Exposing Abstraction-Level Interactions with a Parallel Ray Tracer

Alejandro Valero, Darío Suárez Gracia, Ruben Gran Tejero, Luis M. Ramos,
Agustín Navarro-Torres, Adolfo Muñoz, Joaquín Ezpeleta, José Luis Briz, Ana C. Murillo,
Eduardo Montijano, Javier Resano, María Villarroya-Gaudó, Jesús Alastruey-Benedé,
Enrique Torres, Pedro Álvarez, Pablo Ibáñez, and Víctor Viñals
{alvabre, dario, rgran, luisma, agusnt, adolfo, ezpeleta, briz, acm, emonti,
jresano, mvg, jalastru, ktm, alvaper, imarin, victor}@unizar.es
Department of Computer Science and Systems Engineering
Universidad de Zaragoza
Zaragoza, Spain

ABSTRACT

For students of any Computer Engineering program, attaining an integrated vision of the different abstraction levels is paramount to fully understand and exploit a computer system, especially when tough topics such as parallelism, concurrency, consistency, or atomicity are involved at the hardware-software frontiers. However, the structure of typical engineering programs leads to the creation of self-contained courses, where a single level of abstraction is studied and the overall picture is lost.

This paper provides a practical approach to show actual interactions between abstraction levels. This is achieved by implementing multiple components of a parallel ray tracer from the algorithmic level of the tracer to the atomic instructions required to guarantee atomicity. The students implement the full project throughout laboratories of different courses. Each lab focuses on a single abstraction level, but shows students the interactions with the rest of the levels. In addition, the hardware and software requirements of the approach are introduced, leading to the conclusion that Raspberry Pi is a suitable single-board computer for this project. Finally, this work also includes a preliminary assessment study of the proposed approach through the analysis of pre/post surveys filled out by the students.

1 INTRODUCTION

The development of a Computer Engineering (CE) program must catch up with the fast evolution of the field. Since the end of the Dennard scaling and Moore's Law era poses significant challenges in computing, hardware-software system co-design seems a promising approach to sustain the increasing performance trend. Performance is not granted for free anymore by just cramming more components in a similar area, and power consumption becomes a serious issue. This is one of the main reasons why a professional profile with an integrated vision of a computer system is highly appreciated. Besides, such a global picture allows professionals to assess risks and to deal with further professional training (specialized or not) with guaranteed success [1]. On the other hand, in most CE programs, each course typically resorts to abstractions in order to design and explain computer systems.

Abstractions establish clear boundaries across different parts of a system and aim to hide unnecessary details in the context of a given system level [12]. Abstractions help to strengthen the learning process as well, since they make the students focus on specific aspects. However, in our experience, students often lose the desired overall vision of a computer system with such an approach. This may lead students to the conclusion that some courses are self-contained and do not relate to each other. Many of them forget the hardware implications underlying high-level abstractions, in terms of performance and power.

Previous work proposes distinct high-level abstractions to ease both algorithm and software designs [9, 11]. On the contrary, we tackle the mentioned problem from the highest to the lowest level of abstraction that underlie complex parallel applications in a computer system [6]. More precisely, this work exposes to the students how the Instruction Set Architecture (ISA) and the operating system provide the required support to high-level synchronization operations, which in turn help strengthen the knowledge on how the essential concepts of parallelism, concurrency, consistency, and atomicity entangle among them and with the hardware [1, 13, 19].

To better understand the relations among the aforementioned concepts, this paper proposes to develop a *cross-cutting* project involving several laboratory sessions of different courses of a CE program. The proposal consists of a parallel ray-tracing algorithm as a motivating example that uses a concurrent queue to assign tasks to different execution threads. The queue is accessed in mutual exclusion to preserve data integrity. With this purpose, the access to the queue is managed according to each abstraction level, with *mutexes* or *futexes* (fast userspace mutexes) implemented with library functions, system calls, or directly in assembly language. This way, the proposed project covers the abstraction levels of Application, Library, Operating System, and ISA, implicating the courses of Computer Graphics (CG), Distributed and Concurrent Systems Programming (DCSP), Operating Systems (OS), and Multiprocessors (MP), respectively.

Each project lab is related to a specific level of abstraction, and purposely endowed with a context referring to the rest of the levels, contributing this way to integrate the different abstraction levels. In this work, we introduce the main guidelines and objectives of the project, which allow to implement other projects reinforcing cross-cutting learning.

Prior work has proved the suitability of a single-board computer like Raspberry Pi for teaching parallel computing over mobile devices, student laptops, virtual machines, or remote multicore

servers [14]. We build upon this study by using a common hardware board in all the project labs, which contributes to consolidate an integrated view of the system. To this end, we analyze several boards and conclude that Raspberry Pi meets the vast majority of the hardware and software requirements of a cross-cutting project.

The presented project is intended to be deployed in a CE program during the next academic year, but initial assessment studies of the proposal have already being carried out in the current academic year thanks to a set of volunteer students. This paper shows experimental results for the OS lab, including both the technical details of the lab assignment and the students learning outcomes using pre/post surveys. These surveys expose that students effectively demand a deeper understanding of the interactions between the operating system and the remaining levels, and such demands are fulfilled after the completion of the lab.

