of threads; only x264 creates more threads than cores (as explained in Section 6.2.4). Therefore, these three applications also demonstrate our solution on architecture with lesser core count – 16 and 32 cores.

The results are organized in a series of figures and tables: Figures 6.5, 6.6, and 6.7 depict the total execution time for our application set for 16, 32, and 64 cores, respectively. As not all data can be visualized in these figures, due to the order of magnitude of the total execution time, we have tabulated the most important latencies in the nanoseconds order of magnitude for the same set of executions on Tables 6.8, 6.9, and 6.10 for 16, 32, and 64 cores, respectively. The table values are (i) processing time; (ii) sum of all waiting time for synchronization primitives (called synchronization wait henceforth); (iii) NoC time for Subutai

communication; and (iv) Subutai-HW time. For the SW-only solution, both (iii) and (iv) are handled by software; consequently, their values are computed together with (i) and (ii).

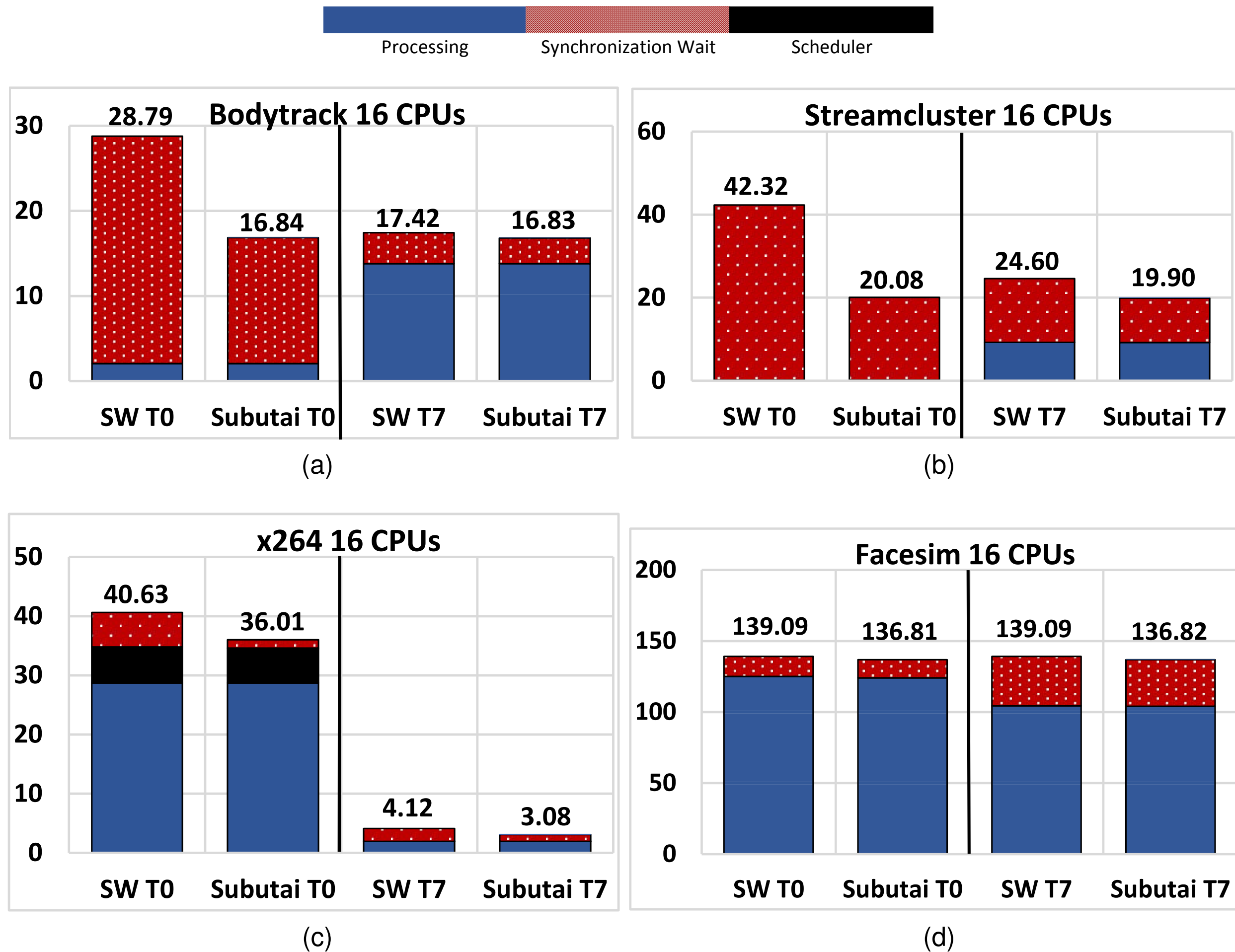


Figure 6.5 – The execution time in seconds (10^9 ns) of our application set on SW-only and Subutai solutions for 16 threads. Due to the order of magnitude, Subutai-HW and NoC latencies are not visible.

For fair comparisons, we consider the same scheduler (RR) for all applications. We 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). The results illustrate that our solution reduces the application total time by handling synchronization faster than its software counterpart handles. As stated earlier, Subutai does not affect the computational portion of the application.

For all applications, the master thread T_0 is responsible for four activities, in this order: (i) initializes synchronization variables; (ii) create worker threads; (iii) joins all of them (i.e., waits for all worker threads to finish); and (iv) executes post-parallel computation, if any. Activities (i) and (ii) should be as short as possible as they are done sequentially. Activity (iv) can be a post-parallel computation (e.g., parallel reduction), output generation, or nothing. Bodytrack, for instance, does not use post-parallel computation, as output generation is done for every frame processed; thus, it is executed inside the parallel region. Streamcluster also lacks post-parallel activities. Therefore, the majority of the execution time of the master thread

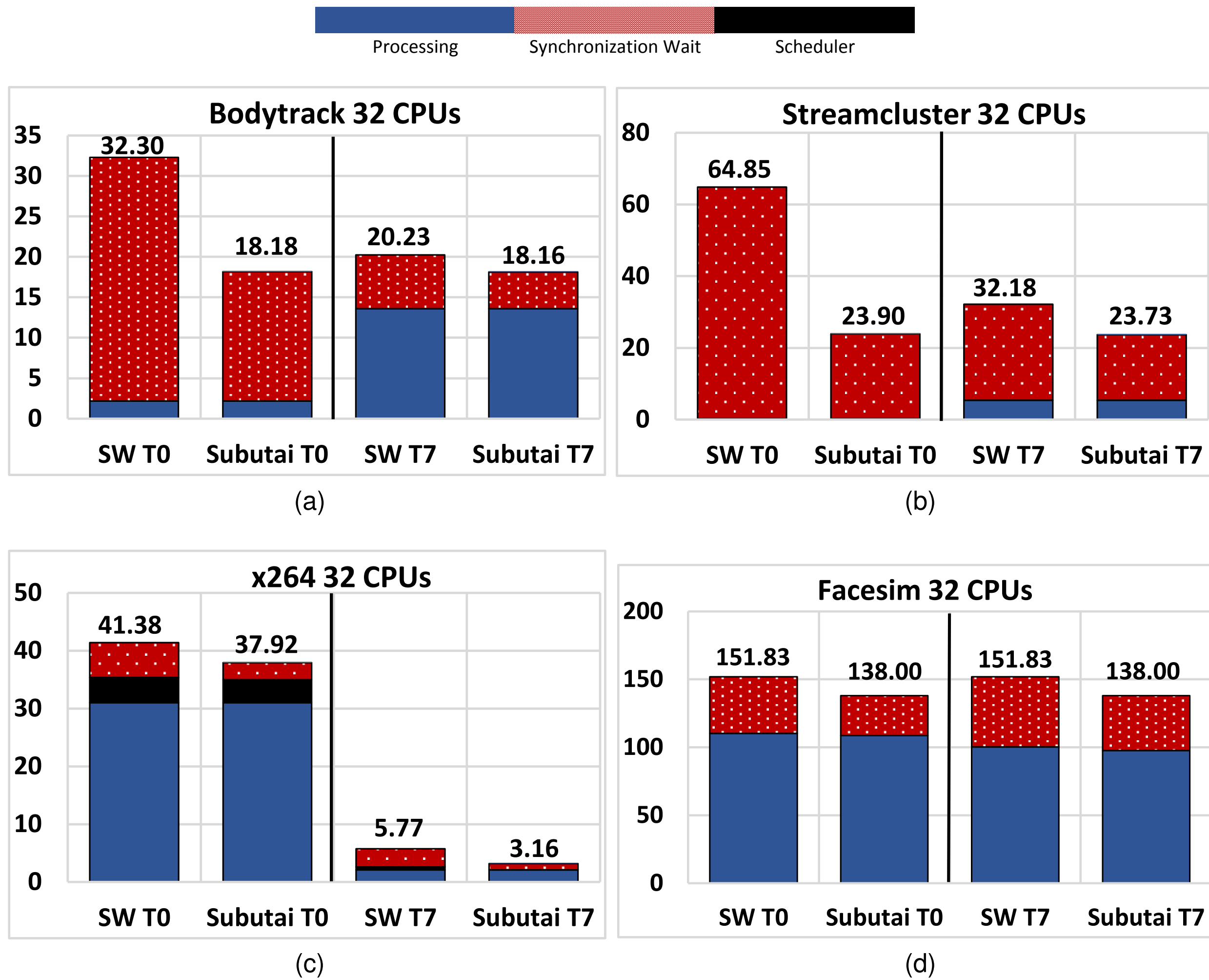


Figure 6.6 – The execution time in seconds (10^9 ns) of our application set on SW-only and Subutai solutions for 32 threads. Due to the order of magnitude, Subutai-HW and NoC latencies are not visible.

is consumed by activity (iii), waiting for all worker threads to finish. Consequently, by making the worker threads faster, the master thread is also accelerated.

In addition, from the designer point-of-view, the master thread (T_0) shows the effective speedup of the application, as it is responsible for initializing and finalizing the application. In other words, only when T_0 finishes, the application can be terminated. Therefore, Bodytrack achieved a speedup of 1.71×, 1.78×, and 1.77× for 16, 32, and 64 threads, respectively. Streamcluster achieved a speedup of 2.11×, 2.71×, 2.20× for the same set of threads, respectively. x264 achieved a speedup of 1.13×, 1.09×, 1.05× for the same set of threads, respectively. Finally, Facesim achieved a speedup of 1.02×, 1.10×, 1.27× for the same set of threads, respectively. On average, our solution achieved a speedup of 1.58× for the target application set without any changes to the application code.

The results also show that the application set is not scalable to 64 or even 32 threads. Southern et al. [SR15] have independently corroborated this limitation as well. Nonetheless, our solution works the same regardless of the application scaling.

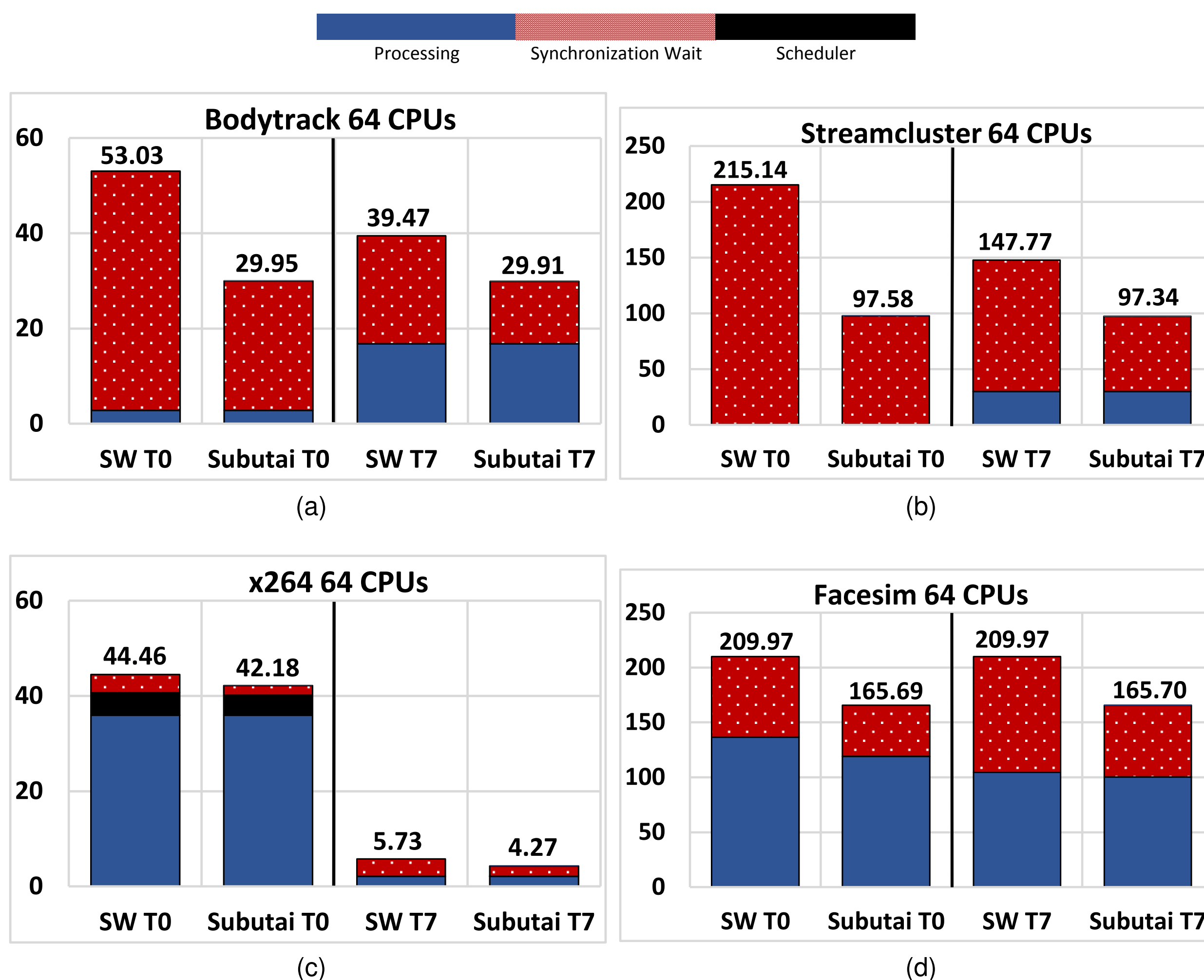


Figure 6.7 – The execution time in seconds (10^9 ns) of our application set on SW-only and Subutai solutions for 64 cores. Due to the order of magnitude, Subutai-HW and NoC latencies are not visible.

Next, we describe the application characteristics that explain the range of speedup values obtained with our solution.

Bodytrack and Streamcluster

We have decided to analyze Bodytrack and Streamcluster in a single place as they share interesting similarities. Both of them share the following characteristics: (i) they utilize barrier-based synchronization control; (ii) they utilize all synchronization primitives we support in this work; (iii) their 'master' thread is almost entirely controlled by the use of synchronization primitives.

The use of barrier-based control for synchronization heavily penalizes a parallel application. The reason for this penalization is two-fold: (i) a thread that enters a barrier is blocked until all other threads also joins the barrier; thus, this thread does not perform useful computation during the blocked time, decreasing the parallelism of the application; and (ii) as the barrier is released only after the last, and, therefore, the slowest, thread enters the barrier, the application always run on the worst-case scenario when employing barriers. In

Table 6.8 – Detailed execution time (ns) and the speedup for the application set executing on 16 cores.

Application	Type	16			
		Thread 0		Thread 7	
		SW-only	Subutai	SW-only	Subutai
Bodytrack	Processing*	2.1×10^9	2.1×10^9	14.0×10^9	14.0×10^9
	Synch. Wait	26.7×10^9	14.8×10^9	3.6×10^9	3.0×10^9
	Synch. NoC	–	0.6×10^4	–	1.0×10^4
	Subutai-HW	–	2.4×10^4	–	4.6×10^4
	Scheduler	–	–	–	–
	Total	28.8×10^9	16.9×10^9	17.6×10^9	17.0×10^9
	Speedup	1×	**1.71**×	1×	**1.04**×
Streamcluster	Processing*	0.1×10^9	0.1×10^9	9.2×10^9	9.2×10^9
	Synch. Wait	42.2×10^9	19.9×10^9	15.4×10^9	10.7×10^9
	Synch. NoC	–	0.3×10^4	–	30.1×10^4
	Subutai-HW	–	0.4×10^4	–	27.9×10^5
	Scheduler	–	–	–	–
	Total	42.3×10^9	20.0×10^9	24.6×10^9	19.9×10^9
	Speedup	1×	**2.11**×	1×	**1.24**×
x264	Processing*	28.7×10^9	28.7×10^9	1.9×10^9	1.9×10^9
	Synch. Wait	5.8×10^9	1.4×10^9	1.3×10^9	7.0×10^6
	Synch. NoC	–	5.9×10^5	–	5.3×10^3
	Subutai-HW	–	3.9×10^5	–	4.5×10^3
	Scheduler	6.1×10^9	5.9×10^9	1.5×10^6	9.7×10^5
	Total	40.6×10^9	36.0×10^9	4.1×10^9	3.1×10^9
	Speedup	1×	**1.13**×	1×	**1.32**×
Facesim	Processing	12.5×10^{10}	12.2×10^{10}	1.04×10^{10}	1.04×10^{10}
	Synch. Wait	13.9×10^9	12.8×10^9	34.7×10^9	32.8×10^9
	Synch. NoC	–	2.6×10^5	–	3.9×10^4
	Subutai-HW	–	3.3×10^5	–	5.7×10^4
	Scheduler	–	–	–	–
	Total	13.9×10^{10}	13.6×10^{10}	13.9×10^{10}	13.6×10^{10}
	Speedup	1×	**1.02**×	1×	**1.02**×

Synch. = Synchronization.
* The processing times of the software solution are higher than Subutai, but the differences are in order of 10^5 ns, which is insignificant from the order 10^9 ns.

other words, any unbalance of work distribution for working threads will generate sequential portions of execution, as threads with less work will finish before threads with more work (assuming a contention-free scheduling scenario, as is the case with the current Section).

The following figures demonstrate the latencies of all threads for each barrier call running with 64 threads: Figures 6.8a and 6.8b for Streamcluster, and Figure 6.9 for Bodytrack. Each colored point represents a single thread, and every barrier comprises 63 worker threads. A barrier is over (i.e., released) when all threads have joined into it; hence, when a given thread is the first to enter the same barrier again, it creates a new instance of the same barrier. These instances are given monotonic values to identify them, starting from 1 and going up to

Table 6.9 – Detailed execution time (ns) and the speedup for the application set executing on 32 cores.