The remainder of this paper is organized as follows. Section 2 introduces the context of the CE program in which the proposed project is intended to be established. Section 3 describes the project. Section 4 presents the requirements to implement cross-cutting projects and the suitability of the selected boards. Section 5 shows the experimental results. Finally, Section 6 summarizes the paper.

2 CONTEXT OF THE CE PROGRAM

The proposed project is planned to be integrated in the CE program at the *Universidad de Zaragoza* (UNIZAR). This program consists of four academic years, 240 ECTS[1] credits in total[2]. The first two and a half years are common for all students. The core courses in this period mostly focus on the knowledge that any CE graduate should learn. Afterward, students reinforce their knowledge in the major that most interests them within five available options: Computing, Computer Engineering, Information Systems, Information Technology, and Software Engineering. Each major consists of eight compulsory courses. In addition, students select two optional courses from any other major, as well as two core courses that are studied regardless of the chosen major. Finally, the students achieve the program by undertaking an undergraduate dissertation of 12 ECTS.

The CE program is a very practical degree, where the theory always applies to the resolution of problems and the development of labs and projects. This practical load is the ideal scenario to assimilate those concepts studied in the different courses, although it usually adds a sizable burden. The lab sessions are mostly oriented to reinforce the theoretical contents of each specific course. At best, they are coordinated with other courses that belong to the same area of knowledge. As mentioned above, this can lead students to perceive a course, or a group of courses, as isolated islands, which makes it difficult for them to apply the knowledge acquired in each course in their professional career. In fact, these divisions are purely organizational and all the courses have many interactions with each other. According to the Computer Engineering Curricula [1], students should learn the development of a *whole computer* in the lab experiences that include exposure to hardware and operating systems in the context of a relevant application, which is, in our project, the ray-tracing algorithm.

[1]ECTS refers to European Credit Transfer and accumulation System: http://ec.europa.eu/education/resources-and-tools/european-credit-transfer-and-accumulation-system-ects_en

[2]https://estudios.unizar.es/estudio/ver?id=148

Table 1: Relations among the abstraction levels, courses, activities, academic years, and semesters.

Abstraction level	*Course*	*Activity*	*Academic year*	*Semester*
Application	CG	Ray tracer	4^{th}	Fall
Library	DCSP	Task queue	2^{nd}	Fall
Operating System	OS	Futex system calls	2^{nd}	Fall
ISA	MP	Futexes with assembly code	3^{rd}	Spring

3 CROSS-CUTTING PROJECT PROPOSAL

This section presents the proposed project that help the students to accomplish a holistic view of a computer system. The lab material and resources for each abstraction level, which is available upon request, consists of a description of the work to be done, code snippets, and a series of milestones, where each one builds on top of the previous one. Each lab session comprises two hours in the course associated with the level. The project involves a total of eight hours.

3.1 Overview

The proposed project allows students to consolidate the concepts of parallelism, concurrency, consistency, and atomicity exploitable in current multicore computers. We focus on a ray tracer, an appealing application which can be efficiently parallelized by learning and using the above concepts. Table 1 shows the involved four levels of abstraction, as well as the courses that have been selected within the program to face the problem jointly; for each course associated with a level of abstraction, the table shows the academic year and semester in which the activity will take place.

According to the chronological distribution of the chosen courses throughout the different academic years, the students will start the project in the second year. The first lab, which belongs to the DCSP core course, focuses on the library level. This lab deals with the implementation and management of a task queue with concurrent access by multiple threads. Such an access must be done in mutual exclusion to avoid race conditions. To do so, the students implement a mutex with library functions.

The subsequent lab will take place shortly, during the same academic year and semester, and focuses on the Operating System level. In this core lab, a mutex is implemented in userspace with a futex mechanism through atomic primitives and operating system calls that are only invoked when the mutex is contended [10], thus replacing the library functions from the above level.

The following academic year covers the third project lab, that is, the Assembly level, which is developed in the MP optional course. In this lab, assembly instructions are used to implement the mutex/futex, which allow to achieve a greater efficiency in energy consumption and performance compared to library functions and system calls.

Finally, in the fourth course, the students focus on the Application level by implementing a ray-tracing algorithm in a lab of the CG optional course. In this activity, the tasks to be performed on an image can be parallelized by dividing the image into regions. These regions are assigned to different threads by using the concurrent task queue. At this moment, the students will fully evaluate and state the differences of protecting the task queue by using library functions, system calls, or assembly instructions.

Note that the development of the presented project is subject to certain risks; e.g., students transferring from one institution to another, or students failing a course or simply not choosing the involved optional courses would not complete the full project. To mitigate such risks, all the labs include two parts. The first one, which is self-contained, includes the material for the actual lab, and the second part links the lab with the others. Therefore, if a student does not complete a preceding or following lab assignment, the faculty can provide a solution, so that students can accomplish the second part of the lab and establish the links between the abstraction levels.