App.	Type	32			
		Thread 0		Thread 7	
		SW-only	Subutai	SW-only	Subutai
Bodytrack	Processing*	2.2×10^9	2.2×10^9	14.0×10^9	14.0×10^9
	Synch. Wait	30.1×10^9	16.0×10^9	6.6×10^9	4.6×10^9
	Synch. NoC	–	1.4×10^4	–	1.0×10^4
	Subutai-HW	–	4.5×10^4	–	8.9×10^4
	Scheduler	–	–	–	–
	Total	32.2×10^9	18.2×10^9	20.6×10^9	18.6×10^9
	Speedup	**1×**	**1.78×**	**1×**	**1.11×**
Streamcluster	Processing*	0.1×10^9	0.1×10^9	5.4×10^9	5.4×10^9
	Synch. Wait	64.7×10^9	23.7×10^9	26.8×10^9	18.4×10^9
	Synch. NoC	–	0.8×10^4	–	35.4×10^4
	Subutai-HW	–	0.8×10^4	–	67.1×10^5
	Scheduler	–	–	–	–
	Total	64.8×10^9	23.8×10^9	32.2×10^9	23.8×10^9
	Speedup	**1×**	**2.71×**	**1×**	**1.36×**
x264	Processing*	31.0×10^9	31.0×10^9	2.1×10^9	2.1×10^9
	Synch. Wait	6.0×10^9	2.9×10^9	3.1×10^9	1.1×10^9
	Synch. NoC	–	1.1×10^6	–	8.1×10^3
	Subutai-HW	–	7.4×10^5	–	4.1×10^3
	Scheduler	4.4×10^9	4.0×10^9	5.4×10^8	1.1×10^6
	Total	41.4×10^9	37.9×10^9	5.8×10^9	3.1×10^9
	Speedup	**1×**	**1.09×**	**1×**	**1.87×**
Facesim	Processing	11.0×10^{10}	10.8×10^{10}	10.0×10^{10}	9.8×10^{10}
	Synch. Wait	41.7×10^9	29.4×10^9	51.6×10^9	40.3×10^9
	Synch. NoC	–	8.9×10^5	–	1.4×10^5
	Subutai-HW	–	6.1×10^5	–	1.1×10^5
	Scheduler	–	–	–	–
	Total	15.2×10^{10}	13.8×10^{10}	15.2×10^{10}	13.8×10^{10}
	Speedup	**1×**	**1.10×**	**1×**	**1.10×**

Synch. = Synchronization.
* The processing times of the software solution are higher than Subutai, but the differences are in order of 10^5 ns, which is insignificant from the order 10^9 ns.

b barriers where b is the total number of barriers created by the application[2]. In addition, the time plotted for each thread is relative to the time of the first thread to enter the barrier.

Figure 6.8b shows that the instances of barriers of Streamcluster have an average lifespan of a couple of milliseconds (i.e., 10^7 ns). The impact of it depends on the duration of the application: Figures 6.5b, 6.6b, and 6.7b show that Streamcluter execution ranges from 42 to 215 seconds (i.e., 10^9 ns). Therefore, the impact of a single instance should be fairly low. Unfortunately, Figure 6.8a shows that Streamcluster creates over 10^4 instances of barriers, which means that the application is limited by some barrier for almost its entire lifespan. Since

[2]For Streamcluster, we have removed the first instance of the barrier call as it has hundreds of milliseconds of latency, which would make Figure 6.8a harder to visualize.

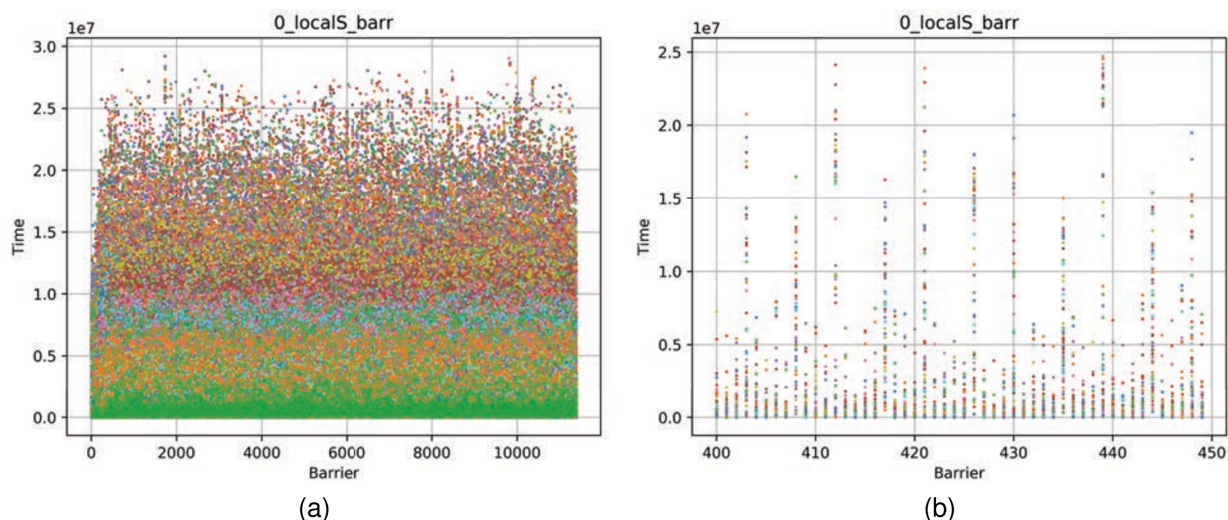
Table 6.10 – Detailed execution time (ns) and the speedup for Bodytrack and Streamcluster executing on 64 cores.

Application	Type	64			
		Thread 0		Thread 7	
		SW-only	Subutai	SW-only	Subutai
Bodytrack	Processing*	2.8×10^9	2.8×10^9	16.8×10^9	16.8×10^9
	Synch. Wait	50.2×10^9	27.1×10^9	22.7×10^9	13.1×10^9
	Synch. NoC	–	3.6×10^4	–	2.0×10^4
	Subutai-HW	–	8.0×10^4	–	17.9×10^4
	Scheduler	–	–	–	–
	Total (ns)	53.0×10^9	29.9×10^9	39.5×10^9	29.9×10^9
	Speedup	$1\times$	**1.77×**	$1\times$	**1.32×**
Streamcluster	Processing*	0.2×10^9	0.2×10^9	30.0×10^9	30.0×10^9
	Synch. Wait	214.9×10^9	97.3×10^9	117.7×10^9	67.3×10^9
	Synch. NoC	–	2.3×10^4	–	17.1×10^4
	Subutai-HW	–	1.7×10^4	–	1.6×10^7
	Scheduler	–	–	–	–
	Total (ns)	215.1×10^9	97.5×10^9	147.7×10^9	97.3×10^9
	Speedup	$1\times$	**2.20×**	$1\times$	**1.52×**
x264	Processing*	35.9×10^9	35.9×10^9	2.1×10^9	2.1×10^9
	Synch. Wait	3.8×10^9	2.1×10^9	3.7×10^9	2.2×10^9
	Synch. NoC	–	2.2×10^6	–	7.6×10^3
	Subutai-HW	–	1.4×10^6	–	4.0×10^3
	Scheduler	4.8×10^9	4.2×10^9	3.4×10^4	1.1×10^6
	Total (ns)	44.5×10^9	42.2×10^9	5.8×10^9	4.3×10^9
	Speedup	$1\times$	**1.05×**	$1\times$	**1.71×**
Facesim	Processing	13.7×10^{10}	11.9×10^{10}	10.5×10^{10}	10.0×10^{10}
	Synch. Wait	73.4×10^9	46.7×10^9	10.5×10^{10}	65.5×10^9
	Synch. NoC	–	2.3×10^6	–	1.4×10^5
	Subutai-HW	–	1.1×10^6	–	6.6×10^4
	Scheduler	–	–	–	–
	Total (ns)	21.0×10^{10}	16.6×10^{10}	21.0×10^{10}	16.6×10^{10}
	Speedup	$1\times$	**1.27×**	$1\times$	**1.27×**

Synch. = Synchronization.
* The processing times of the software solution are higher than Subutai, but the differences are in order of 10^5 ns, which is insignificant from the order 10^9 ns.

barriers limit the parallel execution of the code, as explained earlier, the application speedup potential is diminished significantly as the core count rises. This behavior is observed with the results from Figures 6.5b, 6.6b, and 6.7b.

Bodytrack differs from Streamcluster in three aspects: (i) it creates very few instances of barrier – only 46; (ii) it uses two barrier variables for the core workflow instead of one (both generating 46 instances); and (iii) the lifespan of barriers created with `workDoneBarrier` are over a order of magnitude (e.g., up to 10^9 ns) in comparison to the barriers generated in Streamcluster. The latencies of instances of the `workDoneBarrier` barrier are shown in Figure 6.9. Due to the order of magnitude of the latencies involved in the barriers, Bodytrack is also not able to scale to 64 threads, or even, 32 threads.



(a) (b)

Figure 6.8 – Relative time for every thread to reach the barrier `localS`; (a) is a plot for all barrier calls during the application execution, while (b) is a snippet of 50 barrier calls for better visualization.

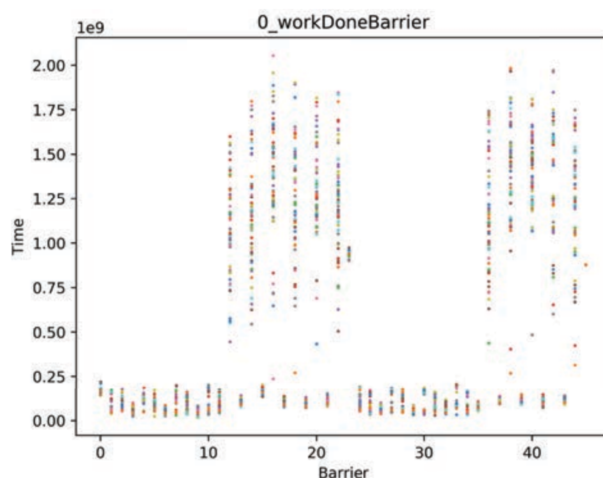


Figure 6.9 – Relative time for every thread to reach all barrier calls of `workDoneBarrier`.

The barrier procedure involves two operations, and Subutai accelerates both of them: (i) entering and (ii) exiting the barrier. Both operations use the same procedure, `pthread_barrier_wait`; thus, the functionality is decided by the procedure, and not by the developer (i.e., a thread will or will not be blocked on a barrier according to the runtime behavior of the application). As will be shown in Section 6.3.3, this procedure can achieve a speed up of 86$\times$ for releasing threads on Subutai compared to the SW-only. For item (ii), the operation switches the application from sequential (i.e., only a single thread has not reached the barrier and is executing) to fully parallel execution. Consequently, the speedup of this operation impacts significantly parallel applications that use it.

The acceleration for barrier operations on these applications are presented in the following figures: (i) Figure 6.10 for T_7 of Streamcluster for a subset of barrier calls (the same as shown in Figure 6.8b); and (ii) Figure 6.11 for T_7 of Bodytrack for all barrier calls,

respectively. In these figures, every instance of a barrier is comprised of two bars: the left and right one for SW-only and Subutai, respectively. They are the latency, expressed in 10^5 ns, experienced by the thread from the moment it calls the barrier procedure until execution is returned to the thread. We did not plot results for the 'master' thread, as well as our example worker threads, in these results for two reasons: (i) the 'master' thread of Streamcluster does not participate in the barrier; and (ii) it is not guaranteed that the 'master' thread will be the first or last thread to reach the barrier; thus, its behavior is similar to any other worker thread in this regard.

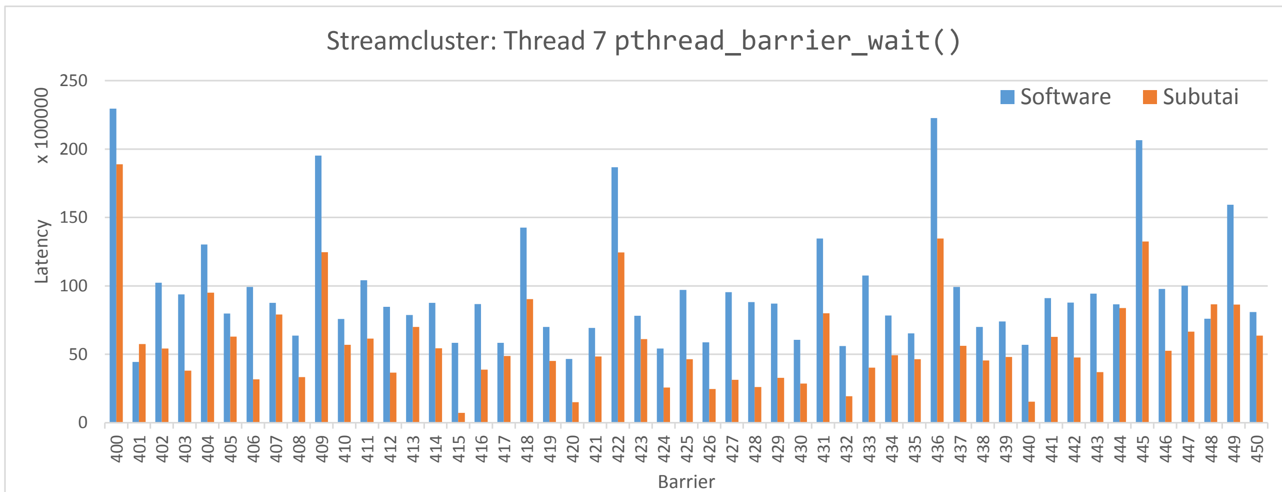


Figure 6.10 – Comparison of latency for the procedure `pthread_barrier_wait` for SW-only and Subutai. The results are for T_7 for a subset of barrier calls (400 up to 450).

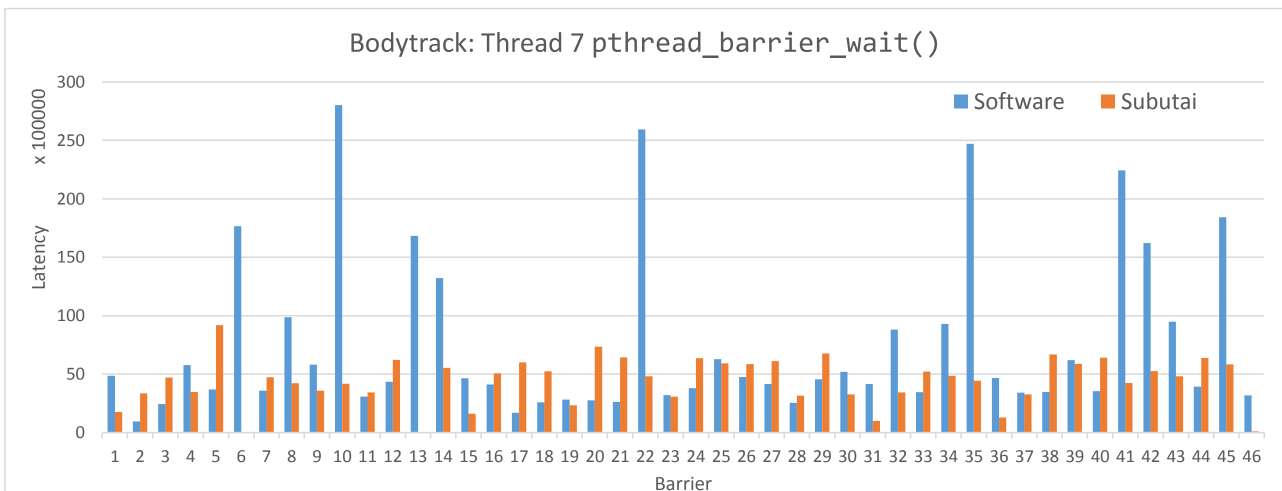


Figure 6.11 – Comparison of latency for the procedure `pthread_barrier_wait` for SW-only and Subutai. The results are for T_7 for all barrier calls of `poolReadyBarrier`.

The barrier operations executed faster for the worker thread T_7 of Streamcluster and Bodytrack on Subutai than on SW-only, in general, as shown in Figure 6.10 and Figure 6.11, respectively. Nevertheless, there are cases where the inverse happens: for instance, the latency of barrier 401 of Streamcluster and barrier 7 of Bodytrack. However, the sum of all

latencies is lower for Subutai in both cases. It is important to note that T_7 will not be the last thread to enter the barrier in all cases; thus, these values are not the total latency experienced for the barrier procedure. Instead, they are the latencies experienced for this thread only.

Nonetheless, we mention two related reasons for Subutai to be slower: (i) Streamcluster and Bodytrack use synchronization primitives in their core workflow. The latency for locks in high contention scenarios is dependent on how fast a thread can request the lock. For instance, a thread that participates in a lock disputed by 48 threads will execute earlier if it can arrive at the lock before the other requesting threads; otherwise, this thread is positioned at the end of the queue, which will demand more time to be processed; (ii) the PThreads standard [IEE16] does not enforce an order for the releasing process of a barrier; hence, SW-only and Subutai can release threads in any order. Thus, threads that are released later because of (ii) will execute slower because of (i). Therefore, these reasons justify situations where Subutai is slower than SW-only.

Subutai also accelerates mutex and condition synchronization primitives besides barriers. However, these primitives may or may not result in sequential execution: it depends if the application shares them on all threads. Bodytrack and Streamcluster share these primitives with all worker threads; hence, we are able to increase the parallelism once again by providing faster synchronization operations.

As a result, Streamcluster achieved a speedup of 2.11×, 2.71×, and 2.20×, for 16, 32, and 64 threads, respectively; and Bodytrack achieved a speedup of 1.71×, 1.78×, and 1.77×, for 16, 32, and 64 threads, respectively. The speedup difference is explained by the number of synchronization calls utilized by each application; Table 6.4 shows that Streamcluter requires, roughly, 18×, 23×, and 31× 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. Tables 6.8, 6.9, and 6.10 show, for all cases, the worker threads of Streamcluster achieving a higher speedup when compared to the worker threads of Bodytrack.

x264

The x264 application avoids the barrier-based control entirely. While increasing the complexity of the implementation compared to Bodytrack and Streamcluster, x264 can distribute the workload as soon as a single thread has finished its work, instead of waiting for threads to finish before distributing the workload. In other words, as long as there is work to be done, and cores are available (i.e., idle), x264 executes entirely in parallel. Therefore, it is much less susceptible to delays in a given thread, as the other threads are not dependent on the former speed; this was the case for the other two applications analyzed earlier.

x264 does not reuse worker threads; once a worker thread has finished, it is terminated, and a new worker thread is created in its place. This explains why our example worker thread T_7 has a short lifespan. Another interesting aspect of x264 is that worker threads show different runtime behavior among themselves. While Bodytrack and Streamcluster

have more or less the same behavior of all worker threads[3], every x264 worker thread has a distinct behavior. As illustrated in the experimental results, some of the threads have almost no synchronization primitive usage at all, and others are predominantly dominated by this mechanism.