3.2 Abstraction Levels

The application under study is presented in the next sections following the chronological order that students will experience.

3.2.1 ***Concurrent Task Queue.*** The aim of this lab is to teach students the implementation of one of the most common concurrent data structures: a queue. Queues, whose sequential version is already described in a previous Data Structures and Algorithms lecture, are a very suitable mechanism for the collaborative resolution of problems where several items need to synchronize. This way, producers and consumers can use one or more queues to share information and to coordinate [24]. As in any shared data structure, in order to preserve data integrity, the concurrent access to shared data requires the use of some synchronization mechanisms.

The main objectives of this lab are as follows: i) implementation of a concurrent bounded queue. Controlling concurrent access to a queue requires to consider not only mutual exclusion access to the components, but also condition synchronization (no first element exists in an empty queue, or no new element can be inserted when the queue is full), ii) understand high-level representations of execution such as C++11 `std::thread`, and iii) identify and use common low-level synchronization elements such as mutexes.

The contents of this lab are organized in two different parts. The first one consists of the bounded queue type implementation, according to the specification seen in the Data Structures and Algorithms course. Subsequently, the operations should be redesigned in order to consider synchronization aspects: in addition to ensuring access in mutual exclusion to the data structure, insertion and deletion operations should be implemented as blocking operations.

Following the focus proposed for the DCSP course, as a first assignment, students have to design the concurrent access to the queue using the coarse-grain atomic statement `<await B S>`, where `B` is a boolean guard, usually concerning shared data, and `S` is a block of sequential statements. The semantics of the statement ensures that `S` starts its execution being `B` true and no internal state of `S` is visible for the rest of processes. The high-level point of view of such an statement makes easier the task of designing correct concurrent programs, which is one of the aims of the course.

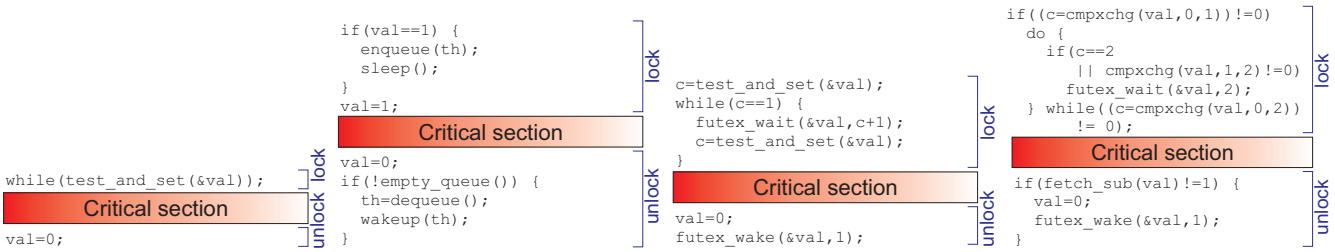
In a previous lecture, the students have studied the *token passing technique* (as proposed in [2], for instance) as a way of implementing `<await ...>` statements by using a mutex. For the second part of this lab, students will adapt the studied general approach to the design of the bounded concurrent queue data type that they propose. Notice that, at this level, the mutex is the lowest abstraction level to manage synchronization, considered as an abstract data type. The students do not know how the mutex operates internally, whether it spins until the access is granted or it goes to sleep controlled by the operating system. This issue is succinctly outlined in the lab, and students will find out the answer by implementing the mutex abstract data type in the two following activities: *Task queue protection with futex system calls* (see Section 3.2.2) and *Futexes with assembly code* (see Section 3.2.3).

As the final result, students will develop two versions of the concurrent bounded queue data type. In the first one, each operation on a queue must be executed in mutual exclusion. In the second one, students will have to adapt the readers-writers approach so as to allow multiple access to *reading* operations (operations with no side effects) while preserving mutual exclusion access for *writing* operations, giving priority to writers in case of conflict.

After completing this lab, students will have reinforced their knowledge about the main concepts related to synchronization in concurrent systems. Besides, the proposed assignment also deals with the use of design techniques focusing on the synthesis of correct concurrent programs.

3.2.2 ***Task Queue Protection with Futex System Calls.*** This lab is intended to present the mechanisms required by the operating system to provide synchronization in concurrent algorithms. The main objectives of this lab are: i) show the operating system as a service provider for the user through system calls, ii) learn an efficient use of the futex system calls and the primitives of atomic instructions provided by the operating system and the C language library, iii) understand the necessary mechanisms to provide execution in mutual exclusion with futexes and atomic instructions, and iv) show and use self-implemented lock and unlock primitives of a mutex abstraction to manage the access to the concurrent task queue implemented in the previous activity.