Additionally, x264 does not use mutexes for accessing shared data. When a worker thread needs information from another, they communicate through condition variables. The communication does not happen very often when compared to Bodytrack and Streamcluster, as shown in Table 6.4. The decrease of synchronization usage improves the parallel performance of the application; unfortunately these are the events that Subutai can accelerate. Since they happen less often, Subutai is limited in its ability to accelerate this application.

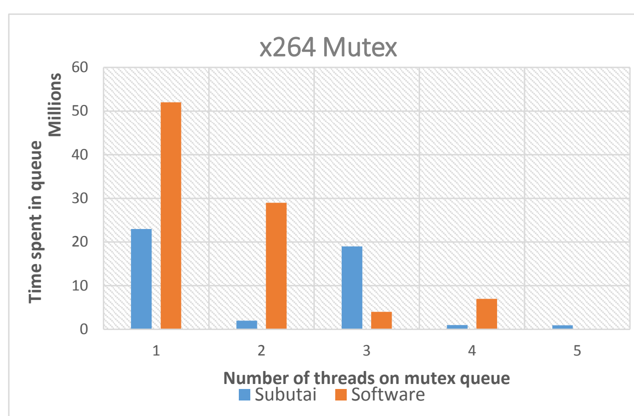
Table 6.4 also shows that x264 uses approximately a hundred of conditions, while Bodytrack and Streamcluster are limited to half a dozen. Consequently, the contention on synchronization variables is decreased for x264. As an example, we demonstrate the contention for a mutex running with 64 threads in Figure 6.12a and Figure 6.12b for x264 and Bodytrack, respectively. These figures plot the time spent on the mutex queue in 10^6 ns (Y-axis) for a given number of threads (X-axis). For instance, for a value of 1 on the X-axis, the Y-axis shows the aggregate time that the application spent on this mutex with one thread on the mutex queue. Thus, as the X-axis increases, so does the contention on the mutex.

Figure 6.12a shows that the contention for the x264 mutex reached 5 threads, while the maximum is 63 threads (the missing 64th thread must be the owner); therefore, the contention is low. Figure 6.12b depicts that the contention for Bodytrack reaches 61 threads, while the maximum is 62 threads (the 'master' thread does not participate in this mutex). For both x264 and Bodytrack, Subutai can accelerate the queue manipulation, and the total acceleration is presented in Figure 6.12c and 6.12d, where the total time spent on the mutex queue is compared with SW-only. While the SW-only implementation has fast operations for the mutex queue [ZK18], it still requires locking and is susceptible to cache misses. Our hardware implementation has exclusive access for its memory (SPM), without the need for locking, and execute operations on it with a latency of 1 ns (Section 4.2).

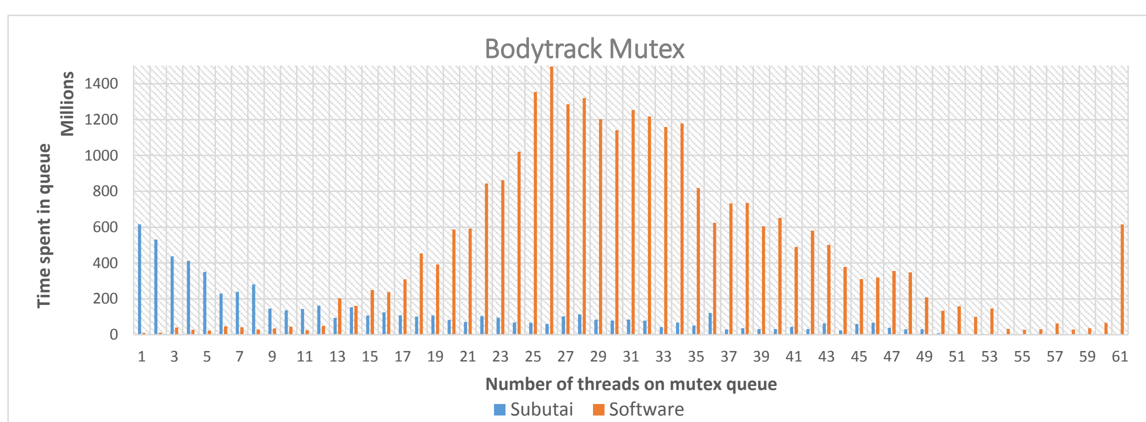
The speedup achieved with Subutai for x264 is 1.13×, 1.09×, and 1.05× for 16, 32, and 64 threads, respectively. We cannot schedule all worker threads perfectly due to the implementation of x264 as it creates more threads than the number requested by the user. Therefore, x264 was the only one affected by the scheduling of other threads on the single application execution.

Another aspect is that we were not able to accelerate all worker threads, as some of them have almost no synchronization usage. Nonetheless, threads that use synchronization mechanisms are accelerated, as shown by the speedup of our example worker thread: 1.32×, 1.87×, and 1.71×, for 16, 32, and 64 threads, respectively.

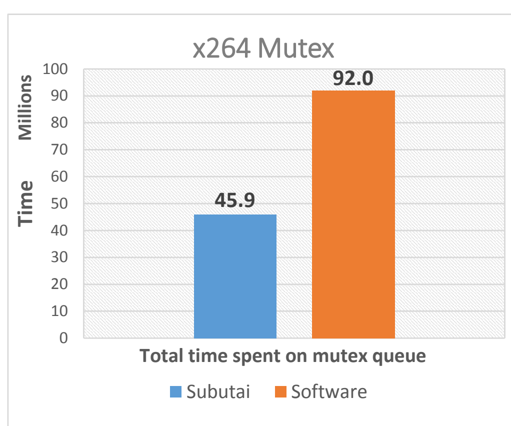
[3]They are not the same due to the use of synchronization primitives and an unbalanced distribution of work.

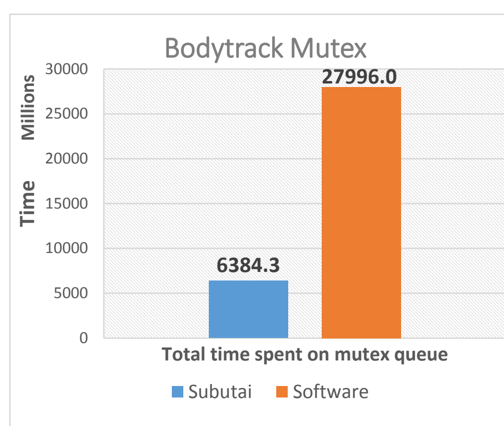


(a)



(b)





(c) (d)

Figure 6.12 – Time spent, in 10^6 ns, on the mutex queue of Bodytrack and x264. (a) and (b) are plots of time spent (Y-axis) by the number of threads on the queue (X-axis) for x264 and Bodytrack, respectively; (c) and (d) are the total time spent for the same application set.

Facesim

Facesim does not employ barrier-based control; instead, it uses condition-based control to synchronize all threads. Such behavior was not found in the set of other applications. Synchronizing all threads would make Facesim susceptible to delays in worker threads, as Bodytrack and Streamcluster are, but because Facesim implements work-stealing abilities,

the worker threads can continue to work in parallel even when their local work has finished, as long as there is remote work available. Consequently, Facesim can maintain parallel execution even for an unbalanced distribution of workload or due to scheduler contention.

Like x264, Facesim use more than a dozen of synchronization primitives. In fact, Facesim is the only application of our set that scales the number of primitives with the number of threads. As discussed in the x264 section, increasing the number of primitives decrease the contention on them.

The comparison of Facesim and Bodytrack presents an interesting aspect of Subutai. From Table 6.4, it is clear that Facesim requires more synchronization operations. Since these operations are the basis of application acceleration from Subutai, it would be expected that Subutai speeds up Facesim more than Bodytrack; however, this is not the case for any number of threads analyzed in this work. Therefore, while the number of synchronization operations usage is an important factor, it should be combined with a second factor: the contention of synchronization primitives. As discussed in the x264 application, queue manipulation is a source of acceleration for Subutai. Facesim also does not have the same contention of synchronization primitives as Bodytrack does. The reasons for this are the same as the ones for x264: (i) the requirements of the application; and (ii) the use of many synchronization primitives instead of a dozen.

Finally, Facesim is the most demanding of our application set for processing information computationally. Tables 6.8, 6.9, and 6.10 showed that the worker threads of Facesim have an order of magnitude higher for processing than any other application (i.e., 10^{10} ns against 10^{9} ns). Additionally, for the entire runtime of the application, only Streamcluster with 64 threads demanded more time (215.14 against 209.97 seconds). Yet, Figure 6.7 shows that Streamcluster is dominated by synchronization operations, while Facesim has the majority of its time dedicated to processing. As Subutai does not accelerate the computational portions of a parallel application, Facesim does not beneficiate from it as the other applications.

For the reasons discussed here, Facesim achieved a speedup of 1.02×, 1.10×, and 1.27× for 16, 32, and 64 threads. The speedup increases with the number of threads since the application scale the use of synchronization operations with it; thus, it is these operations that are accelerated.

6.3.2.2 Speedup of Multiple Parallel Application Execution

Figure 6.13 depicts the experimental results organized into sets of applications using 64 threads. Figures 6.13a, 6.13b, 6.13c illustrate eight instances of Bodytrack, Streamcluster, and x264, respectively. Figure 6.13d shows three instances of Facesim, and Figure 6.13e is a combination of instances, three of Bodytrack, three of Streamcluster and two of x264.

Figure 6.13 depicts the entire execution time in seconds of an application set (i.e., from initialization to termination of all applications). These figures compare three types of

Figure 6.13 – Execution in seconds (10^9 ns) for multiple application sets (lower is better) (Exec = Execution).

schedulers: RR, CSA, and a One Application at a Time (OAT) scheduler. The latter one is used for representing a mono application system (i.e., it runs one application). Lines `b` and `c` of the set of figures display that Subutai decreases the execution time compared to SW-only even in a competitive scheduling scenario.

The results pertaining to x264 and Facesim require clarification. For x264, according to Table 6.5 and the description of the application set (i.e., eight instances), it seems that it is not possible to run eight instances of x264 due to the area limitation of Subutai-HW.

Subutai-HW handles 4 synchronization primitives per hardware instance, and since we use a 64-core system, it handles 256 primitives in total (Section 4.2). However, each instance of x264 and Facesim require 190 and 67 primitives, respectively, per Table 6.5; therefore, these applications utilize 1520 and 536 primitives in total. In reality, x264 works as expected because the primitives are not created at the same time. Consequently, x264 does not need all primitives at the same time. Facesim, however, does require all primitives at the same time; hence, we decided to limit the execution of Facesim to three instances, because such a scenario is handled entirely by Subutai-HW (i.e., 201 synchronization primitives). Chapter 7 presents and discusses other techniques that enable the execution of eight instances of Facesim with Subutai-HW.

OAT Scheduler

Lines a of Figures 6.13a, 6.13b, 6.13c, 6.13d, and 6.13e depict the gains of running Subutai compared to an SW-only implementation with an OAT scheduler. These results are similar to the single application acceleration discussed in the previous section, as only one application is executed at a time. However, there are two differences: (i) we ran multiple instances of the same application instead of one; and (ii) we ran a mixed application set that is not present in the single application section. They achieved a speedup of 1.86×, 2.15×, 1.08×, 1.26×, and 1.91× for a set of Bodytrack, Streamcluster, x264, Facesim, and mixed applications, respectively.

RR Scheduler

The speedup achieved with Subutai while running with a RR scheduler was 1.58×, 2.56×, 4.61×, 1.39×, and 2.08× for a set of Bodytrack, Streamcluster, x264, Facesim, and mixed applications, respectively.

CSA scheduler

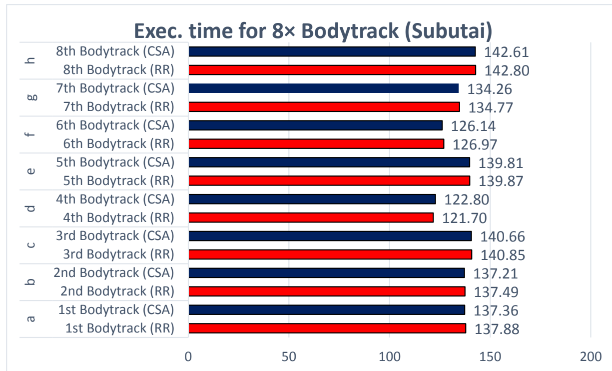
The speedup achieved with Subutai while running with the CSA policy was 1.58×, 2.70×, 4.61×, 1.43×, and 2.09× for a set of Bodytrack, Streamcluster, x264, Facesim, and mixed applications, respectively.

Discussion

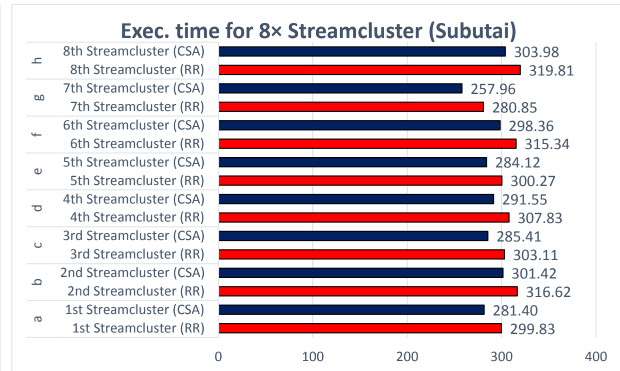
While we have presented the speedup values for Subutai, the figures present gains for SW-only as well; yet, the execution time of SW-only is always higher (i.e., worse) compared to Subutai for the set of applications analyzed here. In fact, for the Streamcluster and mixed applications set (Figures 6.13b and 6.13e), running them on Subutai with an OAT scheduler was faster than running them on SW-only with either scheduler policies used in this work.

The CSA policy speeds up the execution time compared to the baseline RR for all cases of our application set, except x264, where the execution runtime is maintained the same as with RR. Figure 6.14 details the execution time for each application running with CSA and RR on Subutai. The set of Streamcluster with CSA presented the highest speedup when compared to the same set of the application with the RR scheduler, 1.05×, followed by

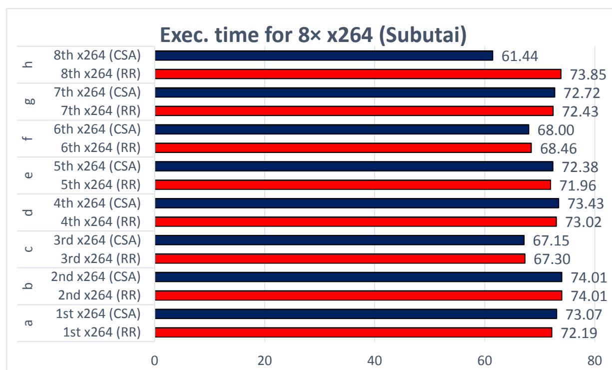
Facesim (1.03×). Bodytrack and x264 presented a less significant speedup of less or equal to 1.01×. However, Subutai against SW-only using the same scheduler (CSA) on both had a speedup of 1.40×, 4.05×, 1.19×, and 2.40× for Bodytrack, Streamcluster, x264 and mixed application, respectively. Facesim had a speedup of less than 1.00× for the same setup.



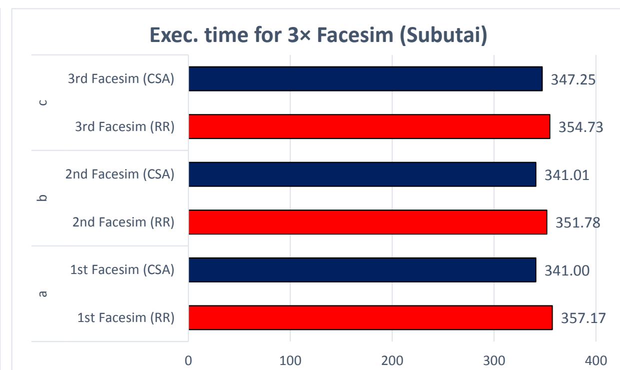
(a) Bodytrack set (Subutai RR × Subutai CSA).



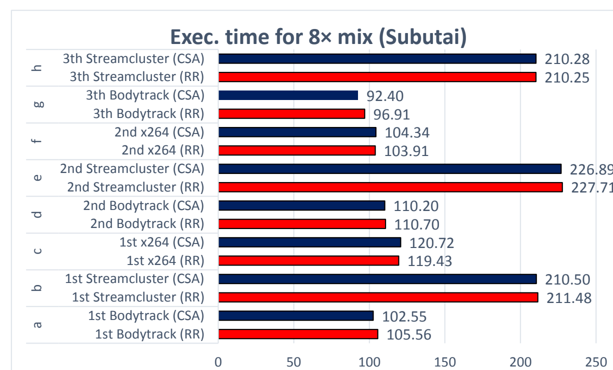
(b) Streamcluster set (Subutai RR × Subutai CSA).



(c) x264 set (Subutai RR × Subutai CSA).



(d) Facesim set (Subutai RR × Subutai CSA).



(e) mix set (Subutai RR × Subutai CSA).

Figure 6.14 – Execution in seconds (10^9 ns) for multiple application sets (lower is better) (Exec = Execution).

There are a few reasons why the speedup achieved using CSA is limited. Firstly, there is a fragile balance of accelerating one application in place of others. For instance, some applications on the SW-only implementation (e.g., x264) had higher execution time by using the CSA policy instead of pure RR. We aim to avoid such scenarios by enforcing

the *CSALimit*; however, they can still occur for some applications[4]. In juxtaposition, the use of *CSALimit* restricts the CSA potential for accelerating applications. Secondly, CSA relies on the premise that accelerating critical sections decreases the overall execution runtime. 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 are blocked waiting for the slowest thread to join the barrier). However, other applications, such as x264 and Facesim, can start working on new data as soon as the first thread has finished. Therefore, CSA has a lesser impact on such applications. Thirdly, CSA only directly accelerates the mutex synchronization primitive. Indirectly, accesses to conditions are also accelerated due to the use of mutexes. As CSA does not accelerate all synchronization primitives, sequential execution can still occur (e.g., barriers).

Table 6.11 aggregates information regarding the execution of the application set on a CSA-enabled scheduler. The table provides four types of information for an application: (i) the number of scheduler events without a critical section; (ii) the number of scheduler events with a critical section, and the CSA policy is in effect; (iii) same as (ii), but the CSA is deactivated due to *CSALimit*; and (iv) the average of the sum of all critical sections for all worker threads.

Table 6.11 – Characteristics of the application set executing on CSA (CS = Critical Section; all data are averages achieved for all applications).