The lab material firstly describes the C11 atomic instructions from `stdatomic.h` and solicits the students to implement a mutex with spin-lock based on atomic instructions. Next, the *sleep* version of a mutex is motivated, introducing the mandatory intervention of the operating system to change the thread status, and providing a naive version of the sleep mutex using hypothetical `sleep` and `wakeup` system calls as well as management operations on a system queue. The limitations of this approach are used to motivate the futex system calls. Then, the syntax and use of the parameters of the `futex_wait` and `futex_wake` system calls are described. By using these calls, the students are guided to implement an intuitive and straightforward version of the sleep mutex referred to as basic implementation. Finally, the pseudo-code algorithm of a more efficient mutex is offered as a guideline to code an advanced implementation. This approach is based on the mutex implementation proposed by U. Drepper [7], which is integrated into the Linux kernel [8].



(a) Spin-lock **(b) Naive-sleep** **(c) Basic-sleep** **(d) Advanced-sleep**

Figure 1: Lock and unlock procedures of spin-lock and sleep mutexes. The sleep versions include operating system calls to change the thread status.

Figure 1(a) shows the lock and unlock procedures of a spin-lock mutex protecting a critical section. The value of the userspace `val` variable represents the two states of the mutex: not taken (`val=0`) and taken (`val=1`). The `test_and_set` atomic instruction changes the mutex state[3]. More precisely, this instruction sets `val` to 1 and loads its previous value into `c` without the overhead of a system call. Then, a thread enters into the critical section if the lock is uncontended (`c=0`). Otherwise, the thread keeps spinning in the lock. In the unlock procedure, the thread simply sets `val` to `0` to release the mutex. Since the spin-lock mutex leaves all the waiter threads in the lock awake, it suffers significant system performance losses when the mutex is contended.

Figure 1(b) illustrates the naive-sleep version of a mutex. These procedures are similar to other versions offered in textbooks of operating system concepts such as [18, 20], and [4]. This code is only correct if both procedures are executed atomically. Unlike the spin-lock mutex, assuming a non-atomic execution presents several problems that are listed in the lab material and should be understood by the students, specifically: i) the reading and writing operations of `val` are not atomically performed, which can lead to multiple threads reading the mutex as not taken, ii) the reading of the mutex and the insertion of the thread in the queue are neither atomic, which can lead to an indefinitely suspended thread if the mutex is freed between the reading and insertion operations, and iii) after waking up from the `sleep` call, a thread has no guarantee of obtaining the mutex in mutual exclusion since another thread can enter into the critical section before the former takes the mutex.

Figure 1(c) shows the basic-sleep implementation addressing all the incorrect behaviors stated above. In the lock function, the atomic operation changes the state of the mutex. If the lock is uncontended, the kernel is not invoked and the thread enters into the critical section. Otherwise, the `futex_wait` system call is invoked. It suspends the calling thread in a system queue if the lock is still taken (`val=1`), or it returns immediately if the lock has been released in the meantime (`val=0`). In the first case, the thread remains suspended until another thread wakes it up. Notice too that every time `futex_wait` returns, the thread tries to acquire the lock again.

The unlock procedure sets `val` to `0` and calls `futex_wake`. This call wakes up a number of threads stated in the second argument (`1` in the example as only a single thread is allowed to enter into the critical section) from those suspended in the system queue. Notice that such a call is invoked regardless of the mutex is uncontested or not, which can impact on the system performance.

The advanced-sleep implementation shown in Figure 1(d) addresses the performance problem of the basic version. In this case, there are three mutex states: not taken (`val=0`), taken and no waiter threads (`val=1`), and taken and at least one waiter thread (`val=2`). In the lock procedure, `test_and_set` is no longer useful since `val` takes three values. Instead, the atomic `cmpxchg` primitive is used, in which a `1` (*desired* third argument) is loaded into `val` on a successful comparison between `val` and `0` (*expected* second argument). Regardless of the result of the comparison, the original value of `val` is loaded into `c`. If `c==0`, the calling thread updates the state of the mutex as taken and no waiters and then enters into the critical section. Otherwise, the thread is suspended in the system queue by calling `futex_wait`. Previously, the second `cmpxchg` sets `val` to `2` if necessary, updating the state of the mutex as taken and at least one waiter. Note that, if the mutex is freed between the first and second `cmpxchg`, the latter returns 0 and the thread is not suspended. The third `cmpxchg` ensures that a thread takes the mutex only if a 0 is returned. In such a case, `val` is set to `2` because there is no certainty of the number of waiters.

The unlock method subtracts 1 to `val` with the atomic `fetch_sub`, which returns the previous value of the argument. The `futex_wake` call is invoked just in the case of a suspended thread in the lock, avoiding such costly system calls when there are no waiter threads. The reader is referred to [7] for further details about the advanced-sleep mutex implementation.

At the end of the lab, the students use the different mutex versions to support a complex abstraction, that is, the concurrent task queue implemented in the previous activity. Additionally, they are also encouraged to assess the suitability of each version to different contention scenarios (see Section 5.1). Overall, the students will be able to use futex system calls and atomic instructions to implement spin-lock and sleep versions of a basic synchronization abstraction such as a mutex.

3.2.3 ***Futexes with Assembly Code.*** The main idea of this lab is to help students understand the support provided by the ISA level to implement fast and reliable mutual exclusion, in terms

[3]For the sake of brevity, we have shortened the original `stdatomic.h` function names; e.g., `test_and_set` corresponds to `atomic_flag_test_and_set` and the assignment operator for `val` refers to `atomic_store`.

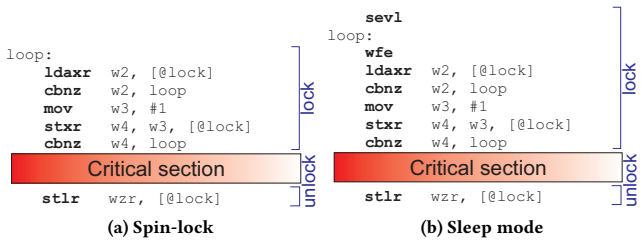


Figure 2: Lock and unlock procedures with ARMv8 assembly code.

of consistency and atomicity. The ARM processors include load-link/store-conditional instructions and barriers, providing the core for higher level structures such as mutexes and futexes. Besides, these instructions do not require any privilege level for them to execute, so programmers can directly exploit them to improve efficiency and reduce the overhead of systems calls.

By the end of this lab, students will have accomplished the following goals: i) understand how atomic instructions operate at the ISA level for the ARMv8 processors, ii) know why data memory barriers are often required when writing atomic instructions, and iii) learn the performance and energy implications of the different implementations of mutexes (spin-lock and sleep mode).

The assignments of this lab are designed to help students to engage with complex code enhancing their low-level programming skills, especially concerning performance and energy efficiency. In addition, they show how important is for an ISA to provide support for complex high-level constructors such as the mutexes used by operating systems, libraries, and applications. Finally, students gain knowledge on the relationship between the C/C++11 memory model and the corresponding consistency models at the ISA level.

The lab material of this session is organized in two parts. In the first part, the students are asked to generate a race condition with the writing of a multi-threaded program that reduces an array by adding all the elements without synchronization primitives. Then, the students code a *fetch and add* primitive with ARMv8's *load-link* (`ldaxr`) and *store-conditional* (`stlxr`) instructions [3]. The implemented *fetch and add* is included in the previous program to verify that the code is now free of race conditions.

The second part comprises two assignments. The first one proposes a basic implementation of lock and unlock mutex functions based on `ldaxr/stlxr` instructions as plotted in Figure 2(a). Threads in the lock function spin until they acquire the lock. The spinning can occur at the two `cbnz` instructions. Either if the lock is already taken (first `cbnz`) or the `stxr` instruction fails the attempt to take the lock (second `cbnz`), the branch instructions return the flow to the beginning of the loop. Notice too that, differently from the previous advanced implementation based on futex system calls, just two mutex states, taken and not taken, are considered in this level. By completing this assignment, the students find out that both functions can be the synchronization primitives for the concurrent task queue.

The second assignment proposes an advanced implementation of the lock function by replacing the power-hungry spin-lock with a `wfe` instruction. This instruction puts the core into a low-power state without returning the control to the operating system. Figure 2(b) shows such an energy-efficient sleep implementation, also with the two mutex states taken and not taken. The student will learn how the operating system considers that the program is running, while it is actually waiting for the lock to be released, and how the thread can regain the lock without a system call. In particular, the `stlr` instruction, located in the unlock function, performs a store with a release barrier and wakes up any core that could be sleeping after executing a `wfe` instruction. To guarantee progress, the threads also wake up after an interruption occurs.

With both spin-lock and sleep mode implementations, students will carry out a quick comparison between them in terms of performance and energy consumption. For the latter purpose, since the Raspberry Pi does not provide any hardware counter for energy, measurements can be done reading the current drawn at the USB power input of the board. To do so, a low-cost alternative is a USB voltage and current meter with display. Another option for more accurate results is a power analyzer such as the Newtons4th PPA500 [15]. Power analyzers support a much higher sampling rate and enable to download the samples in CSV files to perform offline analyses.

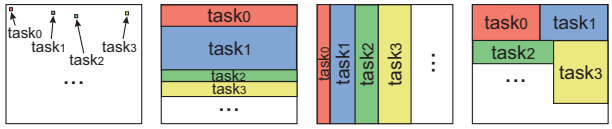
3.2.4 ***Parallel Ray Tracing.*** The CG course proposes a practical assignment involving the implementation of a *path-tracing* algorithm [23], which is parallelized by assigning different tasks to execution threads using a concurrent task queue. The main objectives of this lab are: i) find and understand the computational bottlenecks of the algorithm, ii) devise parallelization strategies that affect performance without any accuracy loss, and iii) test, explore, and analyze the impact (and overhead) of the combination of different parallelization strategies and concurrent task queues on performance.

The contents of this lab include a description of the path-tracing algorithm and why it can be parallelized. This algorithm generates a 2D image from a 3D representation of a virtual scene, including geometry and optical properties of the objects and physical characterizations of sensors (cameras) and light sources. In practice, the algorithm simulates light transport paths across the virtual scene in order to obtain the final color that reaches each of the pixels of the image. A fundamental part of path tracing is ray tracing, that is, generating the paths from a pinhole camera representation that traverse each pixel. Parallelizing the ray tracer is worthwhile, since it is computational intensive and it takes quite a long time to converge (about 1 or 2 hours for a good quality result for a simple virtual scene).