Application set	**Schedule events (not CS)**	**Schedule events (CS)**		**Avg Sum of CS**[1]
		CSA (CS)	**RR (CS)**	
Bodytrack ×8	323379	15274	1274	145.23 ms
Streamcluster ×8	1292934	3417	0	1.71 ms
x264 ×8	203047	18856	0	17.83 ms
Facesim ×3	5196538	378250	52115	3355.40 ms

1 The average of the sum of all critical sections for all worker threads.

Table 6.11 shows the percentage of critical sections according to the scheduling events: 5.12%, 0.26%, 9.29%, and 8.28% for Bodytrack, Streamcluster, x264, and Facesim, respectively. These percentages did not translate directly to better potential for acceleration on the CSA policy. Streamcluster, which has the least amount of critical section in percentage points, achieved the highest speedup with CSA in the examined sets. As we discussed previously, the synchronization workflow is a major factor for speeding up the application, and the number of critical sections alone does not capture it.

Another aspect that provided disparate behavior for the application set is the sum of all critical sections for worker threads: the values range from a couple to thousands of milliseconds. Bodyrtack and Facesim, which are the two most demanding applications in

[4]For our evaluation, the execution time for an application set is determined by the last application to finish; thus, it is not enough to accelerate one of the applications in the set.

terms of average duration of critical section, are limited by *CSALimit*, while the other two applications are never limited by it. Consequently, Bodytrack and Facesim may be further accelerated by a limitless CSA; however, as we target a fair scheduler, a limitless CSA would make the application perform worse regarding the scheduler fairness.

Table 6.12 presents the unfairness experimental values obtained in this work. For all cases, CSA either maintains or decreases the unfairness of the scheduler for the application set, except for x264 (SW-only and Subutai) and Facesim (SW-only).

Table 6.12 – 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
Facesim ×3	1.23	1.01	1.01	1.02
mix ×8	2.00	1.71	1.88	1.83

Bodytrack and Streamcluster maintain the same fairness using CSA, expect for Bodytrack with Subutai: in this case, the fairness increases (i.e., lower unfairness). x264 increases fairness for SW-only but decreases fairness for Subutai. It is visually perceivable why CSA has a higher unfairness for Subutai in this case from Figure 6.14c (line `h`): the 8th instance of x264 has a boost of performance compared to the pure RR policy (from 73.85 to 61.44 seconds). While accelerating a single instance is interesting, our objective was not to burden the other instances while accelerating the total execution time. Thankfully, the objective was achieved in this case even though the unfairness metric increases: the overall execution time remains the same for CSA compared to RR (Figure 6.13c).

The SW-only execution of Facesim illustrastes that using a pure RR scheduler does not necessarily produce a fair distribution of resources on parallel applications. Due to the adverse effects of critical sections on parallel execution performance, the preemption of threads that are inside a critical section has a significant impact on the performance. Table 6.11 displays that moving from RR to CSA diminishes the unfairness from 1.23 to 1.01 on this application. In other words, the instances of the application have approximately the same execution time on CSA as the critical section duration remains to a minimum. For Subutai, the unfairness went from 1.01 to 1.02 using RR and CSA, respectively. Nonetheless, the total execution time of Facesim was shorter on Subutai than on SW-only.

6.3.2.3 Neocondition

Our application set from PARSEC (i.e., Bodytrack, Streamcluster, x264, and Facesim) was adapted to use neocondition according to the procedure described in Section 5.2.3. On

the one hand, the procedure was straightforward for three out of four applications, as they exclusively use the mutex for the condition handling. On the other hand, Facesim employs the associated mutex for protecting a shared global variable. Although it would be possible to disassociate the mutex from the condition, we believe this is a change of the application design, and such action has been excluded from this work for compatibility with legacy code.

Figure 6.15 shows the latency reduction in nanoseconds of the total execution time for three PARSEC applications using neocondition. This reduction is based on the comparison of the baseline execution with Subutai, as presented in Figure 6.7, against the same execution with neocondition running with 64 threads. As expected, decreasing mutex calls by employing neocondition reduce the execution time.

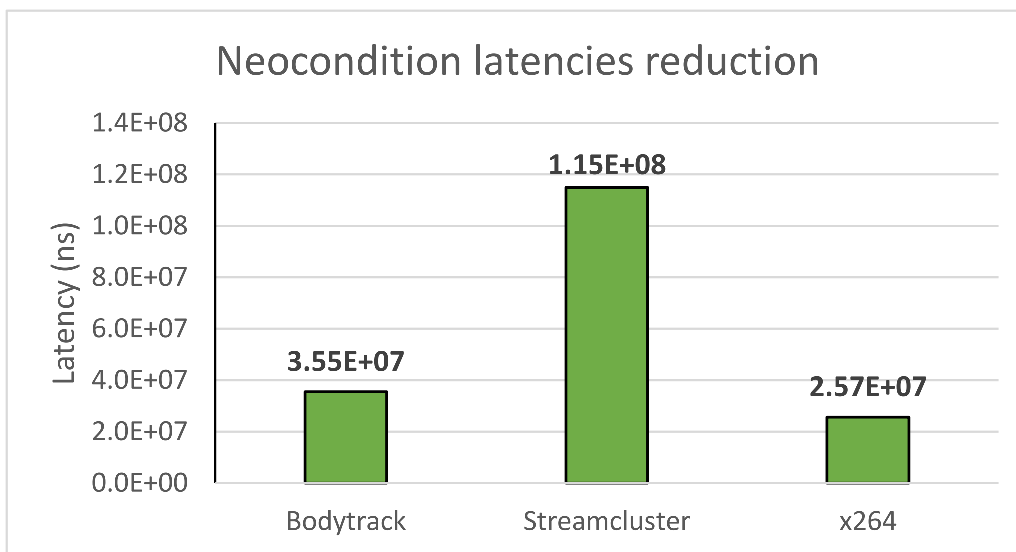


Figure 6.15 – Latency reduction in nanoseconds of the total execution time for three PARSEC applications.

As the applications from our set require 10^9 ns of total execution time, and the reduction by neocondition is in the range from 10^7 to 10^8 ns, the speedup of them was limited to less than 1.01×. Additionally, these applications do not execute more than a couple of hundred condition calls (Table 6.4). Still, serialization for accessing conditions has been eliminated while the applications maintained the same communication model.

6.3.3 Micro-benchmark

The set of 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 employ at least tens of thousands of synchronization primitives during its execution. Consequently, we developed a micro-benchmark to demonstrate individual aspects of Subutai.

The producer-consumer problem is a classic example of a synchronization problem. It describes two processes, where the producer and consumer share a common structure to share data. The producer generates data, shares it in the structure, and repeats such a process until all data has been processed. Simultaneously, the consumer consumes the shared data (i.e., removing it from the shared structure). The problem is two-fold: (i) make sure the producer does not add data into the structure if it is full and (ii) make sure the consumer does not remove data from an empty structure. The solution is a synchronization mechanism.

Listings 6.1 and 6.2 show a solution using two conditions and one mutex using the PThreads synchronization library. Initialization and error-checking have been omitted to simplify the code design. The shared structure (Listing 6.1) is a queue declared on line 6. Lines 4 and 5 determine the queue size and the value to represent an empty position, respectively. The three synchronization variables are declared in lines 7 to 9. The producer works as follows. First, the producer acquires the mutex and checks if the previous data has been consumed (lines 9 and 10): if it is not, it waits for the consumer thread (line 11). Then, it inserts the data in a new position and signals the consumer thread (lines 12-14). During all operations, besides waiting, the producer owns the lock. The consumer follows a similar logic.

Listing 6.1 – Shared structure for the producer-consumer application.

```
1  #include <pthread.h>
2  #include <stdint.h>
3
4  #define WORKSZ          1024
5  #define EMPTY_WORK      0
6  int32_t                 work_q[WORKSZ];
7  pthread_mutex_t         *mu;
8  pthread_cond_t          *c_empty;
9  pthread_cond_t          *c_full;
10
11 /* assumes queue has space */
12 void
13 put(size_t pos);
14
15 /* assumes queue has valid work in pos */
16 int32_t
17 get(size_t pos);
```

Both consumer and producer share the same mutex for controlling the conditions and the shared data. Therefore, when any of these threads block (lines 11 and 26), it is imperative

Listing 6.2 – A producer-consumer solution based on synchronization provided by PThreads. Synchronization procedures are colored red (based on [ADAD15]).

```
1  #include <pthread.h>
2  #include <stdint.h>
3
4  void
5  producer(void)
6  {
7          size_t          i;
8          for (i = 0; i < WORKSZ; i++) {
9                  pthread_mutex_lock(mu);
10                 while (i != 0 && work_q[i - 1] != EMPTY_WORK)
11                         pthread_cond_wait(c_empty, mu);
12                 put(i);
13                 pthread_cond_signal(c_full);
14                 pthread_mutex_unlock(mu);
15         }
16 }
17 void
18 consumer(void)
19 {
20         size_t          i;
21         uint32_t        recv;
22
23         for (i = 0; i < WORKSZ; i++) {
24                 pthread_mutex_lock(mu);
25                 while (work_q[i] == EMPTY_WORK)
26                         pthread_cond_wait(c_full, mu);
27                 recv = get(i);
28                 work_q[i] = EMPTY_WORK;
29                 pthread_cond_signal(c_empty);
30                 pthread_mutex_unlock(mu);
31                 /* work on recv */
32         }
33 }
```

that the lock is not held; otherwise, this can result in a deadlock scenario. Fortunately, PThreads deal with this internally, releasing the lock before blocking and reacquiring before returning to the user application.

Listings 6.1 and 6.2 are a simple implementation to produce and consume a single item. A more efficient implementation can handle sequences of items. Also, it is possible to use different synchronization mechanisms to share data (e.g., atomic operations and barriers).

We employ multiple implementations of the producer-consumer problem to benchmark the performance of different synchronization primitives. We developed three versions of one producer and many consumers design based on three essential primitives: mutex, barrier, and condition. Such a problem allows us to trace the performance of a single primitive at a time. Table 6.13 shows the average absolute time of SW-only and Subutai for these primitives.

Table 6.13 – Results for one producer and many consumer applications running with six threads.

Synchronization	Event type	Avg.[1] SW-only (ns)	Avg.[1] Subutai (ns)
Mutex	Lock Empty	1537	127
	Lock Queued	64178	916
	Unlock	4400	60
Barrier	Wait (released)	102467	1183
Condition	Broadcast	25209	60
	Queued	42844	1022

[1] Avg = Average.

Subutai significantly speeds up 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 scenarios have no response packet for Subutai; consequently, Subutai can return to the application immediately after the request packet is sent. Thus, the processing is offloaded to the hardware, and the primitive is handled faster from the caller perspective.

The SW-only implementation depends on the following costs to handle synchronization primitives: (i) context switching; (ii) synchronization for queue operations; and (iii) kernel space switching. Item (i) is reduced in Subutai by using a distributed OS. As mentioned in Section 4.1, we exploit a decentralized and distributed scheduler for thread manipulation. Additionally, as shown in Section 3.1.1, every group of PThreads handled by Subutai needs thread manipulation for blocking. Item (ii) is reduced by offloading all queue operations to hardware. Finally, item (iii) is not present in our OS; on the other hand, Subutai adds the cost of I/O operations to deal with Subutai-HW, which is not present in the software solution. Nonetheless, these factors explain the gains shown in Table 6.13.

7. CONCLUSIONS

Fancy thinking the Beast was something you could hunt and kill!
You knew, didn't you? I'm part of you? Close, close, close!
I'm the reason why it's no go? Why things are what they are?

The Beast from The Lord of the Flies by William Golding

There exist a broad set of works in contemporary research that addresses data synchronization with the objective of reducing the cost of synchronization for modern applications. These works typically face the same unique set of problems: (i) the burden of understanding and implementing the new concepts are left entirely in the hands of the developers; (ii) the solution requires modification to the application design and source code; and (iii) the improvements of the work are applicable only for a subset of the essential synchronization mechanisms. This set of problems limits the use of new techniques for to-be-developed applications while disregarding existing parallel applications. In this way, they write off the possibility of reusing existing parallel software code. As stated by McKenney, locking in research is often considered to be the worst villain of parallel programming, yet, paradoxically, it is widely employed [McK19a]:

> *"In recent concurrency research, the role of villain is often played by locking. In many papers and presentations, locking stands accused of promoting deadlocks, convoying, starvation, unfairness, data races, and all manner of other concurrency sins. Interestingly enough, the role of workhorse in production-quality shared-memory parallel software is played by, you guessed it, locking."*

In the context of reducing the cost of data synchronization, we saw that even small percentages of sequential execution could significantly diminish the achievable speedup of parallel applications (viz. Figure 1.1). Also, increasing parallelism tends to be directly proportional to increasing code complexity to deal with the consequences of the former increase (e.g., race conditions, stale data, livelocks). Therefore, moving to novel synchronization solutions is a non-trivial task.

This Thesis addresses the three problems raised earlier by creating a faster synchronization library that does not require modification on the application's source code. Thus, existing parallel applications can make use of our solution with little cost, and novel applications can be developed using the same methodology already used in production-quality parallel applications: lock-based designs. This chapter summarizes the original contributions of this Thesis, discusses final remarks and directions for future work.

7.1 Contributions of this Work

During this Thesis, the Author collaborated directly with Grupo de Sistemas Embarcardos (GSE) from Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) in Porto Alegre/Brazil and Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance (Lab-STICC) from Université Bretagne-Sud (UBS) in Lorient/France. This section explores the contributions of this Thesis developed in these research groups. We list three major contributions of this Thesis:

1. **Definition of a novel solution for data synchronization, namely Subutai** — Subutai is a hardware/software approach to speed up parallel applications without modifying the application design and source code. For achieving acceleration while avoiding changes to the application, Subutai modifies the implementation of the underlying synchronization primitives employed by the application. We choose the PThreads library as a reference for this work, yet, several libraries can also be accelerated. Subutai comprises three components: (i) user space library that overwrites the underlying synchronization API; (ii) Subutai-HW, which is an enhanced NI with access to a scratchpad memory; and (iii) an OS driver for communicating with Subutai-HW.

   We demonstrated that Subutai has practical value by accelerating parallel applications provided by the PARSEC benchmark. We employed four applications from PARSEC, a computer-vision, data-mining, video-encoding, and face-simulation application named Bodytrack, Streamcluster, x264, and Facesim, respectively. They achieved a speedup of 1.57$\times$, on average, compared to the same architecture (16, 32, and 64 cores) executing an entire software solution. The speedup achieved is for the entire applications running from start to finish.

2. **Development of a novel NI architecture, namely Subutai-HW** — We have enriched an existing NI implementation with the ability to handle dynamic double-linked queues and dealing with an external memory. The double-linked queues were used to control the three essential data synchronization procedures of PThreads: mutex, barrier, and condition. As such, the local memory relieves the processor memory and cache from handling the data, while saving valuable space in the cache. We compressed the software structure by approximately 90% compared to the glibc implementation.

   The new NI architecture can be integrated into any existing NI with two sources of overhead: (i) FSM to handle the double-linked queues; and (ii) access to a scratchpad memory. In this work, these circuits represented a limited area overhead of 19% and 27% on 28 nm SOI technology for (i) and (ii), respectively.

3. **Two extensions for Subutai: CSA and neocondition** — We have proposed two extensions built on top of the essential components of Subutai. Firstly, we designed

the CSA scheduler policy to accelerate parallel applications in a highly-contended scheduler scenario while maintaining the fairness of the scheduler. Secondly, we propose neocondition, a reimplementation of PThreads conditions that avoids the serialization of access to conditions through the removal of the associated mutex. They provide performance benefits while increasing the cost of adoption. The former is done entirely in software, while the latter requires changes to software and hardware. Subutai will work even in the absence of these extensions.

The CSA policy had a speedup ranging from approximately 1% up to 5% compared to a baseline RR policy. While the speedup is limited, we highlight that the scenarios explored in this work require hundreds of seconds of execution; thus, the speedup removes up to a couple of seconds of execution. Additionally, we demonstrated that the scheduler fairness was not affected negatively by our policy. Meanwhile, neocondition was ported to three out of four of our application set and reduced the execution time ranging from 10^7 up to 10^8 nanoseconds with minimal changes on the application's source code.

7.1.1 Other Contributions

This Author has collaborated in other academic work at PUCRS as well as UBS in the same research laboratory as he was enrolled. At the start of the Ph. D. at PUCRS, this Author published papers related to memory technology, latency, and security in NoC-based systems [CKF+16b] [FSS+16] [CKF+16a] [FMC+16]. Additionally, this Author worked on the hardware design and experimental results for video encoding techniques on the HEVC standard in the course of the Ph. D. duration [SCF+16] [FSC+18a] [SFC+18] [FSC+18b] [SSF+19].

At UBS, the Author worked on a novel hybrid wireless/wired NoC intended for parallel computing. Subutai was extended to support such networks and benefit from multicast and broadcast messages (i.e., NoCs typically exclusively use unicast packets, and that was the case for this work). Then, the NoC was ported to Noxim for energy consumption analysis. The exploration of this research appeared in [MCM+18] [KCM+18] [CKMCD19].

7.2 Discussion and Future Work

Parallel applications are indispensable for current and future systems, as the chip parallelism is a reality, and the applications must be written to use it. However, they require specialized knowledge for their implementation, and programming languages do not commonly resolve concurrency issues at compilation or even at runtime. While debugging tools are available, they incur performance and memory degradations that limit their practicality to

the debugging phase. ThreadSanitizer is an example of such a tool – the slowdown of using it in Chromium was of approximately 25× compared to the native build [SI09]. Currently, the project aims to have a decrease in the range of 5× up to 15× slower [The19], a prohibitive degradation of performance for release builds. Unfortunately, no simple solution exists for a precise definition of parallel applications. Additionally, novel solutions for data synchronization have been proposed to provide more performance while increasing the complexity of their design even further. As parallelization of application is manually performed, this is a very laborious and error-prone task.

In this context, Subutai is a step forward in the direction of improving the performance of parallel applications without modifying the application source code. Even novel applications (or rewritten applications) can still benefit from our solution since Subutai accelerates the synchronization operation itself. Our design was presented as a lecture at the 55th Design Automation Conference (DAC) in its 2018 edition [CFM+18], a flagship conference worldwide in electronic design for academia and industry. According to the H5-index rank from Google Scholar Metrics, DAC is ranked the 5th best venue for Computer Hardware Design and the 2nd conference in the top 5 as of 2019 [Goo19]. This indicates the relevance of our solution and how it is perceived by the research community.

Using the information provided in this Thesis, we built a system that can be extended to provide support for other libraries besides PThreads. We highlight the inclusion of the pseudo-code implementation, Appendix A, as a key contribution to this. The existing support for PThreads already provides ample applicability due to its indirect use in other libraries (Table 3.6); however, direct support for these libraries may provide further gains. Additionally, Subutai was employed in conjunction with other tools for hybrid wireless/wired NoC support on efficient parallel computing. These works were presented in [MCM+18] [CMCD19] and will be presented in [MCCD20].