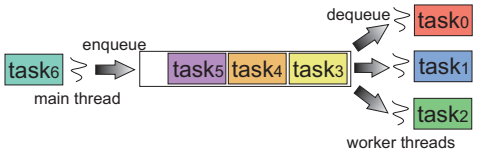
A common ray-tracing parallelization strategy is to subdivide the image into different regions, converting the computation of each of the regions into a render task to be assigned to an execution thread. The students are required to explore different parallelization strategies in different dimensions as illustrated in Figure 3(a):

- Different kinds of image regions: pixels, lines, columns, or rectangles.
- Different region sizes: smaller or larger rectangles and line or column batches.

Depending on the geometry and other properties of the virtual scene, and the different implementation details of the algorithm, the



(a) Kinds of image regions and sizes



(b) Concurrent thread-safe task queue

Figure 3: Diagrams of a 2D image split into render tasks and a concurrent thread-safe task queue to assign tasks to different worker threads.

computational load can vary greatly from one region to another [16]. For this reason, we need a safe mechanism to distribute tasks among threads. In addition, since it is impossible to estimate the computational load of each task beforehand, a concurrent thread-safe task queue such as the one depicted in Figure 3(b) is required. In this queue, a main thread enqueues all the render tasks and multiple worker threads dequeue those tasks and perform the corresponding calculations.

The students should not only implement the parallelization strategies but also combine them with the different thread-safe task-queues coming from previous labs, identifying the different pros and cons of each of the approaches and analyzing and justifying their impact on performance. For instance, the students should answer questions such as *which is the optimal region size? Which of the thread-safe queue implementations and mutex versions work best and under which circumstances?*

Overall, the implementation and parallelization of the path-tracing algorithm together with the performance evaluation of each concurrent queue and mutex will help students understand and analyze the effect of low-level mechanisms, decisions, and implementation details with high-level applications and algorithms, which will reinforce the cross-cutting vision of a computer system.

4 EXPERIMENTAL ENVIRONMENT

To consolidate the overall vision of the presented computer system, a single-board computer is proposed to be used in all the labs. To this end, we analyze a subset of commonly used boards. The selected boards are Raspberry Pi [22], DragonBoard 410C [17], HiKey 960[4], and BeagleBoard X-15 [5]. Table 2 summarizes the main hardware and software requirements that are considered as relevant for the development of this project and which of them are met by the selected boards.

The list of requirements is mainly focused on the subset of courses taking part in the presented project. Nevertheless, it is desirable to choose a base board that allows future expansions by adding more courses to the project. Therefore, we consider a broader range of requirements that would facilitate the use of the selected board for additional courses, such as Computer Architecture and Organization, Systems Administration, Computer Networks, Security, Artificial Intelligence, Machine Learning, Embedded Systems, Robotics, Video-games, or Computer Vision, among others.

[4]https://www.96boards.org/documentation/consumer/hikey960/hardware-docs/hardware-user-manual.md.html

Table 2: Hardware (H) and Software (S) requirements evaluated for the considered boards: Raspberry Pi (RP), DragonBoard 410C (DB), HiKey 960 (HK), and BeagleBoard X-15 (BB).

Type	*Description*	RP	DB	HK	BB
H&S	**Multiprocessor**	✓	✓	✓	✗
H	**JTAG**	✗	✗	✗	✓
H	**Ethernet**	✓	✗	✗	✓
H	**WiFi**	✓	✓	✓	✗
H	**Camera**	✓	✓	✓	✗
H	**Virtualization support**	✓	✓	✓	✗
H	**I/O Extensions (screen, buttons...)**	✓	✓	✓	✓
H	**GPU**	✓	✓	✓	✓
H	**DSP**	✗	✓	✓	✓
S	**Development Framework options**	✓	✓	✓	✓
S	**GPU programming options**	✓	✓	✓	✗
S	**DSP programming options**	✗	✗	✗	✓
S	**High-level/Standard OS support**	✓	✓	✓	✓
H&S	**Bare metal (no OS) support**	✓	✗	✓	✓
H&S	**High reliability**	✓	✓	✓	✓
H&S	**Low cost**	✓	✗	✗	✗
H	**Good *aging* properties**	✓	✓	✓	✓

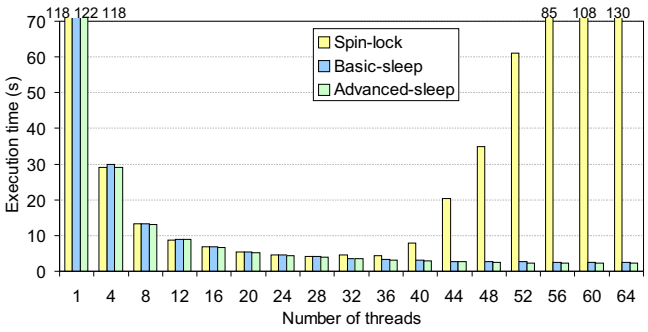
Considering the results from our study of boards, requirements, and potential courses that could use them, both Raspberry Pi and HiKey turn out good choices to be used in our project, since they have multiple cores, support for multiple operating systems, execution without an operating system (*bare metal* operation), and virtualization support [22]. However, we finally chose Raspberry Pi primarily due to its low cost, broad usage, and large amounts of open source and available materials [21].

5 EXPERIMENTAL RESULTS

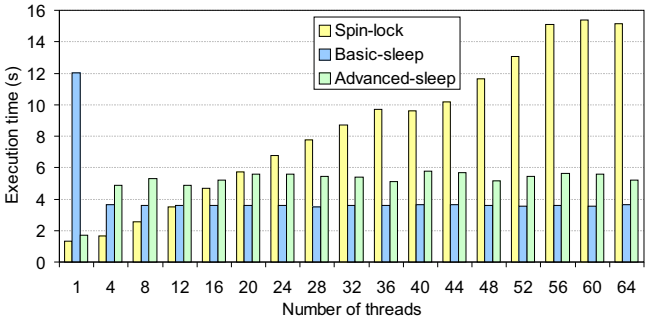
This section discusses experimental results for the OS lab. First, the main results and conclusions that should be obtained by the students from this lab assignment are highlighted. More precisely, the impact on performance of the different spin-lock and sleep versions of a mutex is analyzed. Then, a study of the students learning outcomes is presented.

5.1 Impact on Performance of the Mutex Implementations

To quantify the impact on performance of different mutex implementation alternatives, we have measured the execution time (in s) of a varying number of threads dequeueing tasks from a concurrent task queue protected with a mutex. Three different mutexes are evaluated: spin-lock (Figure 1(a)), basic-sleep (Figure 1(c)), and advanced-sleep (Figure 1(d)). Two different contention scenarios are considered, referred to as real and synthetic. In the real scenario,



(a) Real contention



(b) Synthetic contention

Figure 4: Performance of the different mutex implementation alternatives varying the number of threads.

once the threads dequeue a task from the queue, they compute a private work before coming back to the lock function. In the synthetic scenario, the threads come back to the lock function as soon as they release the mutex without performing any additional work. This scenario covers an extreme case where students can observe how a change in the amount of private work leads to unexpected conclusions. Figure 4 plots the results. All the experiments are run in a Raspberry Pi with a fixed CPU frequency of 600 MHz to guarantee reproducibility and avoid thermal issues.

In the real scenario, all the analyzed mutexes obtain very similar performance results for a relatively low number of threads (from 1 to 36 threads). From this point forward, spin-lock dramatically enlarges the execution, since a higher number of threads spin in a contended lock. On the other hand, the sleep mutexes keep reducing the execution time with the number of threads, extracting parallelism when threads compute work. Notice too that advanced-sleep slightly mitigates the execution time with respect to the basic-sleep. This is mainly because the former does not invoke costly `futex_wake` system calls when there are no waiter threads in the lock.

In the synthetic scenario, compared to the sleep mutexes, spin-lock progressively increases the execution time with the number of threads, since all of them are competing for the lock. On the contrary, the sleep mutexes speed up the execution by keeping awake just the thread that gains the mutex. Note that no performance benefits can be seen in the sleep mutexes with the number of threads. This is due to threads do not exploit any parallelism as they do not compute any work besides the critical section. An interesting observation is that on single-thread executions, basic-sleep largely drops performance. This is because the unlock function forces the `futex_wake` system call, which produces at least a context switch overhead, which possibly leaves the CPU to another process. The performance differences between the sleep mutexes are mainly due to the advanced version translates into more instructions in both lock and unlock procedures. Moreover, advanced-sleep does not take advantage of the conditioned `futex_wake`, since there are always waiter threads in the lock and the system call is always invoked as the basic version does.

5.2 Students Learning Outcomes

This section provides a preliminary qualitative assessment of the proposed OS lab. This lab was scheduled once the OS and DCSP courses finished, giving the students the opportunity to compare between the current lab assignment (i.e., no direct interactions with any DCSP lab) and the proposed lab assignment. The enrollment to this lab was voluntary, and 15% of the student's class accepted to do it.

Two different satisfaction surveys were designed, referred to as pre-survey and post-survey, and filled out by the students before and after completing the proposed lab session, respectively, totalling 17 questions. Each survey consisted on Likert-scale (i.e., 1=*strongly disagree*, 5=*strongly agree*) items and yes/no and open-answer questions. All were aimed at measuring the perceptions of the students about the lab design, its usefulness, the quality of the materials and resources, and the faculty assistance.

The main conclusions of the pre-survey are summarized as follows. First, students considered that all the courses of the degree are somehow related between them (3.75 on average). Namely, 83% of the students perceived that the OS course strongly relates to computer architecture and parallel and distributed computing areas. However, students gave a score of 4.15 to the necessity for a deeper comprehension of these ties, which confirmed us the need of this type of cross-cutting projects. From the set of technical questions, we observed that students see clear interactions between the operating system and the ISA, but not so many between the operating system and high-level concurrent constructs such as library mutexes. This confirms that professors from different areas should collaborate even more.