This Thesis explored the acceleration of parallel applications without modifying their source code using an HW/SW co-design approach. Moreover, the design can be extended to support additional parallel applications besides the ones explored here. Besides that, it paves the way for other future works, some of which will be discussed in the following Sections.

7.2.1 psy: Synthetic Data Synchronization Communication Creator

Simulating real applications in architecture simulators is a laborious and time-demanding task. Additionally, some applications are not supported by the simulator. We were unable to execute Ferret, an application of PARSEC, for any number higher than 8 cores on Gem5. As far as we know, no work has been able to achieve this. We were particularly interested in this application due to its synchronization design: multiple worker threads working on a 6-stage pipeline of condition-controlled queues. Such design is not captured in the

experimental results. Therefore, we intended to design a tool, called psy, to capture the data synchronization communication model of parallel applications.

The methodology for psy is thus. Firstly, the application is run natively in a given machine. Meanwhile, the data synchronization operations would be profiled and recorded for later use. Then, a generic synthetic application would be designed to read the recorded operations and mimic its behavior. Finally, we would aim to run this synthetic application in the Gem5 architecture simulator. Additionally, we would extrapolate other synchronization designs from the recorded ones; for instance, Ferret uses a fixed 6-stage pipeline regardless of thread or input size. In our synthetic version of Ferret, we could experiment using different-sized pipelines. Since extrapolation was a requirement of our proposal, we pursued statistical models for extrapolation of data.

Initially, we aimed to simulate the behavior of applications we already had access to. Therefore, we chose to simulate the barrier behavior of Bodytrack. Our first idea was to find a polynomial for each thread that represented its behavior using polynomial regression. Figure 7.1 compares the resulted data from the polynomial and the recorded execution. Unfortunately, we were only able to generate a polynomial with approximately 50% of the coefficient of determination. Thus, the polynomial was not an appropriate candidate for our methodology.

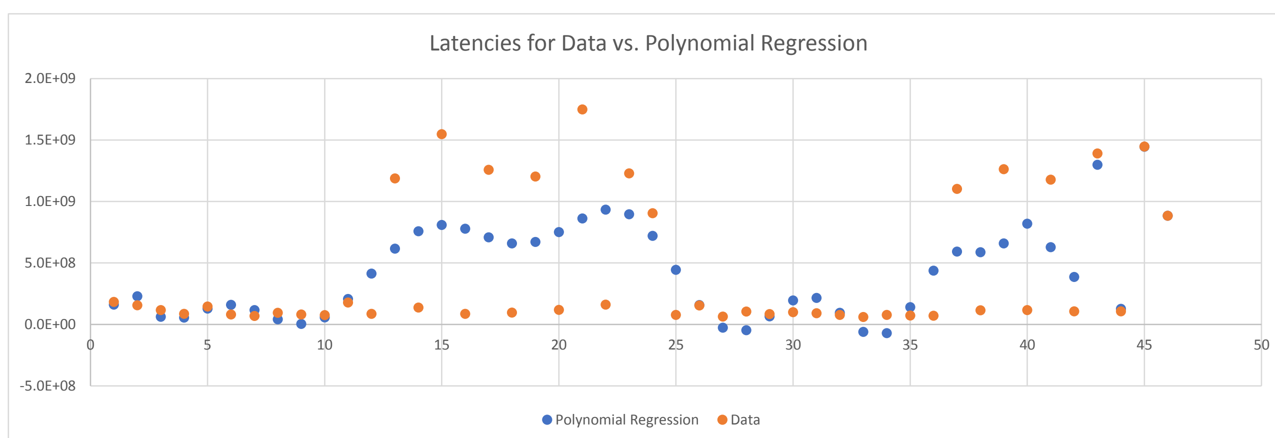
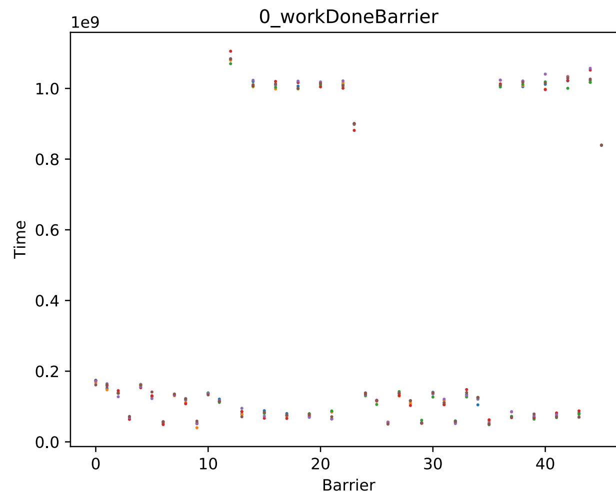


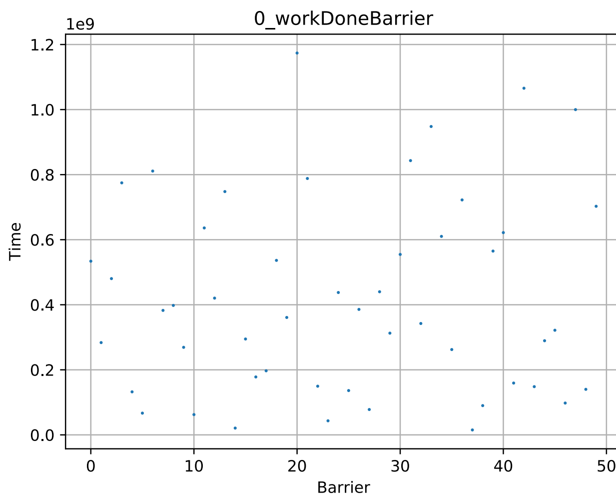
Figure 7.1 – Polynomial Regression and recorded data for the barrier latencies for a worker thread of Bodytrack.

Our second attempt was to use a gaussian distribution for generating barrier values. We produced a single distribution using the recoded data and plotted the results in Figure 7.2b, which shows the generated value for a single thread. Comparing it with the recorded value (Figure 7.2a), we believe this is an acceptable result for a single thread, as the gaussian assumes a normal distribution. However, this distribution would not produce accurate results for other threads, as the standard deviation found was too large to produce the results from Figure 7.2a. Therefore, after the first thread is modeled, we use a second gaussian distribution from the same set of recorded data to reproduce the expected behavior; the result is plotted in Figure 7.2c. A zoomed section of Figure 7.2 for some of the first 10 barriers

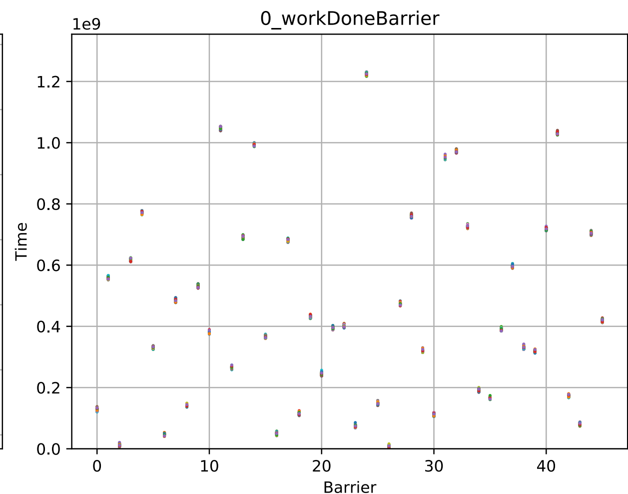
of Bodytrack is depicted in Figure 7.3. Considering the limitations of replicating the data from Figure 7.2a according to a normal distribution, we believe this was a candidate for our methodology.



(a) Recorded data.



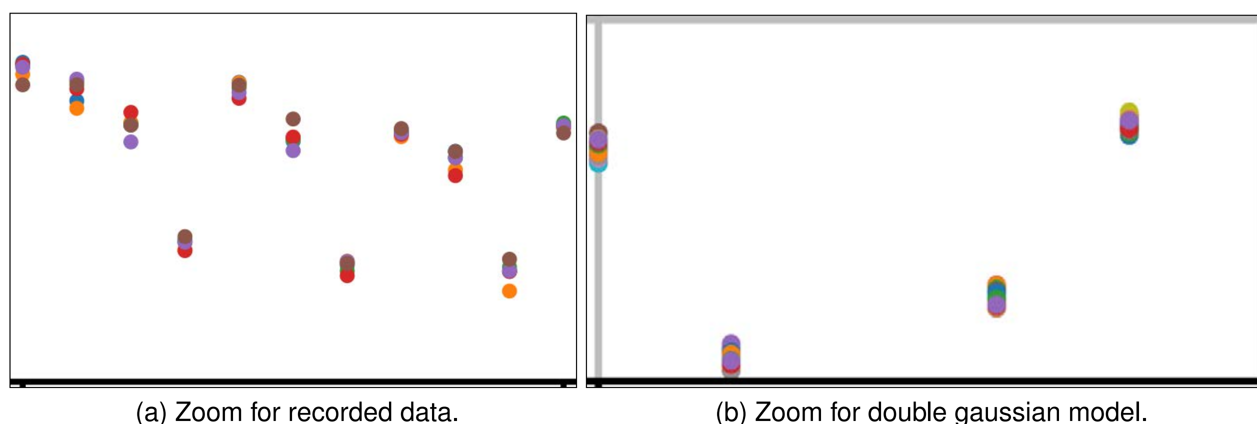
(b) One guassian model.



(c) Double gaussian model.

Figure 7.2 – Barrier latencies from Bodytrack for (a) recorded data, (b) one gaussian and (c) double gaussian models.

The missing steps for psy are the experimental results and verification of the generated values. Assuming that the double gaussian model would prove acceptable, the next steps would be to simulate different input sizes from our application sets and applications that we were unable to run (e.g., Ferret).



(a) Zoom for recorded data. (b) Zoom for double gaussian model.

Figure 7.3 – A zoom for some of the first 10 barriers of Bodytrack from Figure 7.2.

7.2.2 Energy-aware Design Exploration for Subutai-HW

As applications require seconds of execution, and Subutai-HW handles requests in a couple of nanoseconds, the latter is idle for most of the application execution time. Thus, there is ample opportunity for employing energy-efficient schemes for reducing the consumption of Subutai-HW. Besides power-gating and dynamic frequency scaling, non-volatile memories with low power dissipation could replace our current SRAM-based SPM design.

The use of different technology designs for SPM provides some interesting research explorations. Even if the use of such technology produces slower memory accesses, we believe it would be an acceptable compromise as the application operates in orders of magnitude higher than nanoseconds. Additionally, Table 4.2 allows estimating the increase in latency for each operation of Subutai-HW. Another possible compromise is to employ a mixed solution: keep a few high-speed Subutai-HW (employing SRAM-based SPM) and complement them with low-power Subutai-HW (employing non-volatile SPM) across the system. Such a design implies the creation of an algorithm to determine the best allocation place at runtime for a given synchronization primitive.

7.2.3 Barrier-aware Policy for Schedulers intended for Parallel Applications

This work proposes the CSA policy that accelerates critical sections of parallel applications. Figure 1.8 shows that CSA directly accelerates mutexes, by reducing the critical section duration in a contended scheduler scenario, and indirectly accelerates conditions, as they require mutexes for operating. The acceleration of barriers in the contended scheduler scenario is missing from this work and Figure 1.8. Although not present in this document, we believe it is a worthwhile topic of research.

As discussed in this work, barriers penalize parallel applications by blocking some threads while others are working. Thus, the use of barriers does not allow the application to be fully parallel. While some applications may choose to remove the use of barriers, others may require it. Therefore, a barrier acceleration policy on the scheduler may be proposed for the latter applications.

Our intent was to propose a simple policy initially: accelerate the threads that have not reached the instance of a barrier when a certain threshold of threads already have reached it. For instance, a threshold of 80% of threads; for 64 threads, only when 51 threads have already reached the barrier, the other 13 threads would be accelerated. For our target architecture (Section 4.1), this would not be done entirely at the software level, as the OS and scheduler are decentralized. Consequently, Subutai-HW is the component of the system that handles the barrier information. We would, then, create a new type of communication for distribution this information.

7.2.4 SW-only neocondition Implementation

The experimental results for neocondition discussed in Section 6.3.2.3 were limited to the Subutai system only. An implementation for the SW-only system (i.e., Linux kernel) would provide easier access to a wide range of applications to experiment. Such implementation requires modifying the kernel as well since the kernel space does not support neocondition natively, i.e., the kernel space assumes it must release and acquire a lock according to the PThreads specification.

We created a proof-of-concept neocondition implementation that is restricted to user space. There are two caveats because of the restriction to user space: (i) we utilize the `poll` system call with a timeout to sleep in case the condition expression was not satisfied. However, significant time is spent between the user making the system call request, and the thread going to sleep. During such a period, the condition can be satisfied, and the thread will not know. Therefore, the thread will waste the timeout period waiting for a condition that has already been satisfied; and (ii) we employ asynchronous signals from POSIX to wake up sleeping thread. The use of signals may not be the optimal tool for such a task. The implementation was verified with the Bodytrack benchmark in regard to output generated.

Table 7.1 shows the execution time for both PThreads condition and SW-only neocondition for a producer-consumer benchmark. Because the application is small, it is highly susceptible to scheduler decision policies. Hence, we plot three execution times per condition type and use the fastest execution of the PThreads condition as the reference.

We tested two variants of SW-only neocondition: broadcast and signal implementations. The former sends a POSIX signal to all threads, regardless if they are waiting for the condition or not, while the latter sends the same signal just for those waiting on the condition.

Table 7.1 – The software exploration of neocondition: execution time of PThreads condition and SW-only neocondition.

Application	Type	Execution runtime	Overhead
Producer-consumer	PThreads condition	23 844µs	Reference
	PThreads condition	27 405µs	14.9%
	PThreads condition	37 130µs	55.7%
Producer-consumer	Neocondition broadcast	46 290µs	94.1%
	Neocondition broadcast	47 459µs	99.0%
	Neocondition broadcast	50 324µs	111.0%
Producer-consumer	Neocondition signal	32 890µs	37.9%
	Neocondition signal	33 611µs	40.9%
	Neocondition signal	242 488µs	1016.9%

The variants represent a tradeoff, as only the signal version has to maintain a list of waiting threads. For the application shown in Table 7.1, the signal version is faster than the broadcast version. Nonetheless, the SW-only implementation of neocondition is not able to accelerate usages of conditions generally. Furthermore, the last execution of neocondition signal showed an astonishing overhead of execution, which results from the use of the `poll` system call for blocking, as discussed previously. We have used a timeout of 100 000 µs to check externally for new condition events, which is not the ideal implementation, yet it proved the feasibility of the solution regarding functionality.

7.2.5 Queue Optimizer: Scheduler-aware Hardware Queue

When executing multiple threads on the same core, more than one thread may hold a mutex variable; yet, only one of them can execute. As mutexes limit a parallel application to execute sequentially, the delay caused by an application not being able to run is consequential. This event is especially troublesome for executing multiple applications. To avoid such scenarios, we would like to extend Subutai-HW with a new module called Queue Optimizer.

The Queue Optimizer would be able to change the order of the double-linked queue used by Subutai-HW. Currently, the queue is organized according to the arrival order of the requests (i.e., a FIFO behavior). The Queue Optimizer would maintain the FIFO order only if the next thread to own the mutex can execute. For this choice, the Queue Optimizer needs to receive the decision information of the scheduler a priori. Thus, two modifications on the scheduler are required: (i) decide the current and the next thread to execute; and (ii) send this information for the associated Subutai-HW according to information recorded on the synchronization variables. Therefore, periodic packets of scheduler information for any number of Subutai-HW will be injected into the network for every scheduler event.

This scheme can avoid blocked threads from receiving mutexes while creating a new hardware module and increasing the overhead of both scheduler and interconnect network. A study is required to understand its impact on the target architecture.

References

[ACAAT16] Abadal, S.; Cabellos-Aparicio, A.; Alarcon, E.; Torrellas, J. "WiSync: An Architecture for Fast Synchronization Through On-Chip Wireless Communication". In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, 2016, pp. 3–17.

[ADAD15] Arpaci-Dusseau, R.; Arpaci-Dusseau, A. "Operating Systems: Three Easy Pieces". Arpaci-Dusseau Books, 2015, 621p.

[AFA+12] Abellán, J.; Fernández, J.; Acacio, M.; Bertozzi, D.; Bortolotti, D.; Marongiu, A.; Benini, L. "Design of a Collective Communication Infrastructure for Barrier Synchronization in Cluster-Based Nanoscale MPSoCs". In: Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pp. 491–496.

[AGH+11] Attiya, H.; Guerraoui, R.; Hendler, D.; Kuznetsov, P.; Michael, M.; Vechev, M. "Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated". In: Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2011, pp. 487–498.

[Ama19] Amazon Web Services. "The FreeRTOS™ Kernel". Source: https://www.freertos.org/, Dec 2019.

[App19] Apple. "Mac OS X Manual Page For pthread_cond_signal(3)". Source: https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/pthread_cond_signal.3.html, Dec 2019.

[BAAS09] Bocchino, R.; Adve, V.; Adve, S.; Snir, M. "Parallel Programming Must Be Deterministic by Default". In: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, 2009, pp. 4–4.

[Bar19a] Barney, B. "OpenMP". Source: https://computing.llnl.gov/tutorials/openMP/, Dec 2019.

[Bar19b] Barney, B. "POSIX Threads Programming". Source: https://computing.llnl.gov/tutorials/pthreads/, Dec 2019.

[BBB+11] Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.; Krishna, T.; Sardashti, S.; Sen, R.; Sewell, K.; Shoaib, M.; Vaish, N.; Hill, M.; Wood, D. "The Gem5 Simulator", *SIGARCH Comput. Archit. News*, vol. 39–2, Aug 2011, pp. 1–7.

[BBD+09] Baumann, A.; Barham, P.; Dagand, P.-E.; Harris, T.; Isaacs, R.; Peter, S.; Roscoe, T.; Schüpbach, A.; Singhania, A. “The Multikernel: A New OS Architecture for Scalable Multicore Systems”. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, 2009, pp. 29–44.

[BC19a] Bergmann, A.; Cox, A. “BigKernelLock”. Source: https://kernelnewbies.org/BigKernelLock, Dec 2019.

[BC19b] Bergmann, A.; Cox, A. “tty: BKL removal”. Source: https://lwn.net/Articles/390400/, Dec 2019.

[Ber19a] Bergman, A. “BKL: That’s all, folks”. Source: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4ba8216cd90560bc402f52076f64d8546e8aefcb, Dec 2019.

[Ber19b] Bergman, A. “Lock Totals in 2.4”. Source: http://lse.sourceforge.net/lockhier/2.4.locks.html, Dec 2019.

[BGO+15] Butko, A.; Garibotti, R.; Ost, L.; Lapotre, V.; Gamatie, A.; Sassatelli, G.; Adeniyi-Jones, C. “A Trace-driven Approach for Fast and Accurate Simulation of Manycore Architectures”. In: The 20th Asia and South Pacific Design Automation Conference, 2015, pp. 707–712.

[BGOS12] Butko, A.; Garibotti, R.; Ost, L.; Sassatelli, G. “Accuracy Evaluation of GEM5 Simulator System”. In: Proceedings of the International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2012, pp. 1–7.

[BKSL08] Bienia, C.; Kumar, S.; Singh, J.; Li, K. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”, Technical Report, Princeton University, 2008, 22p.

[BM02] Benini, L.; Micheli, G. D. “Networks on Chips: A New SoC Paradigm”, *Computer*, vol. 35–1, Jan 2002, pp. 70–78.

[BMV+08] Bobba, J.; Moore, K.; Volos, H.; Yen, L.; Hill, M.; Swift, M.; Wood, D. “Performance Pathologies in Hardware Transactional Memory”, *IEEE Micro*, vol. 28–1, Jan 2008, pp. 32–41.

[Boe81] Boehm, B. W. “Software Engineering Economics”. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981, 1st ed., 767p.

[Boe07] Boehm, H.-J. “Reordering Constraints for Pthread-style Locks”. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2007, pp. 173–182.

[Bou19] Bourbonnais, R. "Beware of the Performance of RW Locks". Source: https://blogs.oracle.com/roch/beware-of-the-performance-of-rw-locks, Dec 2019.

[Cat15] Cataldo, R. "Design and Exploration of 3D MPSoCs with on-Chip Cache Support", Master's Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, 2015, 129p.

[CBM+08] Cascaval, C.; Blundell, C.; Michael, M.; Cain, H. W.; Wu, P.; Chiras, S.; Chatterjee, S. "Software Transactional Memory: Why Is It Only a Research Toy?", *Queue*, vol. 6–5, Sep 2008, pp. 40:46–40:58.

[CCD+15] Chaker, H.; Cudennec, L.; Dahmani, S.; Gogniat, G.; Sepúlveda, M. "Cycle-based Model to Evaluate Consistency Protocols Within a Multi-protocol Compilation Tool-chain". In: Proceedings of the International Workshop on Code Optimisation for Multi and Many Cores, 2015, pp. 8:1–8:10.

[CFM+18] Cataldo, R.; Fernandes, R.; Martin, K.; Sepulveda, J.; Susin, A.; Marcon, C.; Diguet, J. "Subutai: Distributed Synchronization Primitives in NoC Interfaces for Legacy Parallel-Applications". In: Proceedings of the ACM/ESDA/IEEE Design Automation Conference (DAC), 2018, pp. 1–6.

[CKF+16a] Cataldo, R.; Korol, G.; Fernandes, R.; Matos, D.; Marcon, C. "Architectural Exploration of Last-Level Caches targeting Homogeneous Multicore Systems". In: Proceedings of the Symposium on Integrated Circuits and Systems Design (SBCCI), 2016, pp. 1–6.

[CKF+16b] Cataldo, R.; Korol, G.; Fernandes, R.; Sanchez, G.; Matos, D.; Marcon, C. "Evaluation of Emerging TSV-enabled Main Memories on the PARSEC Benchmark". In: Proceedings of the International Conference on Electronics, Circuits and Systems (ICECS), 2016, pp. 408–411.

[CKMCD19] Chatterjee, N.; Kumar Mondal, H.; Cataldo, R.; Diguet, J.-P. "CDMA-based Multiple Multicast Communications on WiNOC for efficient parallel computing". In: Proceedings of the IEEE/ACM International Symposium on Networks-on-Chip (NOCS), 2019, pp. 1–6.

[CMCD19] Chatterjee, N.; Mondal, H.; Cataldo, R.; Diguet, J.-P. "CDMA-based Multiple Multicast communications on WiNOC for efficient parallel computing". In: Proceedings of the IEEE/ACM International Symposium on Networks-on-Chip, 2019, pp. 1–6.

[CMM+16] Catania, V.; Mineo, A.; Monteleone, S.; Palesi, M.; Patti, D. "Cycle-Accurate Network on Chip Simulation with Noxim", *ACM Trans. Model. Comput. Simul.*, vol. 27–1, Aug 2016, pp. 4:1–4:25.

[Cor19] Corbet, J. "ACCESS_ONCE()". Source: https://lwn.net/Articles/508991/, Dec 2019.

[CRKH05] Corbet, J.; Rubini, A.; Kroah-Hartman, G. "Linux Device Drivers, 3rd Edition". O'Reilly Media, Inc., 2005, 636p.

[Cur19] Curtis, T. "A shoebox in someone's house is not the best place to keep source code". Source: https://www.gamasutra.com/view/news/178514/A_shoebox_in_someones_house_is_not_the_best_place_to_keep_source_code.php, Dec 2019.

[DAS19] Dehasa-Azuara, M.; Stanley, N. "Hardware Transactional Memory with Intel's TSX". Source: http://www.contrib.andrew.cmu.edu/~mdehesaa/, Dec 2019.

[DGT13] David, T.; Guerraoui, R.; Trigonakis, V. "Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask". In: Proceedings of the ACM Symposium on Operating Systems Principles, 2013, pp. 33–48.

[Dia19] Dia. "File:Branch prediction 2bit saturating counter-dia.svg". Source: https://en.wikipedia.org/wiki/File:Branch_prediction_2bit_saturating_counter-dia.svg, Dec 2019.

[DKM+12] Danowitz, A.; Kelley, K.; Mao, J.; Stevenson, J. P.; Horowitz, M. "CPU DB: Recording Microprocessor History", *Queue*, vol. 10–4, Apr 2012, pp. 10:10–10:27.

[DM19] Desnoyers, M.; McKenney, P. "Userspace RCU". Source: http://liburcu.org/, Dec 2019.

[Dre19] Drepper, U. "Futexes Are Tricky". Source: https://cis.temple.edu/~giorgio/cis307/readings/futex.pdf, Dec 2019.

[DRR14] Diegues, N.; Romano, P.; Rodrigues, L. "Virtues and Limitations of Commodity Hardware Transactional Memory". In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT), 2014, pp. 3–14.

[EAW10] Engblom, J.; Aarno, D.; Werner, B. "Full-System Simulation from Embedded to High-Performance Systems". Boston, MA: Springer US, 2010, chap. 3, pp. 25–45.

[EBSA+11] Esmaeilzadeh, H.; Blem, E.; St. Amant, R.; Sankaralingam, K.; Burger, D. "Dark Silicon and the End of Multicore Scaling". In: Proceedings of the Annual International Symposium on Computer Architecture, 2011, pp. 365–376.

[ECC14] Endo, F.; Couroussé, D.; Charles, H. "Micro-architectural Simulation of In-order and Out-of-order ARM Microprocessors with gem5". In: Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014, pp. 266–273.

[EEG18] ElDahshan, K.; Elkader, A.; Ghazy, N. "Round Robin based Scheduling Algorithms, A Comparative Study", *Automatic Control and System Engineering ACSE*, vol. 17–2, Jan 2018, pp. 29–42.

[Ext19] Ext4FS. "Frequently Asked Questions – Ext4". Source: https://ext4.wiki.kernel.org/index.php/Frequently_Asked_Questions, Dec 2019.

[Fle19] Fletcher, J. "Bluepoint Games offers insight into MGS, Ico, God of War remake process". Source: https://www.engadget.com/2011/09/23/bluepoint-games-offers-insight-into-mgs-ico-god-of-war-remake/, Dec 2019.

[FMC+16] Fernandes, R.; Marcon, C.; Cataldo, R.; Silveira, J.; Sigl, G.; Sepúlveda, J. "A Security Aware Routing Approach for NoC-based MPSoCs". In: Proceedings of the Symposium on Integrated Circuits and Systems Design (SBCCI), 2016, pp. 1–6.

[Fog19] Fog, A. "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs". Source: http://www.agner.org/optimize/instruction_tables.pdf, Dec 2019.

[FPMR18] France-Pillois, M.; Martin, J.; Rousseau, F. "Optimization of the GNU OpenMP Synchronization Barrier in MPSoC". In: Proceedings of the International Conference of Architecture of Computing Systems (ARCS), 2018, pp. 57–69.

[Fre19a] FreeBSD Library Functions Manual. "libthr". Source: https://www.freebsd.org/cgi/man.cgi?query=libthr, Dec 2019.

[Fre19b] FreeBSD System Calls Manual. "_UMTX_OP(2)". Source: http://kib.kiev.ua/kib/pshared/_umtx_op.2.pdf, Dec 2019.

[FRK02] Franke, H.; Russel, R.; Kirkwood, M. "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux". In: Proceedings of the Ottawa Linux Symposium, 2002, pp. 479–495.

[FSC+18a] Fernandes, R.; Sanchez, G.; Cataldo, R.; Agostini, L.; Marcon, C. "Least-Squares Approximation Surfaces for High Quality Intra-Frame Prediction in Future Video Standards". In: Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2018, pp. 205–208.

[FSC+18b] Fernandes, R.; Sanchez, G.; Cataldo, R.; Webber, T.; Marcon, C. "Efficient hevc intra-frame prediction using curved angular modes", *Electronics Letters*, vol. 54–21, 2018, pp. 1214–1216.

[FSS+16] Ferreira, J.; Silveira, J.; Silveira, J.; Cataldo, R.; Webber, T.; Moraes, F.; Marcon, C. "Efficient Traffic Balancing for NoC Routing Latency Minimization". In: Proceedings of the International Symposium on Circuits and Systems (ISCAS), 2016, pp. 2599–2602.

[Gai04] Gaisler, J. "A structured VHDL design method". In: Fault-tolerant Microprocessors for Space Applications, 2004, pp. 41–50.

[GCC19a] GCC Team. "5.44 Built-in functions for atomic memory access". Source: https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html, Dec 2019.

[GCC19b] GCC Team. "Nonreentrancy (The GNU C Library)". Source: https://www.gnu.org/software/libc/manual/html_node/Nonreentrancy.html, Dec 2019.

[GCC19c] GCC Team. "Welcome to the home of GOMP". Source: https://gcc.gnu.org/projects/gomp/, Dec 2019.

[GGSPM18] Garcia-Garcia, A.; Saez, J.; Prieto-Matias, M. "Contention-Aware Fair Scheduling for Asymmetric Single-ISA Multicore Systems", *IEEE Transactions on Computers*, vol. 67–12, Dec 2018, pp. 1703–1719.

[GMT16] Gangwani, T.; Morrison, A.; Torrellas, J. "CASPAR: Breaking Serialization in Lock-Free Multicore Synchronization", *SIGPLAN Not.*, vol. 51–4, Mar 2016, pp. 789–804.

[Goo19] Google Scholar. "Computer Hardware Design". Source: https://scholar.google.com.br/citations?view_op=top_venues&hl=en&vq=eng_computerhardwaredesign, Dec 2019.

[GPD+14] Gutierrez, A.; Pusdesris, J.; Dreslinski, R.; Mudge, T.; Sudanthi, C.; Emmons, C.; Hayenga, M.; Paver, N. "Sources of Error in Full-System Simulation". In: Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014, pp. 13–22.

[Gus11] Gustafson, J. "Amdahl's Law". Boston, MA: Springer US, 2011, chap. A, pp. 53–60.

[Har19] Hart, D. "A futex overview and update". Source: https://lwn.net/Articles/360699/, Dec 2019.

[HdM16] Hadade, I.; di Mare, L. "Modern multicore and manycore architectures: Modelling, optimisation and benchmarking a multiblock CFD code", *Computer Physics Communications*, vol. 205–Supplement C, 2016, pp. 32 – 47.

[HMDZ19] Howells, D.; McKenney, P.; Deacon, W.; Zijlstra, P. "LINUX KERNEL MEMORY BARRIERS". Source: https://www.kernel.org/doc/Documentation/memory-barriers.txt, Dec 2019.

[IBM19] IBM. "pthread_cond_wait() – Wait For Condition". Source: https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/apis/users_76.htm, Dec 2019.

[IEE16] IEEE Standard. "Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7", *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, vol. 1–1, Sep 2016, pp. 1–3957.

[Int17] Intel Corporation. "Intel® 64 and IA-32 Architectures Software Developer's Manual", *Volume 3A: System Programming Guide, Part 1*, vol. 1–1, Oct 2017, pp. 1–462.

[Int19a] Intel Corporation. "FAQs | Threading Building Blocks". Source: https://www.threadingbuildingblocks.org/faq, Dec 2019.

[Int19b] Intel Corporation. "Intel® OpenMP* Runtime Library". Source: https://www.openmprtl.org/, Dec 2019.

[Int19c] Intel Corporation. "Intel® Xeon Phi™ Processor 7290 (16GB, 1.50GHz, 72 core) Product Specifications". Source: https://ark.intel.com/content/www/us/en/ark/products/95830/intel-xeon-phi-processor-7290-16gb-1-50-ghz-72-core.html, Dec 2019.

[Int19d] Intel Corporation. "Mutex Flavors". Source: https://software.intel.com/en-us/node/506086, Dec 2019.

[KCM+18] Kumar, H.; Cataldo, R.; Marcon, C.; Martin, K.; Deb, S.; Diguet, J.-P. "Broadcast- and Power-Aware Wireless NoC for Barrier Synchronization in Parallel Computing". In: Proceedings of the IEEE International System-on-Chip Conference (SOCC), 2018, pp. 1–6.

[KGG+18] Kocher, P.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; Schwarz, M.; Yarom, Y. "Spectre Attacks: Exploiting Speculative Execution", *CoRR*, vol. abs/1801.01203, 2018, 1801.01203.

[Kle19a] Kleen, A. "Lock elision in the GNU C library". Source: https://lwn.net/Articles/534758/, Dec 2019.

[Kle19b] Kleen, A. "TSX anti patterns in lock elision code". Source: https://software.intel.com/en-us/articles/tsx-anti-patterns-in-lock-elision-code, Dec 2019.

[Kle19c] Kleen, A. "TSX fallback paths". Source: https://software.intel.com/en-us/blogs/2013/06/23/tsx-fallback-paths, Dec 2019.

[Kå05] Kågström, S. "Performance and Implementation Complexity in Multiprocessor Operating System Kernels", Ph.D. Thesis, Blekinge Institute of Technology, 372 25 Ronneby, Sweden, 2005, 144p.

[Lar19a] Larabel, M. "A Look At The MDS Cost on Xeon, EPYC & Xeon Total Impact Of Affected CPU Vulnerabilities". Source: https://www.phoronix.com/scan.php?page=article&item=intel-mds-xeon, Dec 2019.

[Lar19b] Larabel, M. "Benchmarking AMD FX vs. Intel Sandy/Ivy Bridge CPUs Following Spectre, Meltdown, L1TF, Zombieload". Source: https://www.phoronix.com/scan.php?page=article&item=sandy-fx-zombieload, Dec 2019.

[Lea19] Lea, D. "The JSR-133 Cookbook for Compiler Writers". Source: http://g.oswego.edu/dl/jmm/cookbook.html, Dec 2019.

[LH02] Lindsley, R.; Hansen, D. "BKL: One Lock to Bind Them All", *Ottawa Linux Symposium (OLS)*, vol. 1–1, 2002.

[Lin19a] Linux man page. "futex(2)". Source: https://linux.die.net/man/2/futex, Dec 2019.

[Lin19b] Linux Programmer's Manual. "nptl(7) – Linux manual page". Source: http://man7.org/linux/man-pages/man7/nptl.7.html, Dec 2019.

[LLF+16] Lozi, J.-P.; Lepers, B.; Funston, J.; Gaud, F.; Quéma, V.; Fedorova, A. "The Linux Scheduler: A Decade of Wasted Cores". In: Proceedings of the Eleventh European Conference on Computer Systems, 2016, pp. 1:1–1:16.

[Lov10] Love, R. "Linux Kernel Development". Addison-Wesley Professional, 2010, 3rd ed., 480p.

[LSG+18] Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Mangard, S.; Kocher, P.; Genkin, D.; Yarom, Y.; Hamburg, M. "Meltdown", *CoRR*, vol. abs/1801.01207, 2018.

[Mai19] MaiZure. "Evolution of the x86 context switch in Linux". Source: https://www.maizure.org/projects/evolution_x86_context_switch_linux/, Dec 2019.

[Mar19] Marejka, R. "Atomic SPARC: Using the SPARC Atomic Instructions". Source: http://www.oracle.com/technetwork/server-storage/solaris10/index-142944.html, Dec 2019.

[Mat19] Matulef, J. "Silent Hill HD Collection ported from unfinished code". Source: http://www.eurogamer.net/articles/2012-05-29-silent-hill-hd-collection-ported-from-unfinished-code, Dec 2019.

[MBWW17] McKenney, P. E.; Boyd-Wickizer, S.; Walpole, J. "RCU Usage In the Linux Kernel: One Decade Later". Source: https://pdos.csail.mit.edu/6.828/2017/readings/rcu-decade-later.pdf, Dec 2017.

[McA19] McAllon, A. "Without code from the original, Blizzard had to build StarCraft: Remastered from scratch". Source: https://www.gamasutra.com/view/news/300931/Without_code_from_the_original_Blizzard_had_to_build_StarCraft_Remastered_from_scratch.php, Dec 2019.

[McC04] McConnell, S. "Code Complete, Second Edition". Redmond, WA, USA: Microsoft Press, 2004, 912p.

[MCCD20] Mondal, H.; Chatterjee, N.; Cataldo, R.; Diguet, J.-P. "Broadcast Mechanism Based on Hybrid Wireless/Wired NoC for Efficient Barrier Synchronization in Parallel Computing". In: Proceedings of the Asia and South Pacific Design Automation Conference, 2020, pp. 1–6.

[McK04] McKenney, P. "Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels", Ph.D. Thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004, 380p, available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf[Viewed Dec 15, 2017].

[McK19a] McKenney, P. "Is Parallel Programming Hard, And, If So, What Can You Do About It?" Corvallis, OR, USA: kernel.org, 2019, 729p.

[McK19b] McKenney, P. "RCU part 3: the RCU API". Source: https://lwn.net/Articles/264090/, Dec 2019.

[McK19c] McKenney, P. "What is RCU, Fundamentally?" Source: https://lwn.net/Articles/262464/, Dec 2019.

[MCM+18] Mondal, H.; Cataldo, R.; Marcon, C.; Martin, K.; Deb, S.; Diguet, J. "Broadcast- and Power-Aware Wireless NoC for Barrier Synchronization in Parallel Computing". In: Proceedings of the IEEE International System-on-Chip Conference (SOCC), 2018, pp. 1–6.

[Mic04] Michael, M. "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects", *IEEE Trans. Parallel Distrib. Syst.*, vol. 15–6, Jun 2004, pp. 491–504.

[MLBR17] Moréac, E.; Laurent, J.; Bomel, P.; Rossi, A. “A bit-accurate power estimation simulator for NoCs”. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2017, 2017, pp. 1–1.

[Mol19] Molnár, I. “kill the Big Kernel Lock (BKL) tree”. Source: https://lwn.net/Articles/282319/, Dec 2019.

[Moy13] Moyer, B. “Real World Multicore Embedded Systems”. Newton, MA, USA: Newnes, 2013, 1st ed., 624p.

[MRSD16] Martin, K.; Rizk, M.; Sepulveda, M.; Diguet, J.-P. “Notifying Memories: a case-study on Data-Flow Applications with NoC Interfaces Implementation”. In: Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC), 2016, pp. 1–6.

[MS96] Michael, M.; Scott, M. “Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms”. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, 1996, pp. 267–275.

[MW08] McKenney, P.; Walpole, J. “Introducing Technology into the Linux Kernel: A Case Study”, *SIGOPS Oper. Syst. Rev.*, vol. 42–5, Jul 2008, pp. 4–17.

[Nas06] Nasir, M. “A Survey of Software Estimation Techniques and Project Planning Practices”. In: Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD’06), 2006, pp. 305–310.

[NBSG17] Nocua, A.; Bruguier, F.; Sassatelli, G.; Gamatie, A. “ElasticSimMATE: a Fast and Accurate gem5 Trace-Driven Simulator for Multicore Systems”. In: Proceedings of the International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2017, pp. 1–8.

[Nee19] Neely, T. “Fear and Loathing in Lock-Free Programming”. Source: https://medium.com/@tylerneely/fear-and-loathing-in-lock-free-programming-7158b1cdd50c, Dec 2019.

[NO16] Nakagawa, G.; Oikawa, S. “Fork Bomb Attack Mitigation by Process Resource Quarantine”. In: Proceedings of the International Symposium on Computing and Networking (CANDAR), 2016, pp. 691–695.

[NSM+15] Nilakantan, S.; Sangaiah, K.; More, A.; Salvadory, G.; Taskin, B.; Hempstead, M. “SynchroTrace: Synchronization-aware Architecture-agnostic Traces for Light-Weight Multicore Simulation”. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 278–287.

[Olo16] Olofsson, A. “Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip”, *CoRR*, vol. abs/1610.01832, 2016.

[Ope15] OpenMP Architecture Review Board. “OpenMP Application Programming Interface”, 2015.

[Ora19] Oracle. “Multithreaded Programming Guide”. Source: https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032r/index.html, Dec 2019.

[PH13] Patterson, D.; Hennessy, J. “Computer Organization and Design, Fifth Edition: The Hardware/Software Interface”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013, 5th ed., 800p.

[PKMS17] Patel, S.; Kalayappan, R.; Mahajan, I.; Sarangi, S. “A Hardware Implementation of the MCAS Synchronization Primitive”. In: Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), 2017, pp. 918–921.

[Pre19a] Preshing, J. “Acquire and Release Semantics”. Source: http://preshing.com/20120913/acquire-and-release-semantics/, Dec 2019.

[Pre19b] Preshing, J. “Memory Reordering Caught in the Act”. Source: http://preshing.com/20120515/memory-reordering-caught-in-the-act/, Dec 2019.

[Pri19] Princeton University. “PARSEC”. Source: parsec.cs.princeton.edu, Dec 2019.

[Rei07] Reinders, J. “Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism”. O’Reilly Media, 2007, 1 ed., 336p.

[RPS00] Rashinkar, P.; Paterson, P.; Singh, L. “System-on-a-chip Verification: Methodology and Techniques”. Norwell, MA, USA: Kluwer Academic Publishers, 2000, 372p.

[SCF+16] Sanchez, G.; Cataldo, R.; Fernandes, R.; Agostini, L.; Marcon, C. “3D-HEVC Depth Maps Intra Prediction Complexity Analysis”. In: Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2016, pp. 348–351.

[SDS08] Shriraman, A.; Dwarkadas, S.; Scott, M. “Flexible Decoupled Transactional Memory Support”. In: International Symposium on Computer Architecture (ISCA), 2008, pp. 139–150.

[SFC+18] Sanchez, G.; Fernandes, R.; Cataldo, R.; Agostini, L.; Marcon, C. “Low Area Reconfigurable Architecture for 3D-HEVC DMMs Decoder Targeting 1080p Videos”. In: Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2018, pp. 201–204.

[SI09] Serebryany, K.; Iskhodzhanov, T. "ThreadSanitizer: Data Race Detection in Practice". In: Proceedings of the Workshop on Binary Instrumentation and Applications, 2009, pp. 62–71.

[Sna19] Snavely, W. "CON09-C. Avoid the ABA problem when using lock-free algorithms". Source: https://wiki.sei.cmu.edu/confluence/display/c/CON09-C.+Avoid+the+ABA+problem+when+using+lock-free+algorithms, Dec 2019.

[SPA93] SPARC International. "SPARC Architecture Manual Version 9". Prentice Hall, 1993, 1 ed., 399p.

[Spe19] Sperling, E. "How Much Will That Chip Cost?" Source: http://semiengineering.com/how-much-will-that-chip-cost/, Dec 2019.

[SR15] Southern, G.; Renau, J. "Deconstructing PARSEC Scalability", *Annual Workshop on Duplicating, Deconstructing and Debunking*, vol. 1–1, 2015, pp. 1–10.

[SSF+19] Sanchez, G.; Saldanha, M.; Fernandes, R.; Cataldo, R.; Agostini, L.; Marcon, C. "3D-HEVC Bipartition Modes Encoder and Decoder Design Targeting High-Resolution Videos", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 1, 2019, pp. 1–13.

[SSP97] Sivaram, R.; Stunkel, C.; Panda, D. "A Reliable Hardware Barrier Synchronization Scheme". In: Proceedings of the International Parallel Processing Symposium, 1997, pp. 274–280.

[Sta19] Stack Exchange. "Lock-free FIFO queue implementation". Source: https://codereview.stackexchange.com/questions/158696/lock-free-fifo-queue-implementation, Dec 2019.

[sz19] sz. "The Cost of Linux". Source: http://linuxcost.blogspot.fr/2011/03/cost-of-linux.html, Dec 2019.

[The19] The Clang Team. "ThreadSanitizer". Source: https://clang.llvm.org/docs/ThreadSanitizer.html, Dec 2019.

[Wal19] Wallace, K. "Square Enix Committed to Making Its Complete Library Available Digitally". Source: https://www.gameinformer.com/e3-2019/2019/06/12/square-enix-committed-to-making-its-complete-library-available-digitally, Dec 2019.

[Was19] Wasson, S. "Errata prompts Intel to disable TSX in Haswell, early Broadwell CPUs". Source: https://techreport.com/news/26911/errata-prompts-intel-to-disable-tsx-in-haswell-early-broadwell-cpus, Dec 2019.

[Whe19] Wheeler, D. “The Linux Kernel: It’s Worth More!” Source: https://www.dwheeler.com/essays/linux-kernel-cost.html, Dec 2019.

[Wik19] Wikipedia. “File: AmdalhsLaw.svg – Wikipedia”. Source: https://en.wikipedia.org/wiki/File:AmdahlsLaw.svg, Dec 2019.

[Wra19] Wragg, D. “Low-memory-footprint mutexes for pthreads”. Source: https://github.com/dpw/skinny-mutex, Dec 2019.

[WVBM+18] Weisse, O.; Van Bulck, J.; Minkin, M.; Genkin, D.; Kasikci, B.; Piessens, F.; Silberstein, M.; Strackx, R.; Wenisch, T. F.; Yarom, Y. “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”, *Technical report*, vol. 1, 2018.

[YHLR13] Yoo, R.; Hughes, C.; Lai, K.; Rajwar, R. “Performance evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing”. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2013, pp. 1–11.

[ZK18] Zuepke, A.; Kaiser, R. “Deterministic Futexes Revisited”. In: Operating Systems Platforms for Embedded Real-Time applications, 2018, pp. 1–6.

Appendix A – SUBUTAI-HW PSEUDO-CODE IMPLEMENTATION

> The essential Saltes of Animals may be so prepared and preserved, that an ingenious Man may have the whole Ark of Noah in his own Studie, and raise the fine Shape of an Animal out of its Ashes at his Pleasure; and by the lyke Method from the essential Saltes of humane Dust, a Philosopher may, without any criminal Necromancy, call up the Shape of any dead Ancestour from the Dust whereinto his Bodie has been incinerated.
>
> Borellus from Charles Dexter Ward by H.P. Lovecraft

This Appendix provides a pseudo-code implementation of the most important procedures provided by Subutai-HW. It uses a mixture of the C language with RTL description to provide a detailed description of the states of Subutai-HW. It is important to note that this is not a replication of the implementation of Subutai-HW, which was done in RTL; rather, this pseudo-code implementation was the foundation that was used for the actual implementation.

This Chapter is organized as follows. Section A.1 defines the characteristics of the SPM associated with every NI. Section A.2 defines a number of macros that will be used by the queue procedures and the state machine. Finally, Section A.3 and A.4 define the queue procedures and the state machine, respectively.

A.1 Queue sizes

The following lines define macros related to the SPM memory size. The SPM is logically divided into two areas: queue and synchronization area. However, we assume they are physically comprised of a single contiguous memory; thus, we define the boundary between them with the following set of macros.

Listing A.1 – Queue-related macros for the SPM.

```
#define QUEUE_MEM_FIRST_ADDR            /* implementation-dependent */
#define QUEUE_MEM_ENTRY_SIZE            (2 * 4)
#define QUEUE_MEM_SIZE                  /* implementation-dependent */
#define QUEUE_MEM_LAST_ENTRY                                            \
        (((QUEUE_MEM_SIZE/QUEUE_MEM_ENTRY_SIZE) - 1) * QUEUE_MEM_ENTRY_SIZE)
#define QUEUE_SYNCH_FIRST_ADDR          /* implementation-dependent */
#define QUEUE_SYNCH_ENTRY_SIZE          (3 * 4)
```

```
#define QUEUE_SYNCH_SIZE                /* implementation-dependent */

constant queue_memory (range 0 to QUEUE_MEM_SIZE) = QUEUE_MEM_FIRST_ADDR;
constant synch_memory (range 0 to QUEUE_SYNCH_SIZE) = QUEUE_SYNCH_FIRST_ADDR;
```

A.2 Macros

The following lines provide a number of macros that define the pointer arithmetic used for Subutai-HW. All pointer accesses should use at least one of them, as Subutai-HW normally compresses two pointers into a single memory operation. In addition, all memory access is assumed to be aligned in relation to the data structure (i.e., multiple of 2 bytes for 16 bits, multiple of 4 bytes for 32 bits, and so on); thus, the empty pointer, `NULL_PTR`, is defined as an improper unaligned address for a pointer.

Listing A.2 – Pointer-related macros for Subutai-HW.

```
/* we cannot use 0 as null pointer, sadly */
#define NULL_PTR                        1
#define FREE_BIT                        31
#define SET_FREE_BIT(p)                 (p) |= (1 << FREE_BIT)
#define CLR_FREE_BIT(p)                 (p) &= ~(1 << FREE_BIT)

#define FREE_QUEUE_HEAD()               HEAD_PTR(reg_free_queue)
#define FREE_QUEUE_TAIL()               TAIL_PTR(reg_free_queue)
#define SET_FREE_QUEUE_HEAD(val)                                            \
        reg_free_queue = SET_PREV_PTR(val) | TAIL_PTR(reg_free_queue)
#define SET_FREE_QUEUE_TAIL(val)                                            \
        reg_free_queue = CLR_NEXT_PTR(reg_free_queue) | SET_NEXT_PTR(val)
#define SET_FREE_QUEUE_DUAL(head, tail)                                     \
        reg_free_queue = SET_PREV_PTR(head) | SET_NEXT_PTR(tail)

#define HEAD_PTR(addr)                  PREV_PTR(addr)
#define TAIL_PTR(addr)                  NEXT_PTR(addr)
#define PREV_PTR(addr)                  CLR_FREE_BIT((addr >> 16))
#define NEXT_PTR(addr)                  CLR_FREE_BIT((addr & 0xFFFF))
#define CLR_NEXT_PTR(addr)              (addr & 0xFFFF0000)

#define SET_PREV_PTR(addr)              (addr << 16)
#define SET_NEXT_PTR(addr)              (addr & 0xFFFF)
#define SYNCH_ID_PTR(addr)              (addr & 0xFFFF)
```

```
#define MUTEX_NO_OWNER                    (0xFFFF)
```

A.3 Queue Procedures

Listing A.3 – Queue procedures pseudo-code implementation.

```
procedure do_mem_op_and_wait()
{
        /**
         * Chip enable and related flags
         **/
}

proceduce clock_wait()
{
        /**
         * if clock'event etc etc
         **/
}

/**
 * IMPORTANT:
 * (1) out is a 16-bit pointer!! no prev/next.
 * (2) assumes there's at least 1 free position.
 **/
procedure pop_free_queue(int *out: range 0 to QUEUE_MEM_SIZE):
variable prev_addr: integer i range (0 to 2^32);
variable prev_mem: integer i range (0 to 2^32);
{
        /* the tail will be the popped element */
        *out := TAIL_PTR(FREE_QUEUE_TAIL());

        prev_addr := queue_memory[*out].pointers;
        do_mem_op_and_wait();

        /* automagically removes FREE_BIT */
        queue_memory[*out].pointers =
            SET_PREV_PTR(NULL_PTR) | SET_NEXT_PTR(NULL_PTR);
        do_mem_op_and_wait();
```

```
        /* fix new tail */
        /* the end of free space :( */
        if (PREV_PTR(prev_addr) == NULL_PTR)
                SET_FREE_QUEUE_DUAL(NULL_PTR, NULL_PTR);
        else {
                prev_mem := queue_memory[PREV_PTR(prev_addr)].pointers;
                do_mem_op_and_wait();

                prev_mem := CLR_NEXT_PTR(prev_mem) | SET_NEXT_PTR(NULL_PTR);
                queue_memory[PREV_PTR(prev_addr)].pointers = prev_mem;
                do_mem_op_and_wait();

                if (PREV_PTR(prev_mem) == NULL_PTR)
                        SET_FREE_QUEUE_DUAL(PREV_PTR(prev_addr),
                            PREV_PTR(prev_addr));
                else
                        SET_FREE_QUEUE_TAIL(PREV_PTR(prev_addr));
        }
}

/**
  * receives pointers in reg_mem_read
  * pop_addr: 16-bit pointer
  * always inserts in TAIL
 **/
procedure push_synch_queue_checked(int pop_addr: range 0 to QUEUE_MEM_SIZE):
variable new_pointers: integer i range (0 to 2^32);
{
        /* need to fix second to last tail */
        new_pointers := queue_memory[TAIL_PTR(reg_mem_read)].pointers;
        do_mem_op_and_wait();

        new_pointers := CLR_NEXT_PTR(new_pointers) | SET_NEXT_PTR(pop_addr);
        queue_memory[TAIL_PTR(reg_mem_read)].pointers = new_pointers;
        do_mem_op_and_wait();

        /* fix next of tail */
        queue_memory[pop_addr].pointers =
            SET_PREV_PTR(TAIL_PTR(reg_mem_read)) | SET_NEXT_PTR(NULL_PTR);
        do_mem_op_and_wait();
```

```
        /* new tail */
        synch_memory[SYNCH_ID_PTR(reg_synch_id)].queue_pointers =
            SET_PREV_PTR(HEAD_PTR(reg_mem_read)) | SET_NEXT_PTR(pop_addr);
        do_mem_op_and_wait();
}
/**
 * Receives reg_synch_id and reg_core_id
 **/
procedure push_synch_queue()
variable pop_addr: integer i range (0 to QUEUE_MEM_SIZE);
{
        check_full_queue();

        pop_free_queue(&pop_addr);

        reg_mem_read =
            synch_memory[SYNCH_ID_PTR(reg_synch_id)].queue_pointers;
        do_mem_op_and_wait();

        /* is it empty? */
        if (HEAD_PTR(reg_mem_read) == NULL_PTR) {
                synch_memory[SYNCH_ID_PTR(reg_sync_id)].queue_pointers =
                    SET_PREV_PTR(pop_addr) | SET_NEXT_PTR(pop_addr);
                do_mem_op_and_wait();
        } else
                push_synch_queue_checked(pop_addr);

        queue_memory[pop_addr].data = reg_core_id;
        do_mem_op_and_wait();
}

/**
 * This means we cannot handle this request here
 * ask someone else, etc, etc..
 **/
procedure check_full_queue()
{
        if (FREE_QUEUE_TAIL() == NULL_PTR)
                ...
}
```

```
/**
 * reg_mem_read:
 *              head = 22  tail = 13
 *
 * synch_mem:
 *      22(r_m_h/head)             49                        13 (r_m_h/tail)
 *    ---------                ---------                ---------
 *  | NU | 49 |       (old_prev)| 22 | 13 |               | 49 | NU |
 *    ---------(old_addr)       ---------                ---------
 *
 * free_queue:
 *      99                  77
 *    ---------          ---------
 *  | 88 | 77 |         | 99 | NU |
 *    ---------          ---------
 * reg_free_queue:
 *             head = XX tail = 77
 *
 *
 * (1)
 *         49
 *       ---------
 *      |*NU*|13|
 *       ---------
 *      or
 * (1b)
 *         22
 *       ---------
 *      | NU | NU |
 *       ---------
 *
 * (2)
 *         77            22
 *       ---------    ---------
 *      | 99 |*13*|  |*77*| NU |
 *       ---------    ---------
 *     or
 * (2b)
 *         22
 *       ---------
 *      |*NU*| NU |
```

```
  *      ---------
 **/
/**
  * receives queue_pointers in reg_mem_read
  * assumes there's at least one position on the synch_queue
  * always removes from HEAD
 **/
procedure pop_synch_queue()
variable old_head integer range (0 to 2^32);
variable next_mem integer range (0 to 2^32);
{
        /* (1) */
        old_head := queue_memory[HEAD_PTR(reg_mem_read)].pointers;
        do_mem_op_and_wait();

        if (NEXT_PTR(old_head) != NULL_PTR) {
                /* retrieve second last entry */
                next_mem := queue_memory[NEXT_PTR(old_head)].pointers;
                do_mem_op_and_wait();

                /* kill prev */
                next_mem := SET_PREV_PTR(NULL_PTR) | SET_NEXT_PTR(next_mem);
                queue_memory[NEXT_PTR(old_head)].pointers = next_mem;
                do_mem_op_and_wait();

                /* prepare to write head */
                synch_memory[SYNCH_ID_PTR(reg_synch_id)].pointers =
                    SET_PREV_PTR(NEXT_PTR(old_head)) |
                    SET_NEXT_PTR(TAIL_PTR(reg_mem_read));
        } else {
                /* last elm */
                /* (1b) */
                synch_memory[SYNCH_ID_PTR(reg_synch_id)].pointers =
                    SET_PREV_PTR(NULL_PTR) | SET_NEXT_PTR(NULL_PTR);
        }
        /* for both cases, update head */
        do_mem_op_and_wait();

        /* (2) */
        /* add recently removed entry to free queue */
        old_head := SET_FREE_BIT(
            SET_PREV_PTR(FREE_QUEUE_TAIL()) | SET_NEXT_PTR(NULL_PTR));
```

```
        /* finally save updated old_head */
        queue_memory[HEAD_PTR(reg_mem_read)].pointers = old_head;
        do_mem_op_and_wait();

        if (FREE_QUEUE_HEAD() != NULL_PTR) {
                /* fix next from second to last */
                next_mem := queue_memory[FREE_QUEUE_TAIL()].pointers;
                do_mem_op_and_wait();

                next_mem = CLR_NEXT_PTR(next_mem) | HEAD_PTR(reg_mem_read);
                queue_memory[FREE_QUEUE_TAIL()].pointers = next_mem;
                do_mem_op_and_wait();

                /* save new tail */
                SET_FREE_QUEUE_TAIL(HEAD_PTR(reg_mem_read));
        } else
                SET_FREE_QUEUE_DUAL(HEAD_PTR(reg_mem_read),
                    HEAD_PTR(reg_mem_read));
}

/**
 * reg_value, reg_mem_read already set
 **/
procedure handle_cond_no_mutex_owner(int *state)
variable aux: integer i range (0 to 2^32);
{
        /**
         * We relocate one/ALL positions from condition queue to mutex queue
         * There's a hair in this egg tho: it is possible that NOTIFY_ONE
         * didnt get the mutex.. so it can be unlocked. We need to check this
         * hence why this procedure exists.
         **/
        /* is the mutex locked or not? */
        aux := synch_memory[reg_value].data;
        do_mem_op_and_wait();

        /* it isnt locked! */
        if (aux >> 16 == MUTEX_NO_OWNER) {
                /* we gonna give the mutex for the first on the condition */
                reg_core_id = queue_memory[HEAD_PTR(reg_mem_read)].data;
                do_mem_op_and_wait();
```

```
                pop_synch_queue();

                synch_memory[reg_value].data = (reg_core_id & 0xFFFF0000);
                do_mem_op_and_wait();
                /* NOTIFY */
                prepare_pkt(HW_REQ_MUTEX_UNLOCK, NULL_PTR)
                *state = ST_WAIT;
        }
}

/**
 * receives queue pointers of condition on reg_mem_read
 **/
procedure cat_queue()
variable fifo_addr: integer i range 0 to 2^32;
variable fifo_mem: integer i range 0 to 2^32;
{
        fifo_addr := synch_memory[SYNCH_ID_PTR(reg_value)].queue_pointers;
        do_mem_op_and_wait();

        /* easy case, empty mutex queue */
        if (TAIL_PTR(fifo_addr) == NULL_PTR) {
                synch_memory[SYNCH_ID_PTR(reg_value)].queue_pointers =
                    reg_mem_read;
                do_mem_op_and_wait();
        } else {
                /* hard case */
                /* link prev fifo with next fifo */
                fifo_mem := queue_memory[TAIL_PTR(fifo_addr)].pointers;
                do_mem_op_and_wait();

                fifo_mem :=
                    PREV_PTR(fifo_mem) | NEXT_PTR(HEAD_PTR(reg_mem_read));
                queue_memory[TAIL_PTR(fifo_addr)].pointers = fifo_mem;
                do_mem_op_and_wait();

                /* now link next fifo with prev fifo */
                fifo_mem := queue_memory[HEAD_PTR(reg_mem_read)].pointers;
                do_mem_op_and_wait();

                fifo_mem :=
                    SET_PREV_PTR(TAIL_PTR(fifo_addr)) |
```

```
                SET_NEXT_PTR(fifo_mem);
            queue_memory[HEAD_PTR(reg_mem_read)].pointers = fifo_mem;
            do_mem_op_and_wait();


            /* update queue_pointers */
            synch_memory[SYNCH_ID_PTR(reg_value)].queue_pointers =
                SET_PREV_PTR(HEAD_PTR(fifo_addr))           |
                SET_NEXT_PTR(TAIL_PTR(reg_mem_read));
            do_mem_op_and_wait();
        }

        synch_memory[SYNCH_ID_PTR(reg_synch_id)].queue_pointers =
            SET_PREV_PTR(NULL_PTR) | SET_NEXT_PTR(NULL_PTR);
        do_mem_op_and_wait();
}
```

A.4 Subutai-HW state machine

Listing A.4 – Subutai-HW state machine pseudo-code implementation.

```
integer i                (range 0 to QUEUE_MEM_SIZE);
integer pop_addr         (range 0 to 2^32);

/**
 * Subutai machine state
 **/
state ST_SYNCH_ALLOC:
        clock_wait();
        alloc_synch_area(&i);

        prepare_pkt(reg_req_type, NULL_PTR);

        /* still some work to do */
        synch_memory[i].queue_pointers =
            SET_PREV_PTR(NULL_PTR) | SET_NEXT_PTR(NULL_PTR);
        do_mem_op_and_wait();

        if (reg_req_type == HW_REQ_MUTEX_ALLOC)
```

```
                synch_memory[i].data = (MUTEX_NO_OWNER << 16);
        else if (reg_req_type == HW_REQ_BARR_ALLOC ||
            reg_req_type == HW_REQ_COND_ALLOC)
                synch_memory[i].data = reg_value;

        do_mem_op_and_wait();

        goto ST_WAIT;
state ST_MUTEX_LOCK:
        clock_wait();
        reg_value = synch_memory[SYNCH_ID_PTR(reg_synch_id)].data;
        do_mem_op_and_wait();

        /* happy path */
        if (reg_value >> 16 == MUTEX_NO_OWNER) {
                synch_memory[SYNCH_ID_PTR(reg_synch_id)].data =
                    (reg_core_id & 0xFFFF0000);
                do_mem_op_and_wait();

                /* congrats */
                prepare_pkt(HW_REQ_MUTEX_LOCK, NULL_PTR);
                goto ST_WAIT;
        }
        /* sad and slow path */
        push_synch_queue();
        goto ST_WAIT;

state ST_MUTEX_UNLOCK:
        clock_wait();

        /* is there someone waiting for this lock? */
        reg_mem_read =
                synch_memory[SYNCH_ID_PTR(reg_synch_id)].queue_pointers;
        do_mem_op_and_wait();

        /* happy path */
        if (HEAD_PTR(reg_mem_read) == NULL_PTR) {
                synch_memory[SYNCH_ID_PTR(reg_synch_id)].data =
                    MUTEX_NO_OWNER << 16;
                do_mem_op_and_wait();
                goto ST_WAIT;
        }
```

```
        /* sad path */
        /* notify ! */
        prepare_pkt(HW_REQ_MUTEX_LOCK, HEAD_PTR(reg_mem_read));

        synch_memory[SYNCH_ID_PTR(reg_synch_id)].data =
            (reg_core_id & 0xFFFF0000);
        do_mem_op_and_wait();
        /* now lets fix the local queue and free queue register */
        pop_synch_queue();
        /* done! */
        goto ST_WAIT;

state ST_SYNCH_FREE:
        clock_wait();
        reg_mem_read =
                synch_memory[SYNCH_ID_PTR(reg_synch_id)].queue_pointers;
        do_mem_op_and_wait();

        /*
         * here we assume the queue pointers are empty, so no need to change
         * it
         */
        reg_mem_read = synch_memory[SYNCH_ID_PTR(reg_synch_id)].synch_id;
        do_mem_op_and_wait();

        synch_memory[SYNCH_ID_PTR(reg_synch_id)].synch_id =
            SET_FREE_BIT(reg_mem_read);
        do_mem_op_and_wait();

        /* done! */
        goto ST_WAIT;

state ST_BARR_WAIT:
        clock_wait();
        reg_value = synch_memory[SYNCH_ID_PTR(reg_synch_id)].data;
        do_mem_op_and_wait();

        /* have we reached max value? */
        if ((reg_value >> 16) + 1 == (reg_value & 0xFFFF)) {
                /* we have to notify everyone */
                /* first, the last one requesting the barrier */
                prepare_pkt(HW_REQ_BARR_WAIT, NULL_PTR);
```

```
            /* now remove one by one */
            /* -1 bc we already notified one above */
            for (i = 0; i < (reg_value & 0xFFFF) - 1; i++) {
                    /* this may be removed if reg_mem_read is updated */
                    /* accordingly in pop_synch_queue */

                    reg_mem_read = synch_memory[
                        SYNCH_ID_PTR(reg_synch_id)].queue_pointers;
                    do_mem_op_and_wait();

                    /* popping heads */
                    prepare_pkt(HW_REQ_BARR_WAIT, HEAD_PTR(reg_mem_read));

                    /* fix queue */
                    pop_synch_queue();
            }
            /* clear val */
            synch_memory[SYNCH_ID_PTR(reg_synch_id)].data =
                reg_value & 0xFFFF;
            do_mem_op_and_wait();
        } else {
            /* add one */
            synch_memory[SYNCH_ID_PTR(reg_synch_id)].data =
                ((reg_value >> 16) + 1) << 16 | reg_value & 0xFFFF;
            do_mem_op_and_wait();

            /* add to queue */
            push_synch_queue();
        }
        reg_end = 1;
        goto ST_WAIT;

state ST_COND_WAIT:
        clock_wait();

        /**
         * if mutex_owner != reg_core_id => UNDEFINED BEHAVIOR
         **/
        /* add to condition queue */
        push_synch_queue();
        /* unlock mutex */
```

```
        /* change synch id to mutex */
        reg_synch_id = reg_value;
        /**
         * No response packet is generated yet -- this will only happen
         * when the mutex is locked again
         **/
        goto ST_MUTEX_UNLOCK;
state ST_COND_NOTIFY_ALL:
        clock_wait();
        reg_mem_read =
            synch_memory[SYNCH_ID_PTR(reg_synch_id)].queue_pointers;
        do_mem_op_and_wait();

        /* great, its a notification to no one */
        if (HEAD_PTR(reg_mem_read) == NULL_PTR)
                goto ST_WAIT;

        handle_cond_no_mutex_owner(&state);
        /**
         * now we pass along the entire condition queue
         **/
        cat_queue();
        goto ST_WAIT;

state ST_COND_NOTIFY_ONE:
        clock_wait();
        reg_mem_read =
            synch_memory[SYNCH_ID_PTR(reg_synch_id)].queue_pointers;
        do_mem_op_and_wait();

        /* great, its a notification to no one */
        if (HEAD_PTR(reg_mem_read) == NULL_PTR)
                goto ST_WAIT;

        handle_cond_no_mutex_owner(&state);
        /* did it already have a owner? then we need to change the queue */
        if (state == ST_COND_NOTIFY_ONE) {
                /* we do a little dance */
                reg_core_id = queue_memory[HEAD_PTR(reg_mem_read)].data;
                do_mem_op_and_wait();

                /* remove from condition */
```

```
            pop_synch_queue();
            /* now, we put it on the mutex queue */
            reg_synch_id = reg_value;
            clock_wait();
            push_synch_queue();
        }
        goto ST_WAIT;
```

Appendix B – LIST OF PUBLISHED ARTICLES

> In my restless dreams, I see that town. . . Silent Hill. You promised you'd take me there some day. . . but you never did. Well, I'm alone there now, in our special place. . . waiting for you.
>
> Mary's Letter from Silent Hill by Team Silent

During the development of the work presented in this document, 13 scientific articles were authored or co-authored by this Author. From these articles, 2 were published in scientific journals and 11 in international conferences. Among these publications, the Author collaborated with 20 co-authors from Brasil, France, Germany, and India, including 12 professors, 3 postdoctoral researchers, 3 Ph.D. students, and 1 undergraduate student.

The following papers have been published in scientific journals:

1. Sanchez, G.; Saldanha, M.; Fernandes, R.; Cataldo, R.; Agostini, L.; Marcon, C.
   3D-HEVC Bipartition Modes Encoder and Decoder Design Targeting High-Resolution Videos.
   In: IEEE Transactions on Circuits and Systems I (TCAS-I), Regular Papers, pp. 1-13, 2019.

2. Fernandes, R.; Sanchez, G.; Cataldo, R.; Webber, T.; Marcon, C.
   Efficient HEVC intra-frame prediction using curved angular modes.
   In: Electronics Letters, pp. 1-2, 2018.

The following papers have been published in international conferences:

1. Chatterjee, N.; Mondal, H.; Cataldo, R.; Diguet, J.-P.
   CDMA-based Multiple Multicast communications on WiNoC for efficient parallel computing.
   In: 13th IEEE/ACM International Symposium on Networks-on-Chip (NOCS 2019), pp. 1-6, 2019.

2. Mondal, H.; Chatterjee, N.; Cataldo, R.; Diguet, J.-P.
   Broadcast Mechanism Based on Hybrid Wireless/Wired NoC for Efficient Barrier Synchronization in Parallel Computing.
   In: 25th Asia and South South Pacific Design Automation Conference (ASP-DAC 2019), pp. 1-6, 2019.

3. Cataldo, R.; Fernandes, R.; Martin, K.; Sepúlveda, J.; Altamiro, S.; Marcon, C.; Diguet, J.-P.
   Subutai: Distributed Synchronization Primitives in NoC Interfaces for Legacy Parallel-Applications.
   In: 55th Annual Design Automation Conference (DAC), pp. 1-6, 2018.

4. Mondal, H.; Cataldo, R.; Marcon, C.; Martin, K.; Deb, S.; Diguet, J.-P.
   Broadcast- and Power-Aware Wireless NoC for Barrier Synchronization in Parallel Computing.
   In: 31st IEEE International System on Chip Conference (SOCC), pp. 1-6, 2018.

5. Sanchez, G.; Fernandes, R.; Cataldo, R.; Agostini, L.; Marcon, C.
   Low Area Reconfigurable Architecture for 3D-HEVC DMMs Decoder Targeting 1080p Videos.
   In: 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp 201-205, 2018.

6. Fernandes, R.; Sanchez, G.; Cataldo, R.; Agostini, L.; Marcon, C.
   Least-Squares Approximation Surfaces for High Quality Intra-Frame Prediction in Future Video Standards.
   In: 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp 205-208, 2018.

7. Fernandes, R.; Marcon, C.; Cataldo, R.; Silveira, J.; Sigl, G.; Sepúlveda, J.
   A security aware routing approach for NoC-based MPSoCs.
   In: 29th Symposium on Integrated Circuits and Systems Design (SBCCI), pp 1-6, 2016.

8. Cataldo, R.; Korol, G.; Fernandes, R.; Matos, D.; Marcon, C.
   Architecture Exploration of Last-Level Caches targeting homogeneous multicore systems.
   In: 29th Symposium on Integrated Circuits and Systems Design (SBCCI), pp 1-6, 2016.

9. Sanchez, G.; Cataldo, R.; Fernandes, R.; Agostini, L.; Marcon, C.
   3D-HEVC depth maps intra prediction complexity analysis.
   In: 23rd IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp 348-354, 2016.

10. Cataldo, R.; Korol, G.; Fernandes, R.; Sanchez, G.; Matos, D.; Marcon, C.
    Evaluation of emerging TSV-enabled main memories on the PARSEC benchmark.
    In: 23rd International Conference on Electronics, Circuits and Systems (ICECS), pp 408-414, 2016.

11. Ferreira, J.; Silveira, J.; Silveira, J.; Cataldo, R.; Webber, T.; Moraes, F.; Marcon, C. **Efficient traffic balancing for NoC routing latency minimization**. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp 2599-2605, 2016.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

Titre : SUBUTAI : Primitives de synchronisation distribuées pour applications parallèles antérieures et émergentes

Mots clés : Primitives de synchronisation, Applications parallèles, Architectures matérielles

Les applications parallèles sont essentielles pour utiliser efficacement la puissance de calcul des systèmes multi-processeurs (MPSoC). Cependant, ces applications ne s'adaptent pas sans effort au nombre de cœurs à cause des opérations de synchronisation qui limitent les gains de parallélisation. Les solutions existantes soit se restreignent à un sous-ensemble de primitives de synchronisation, soit nécessitent de modifier le code source de l'application, ou les deux.
Nous présentons Subutai, une solution logiciel/matériel conçue pour distribuer les mécanismes de synchronisation sur le réseau sur puce, tout en restant compatible avec le code source originel. Subutai est composé d'un matériel spécialisé dans l'accélération des opérations de synchronisation, une mémoire privée, un pilote de système d'exploitation et une bibliothèque personnalisée.
Nous ciblons la bibliothèque POSIX Threads (PThreads), largement utilisée comme bibliothèque de synchronisation native et en interne par d'autres bibliothèques telles que OpenMP ou TBB. Nous fournissons aussi des extensions à Subutai destinées à accélérer encore davantage les applications dans deux cas: (i) plusieurs applications dans un contexte d'exécution fortement disputé; et (ii) sérialisation d'accès pour les variables condition dans PThreads. Les résultats expérimentaux sur quatre applications du benchmark PARSEC fonctionnant sur un MPSoC à 64 cœurs montrent une accélération moyenne des applications de 1,57× par rapport à des solutions purement logicielles. Une accélération de 5% en plus est obtenue en utilisant notre politique d'ordonnancement Critical Section-aware comparée à un ordonnanceur Round-Robin de base.

Title : SUBUTAI: Distributed synchronization primitives for legacy and novel parallel applications

Keywords : synchronization primitives, parallel applications, HW/SW Co-design, PThreads.

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. The existing solutions either restrict the application to a subset of synchronization primitives, require refactoring the source code of it, or both.
We introduce Subutai, a hardware/software architecture designed to distribute the synchronization mechanisms over the Network-on-Chip. Subutai is comprised of novel hardware specialized in accelerating synchronization operations, a small private memory for recording events, an operating system driver, and a user space custom library that supports legacy and novel parallel applications.
We target the POSIX Threads (PThreads) library as it is widely used as a synchronization library, and internally by other libraries such as OpenMP and Threading Building Blocks. We also provide extensions to Subutai intended to further accelerate parallel applications in two scenarios: (i) multiple applications running in a highly-contended scheduling scenario; (ii) remove the access serialization to condition variables in PThreads. Experimental results with four applications from the PARSEC benchmark running on a 64-core MPSoC show an average application speedup of 1.57× compared with the legacy software solutions. The same applications are further sped up to 5% using our proposed Critical Section-aware scheduling policy compared to a baseline Round-Robin scheduler without any changes in the application source code.