The post-survey revealed that the lab session was well received. More than 4 out of 5 students successfully completed the lab, and they gave an overall score of 4.42 to the quality of the lab design, the materials and resources, and the faculty assistance. After the lab, students have reached a broader vision of the relations among operating systems, computer architecture, and parallel and distributed computing, rising the perception of such interactions from 83% (pre-survey) to 92%. As a learning outcome, students discerned among the three mutex implementation alternatives and clearly identified the programmability, execution time, and efficiency trade-offs.

6 CONCLUSIONS AND FUTURE WORK

The current structure of the Computer Engineering (CE) program, arranged in isolated courses, causes students to lose sight of the overall view of a typical computer system organized in abstraction levels. This paper has presented a cross-cutting project that aims to reinforce this vision as a whole.

The presented project covers the abstraction levels of Application, Library, Operating System, and Instruction Set Architecture, and consists in the implementation of a parallel ray-tracing algorithm that uses a concurrent queue to assign tasks to different execution threads. The accesses in mutual exclusion to these queues are managed by mutexes or futexes implemented with library functions, system calls, or assembler.

The aforementioned abstraction levels have been introduced and related to each other in a subset of courses of the CE program, allowing students to consolidate the concepts of parallelism, concurrency, atomicity, and consistency. The project is supported by the elaboration of the corresponding laboratory materials and resources, as well as a detailed study of the hardware and software requirements and the consequent choice of Raspberry Pi as the hardware development platform.

The proposed project is currently being deployed in the chosen courses of the CE program. So far, the feedback received by the students is encouraging. Most students see the need to consolidate a holistic vision of CE concepts. After the proposed learning experience, the students show an improvement in the integrated perception of the presented concepts. Moreover, in the evaluated lab session, the students also showed acquisition of the knowledge addressed in the session.

As for future work, we plan to involve more courses to strengthen the project and to obtain a more detailed analysis of assessment results from the remaining lab sessions.

ACKNOWLEDGEMENTS

This work has been supported by the *Universidad de Zaragoza* under Grant PIIDUZ_18_246, by the *Ministerio de Ciencia, Innovación y Universidades* and the European ERDF under Grant TIN2016-76635-C2-1-R (AEI/ERDF, EU), and by the Aragon Government (T58_17R research group) and ERDF 2014-2020 *"Construyendo Europa desde Aragón"*.

REFERENCES

[1] ACM. 2016. Computer Engineering Curricula 2016 Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering.
[2] G. R. Andrews. 1991. *Concurrent Programming. Principles and Practice* (1st ed.). The Benjamin/Cummings Publishing Company, Inc.
[3] ARM. 2018. ARM DS-5 Development Studio Examples.
[4] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces* (1st ed.). Arpaci-Dusseau Books, LLC.
[5] G. Coley. 2016. *BeagleBoard X15 System Reference Manual.* BeagleBoard.org.
[6] J. Cuenca and D. Giménez. 2016. A Parallel Programming Course Based on an Execution Time-Energy Consumption Optimization Problem. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops.* 996–1003.
[7] U. Drepper. 2011. Futexes Are Tricky. http://people.redhat.com/drepper/futex.pdf.
[8] U. Drepper and I. Molnar. 2005. The Native POSIX Thread Library for Linux. https://akkadia.org/drepper/nptl-design.pdf.
[9] C. Ferner, B. Wilkinson, and B. Heath. 2013. Toward Using Higher-Level Abstractions to Teach Parallel Computing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops.* 1291–1296.
[10] H. Franke, R. Russell, and M. Kirkwood. 2002. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium.* 479–495.
[11] D. Ginat and Y. Blau. 2017. Multiple Levels of Abstraction in Algorithmic Problem Solving. In *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education.* 237–242.
[12] J. Kramer. 2007. Is Abstraction the Key to Computing? *Commun. ACM* 50, 4 (2007), 36–42.
[13] V. Kumar. 2002. *Introduction to Parallel Computing* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc.
[14] S. J. Matthews, J. C. Adams, R. A. Brown, and E. Shoop. 2018. Portable Parallel Computing with the Raspberry Pi. In *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education.* 92–97.
[15] N4L. 2018. N4L Newtons4th Ltd PPA500/1500 KinetiQ. User Manual. https://www.newtons4th.com/wp-content/uploads/2014/07/PPA5xx_15xx-User-Manual-v2-91.pdf.
[16] M. Pharr, W. Jakob, and G. Humphreys. 2017. *Physically Based Rendering: From Theory to Implementation* (3rd ed.). Morgan Kaufmann Publishers Inc.
[17] Qualcomm. 2016. *DragonBoard™410c based on Qualcomm®Snapdragon™410E processor. Peripherals Programming Guide Linux Android.* Qualcomm Technologies, Inc.
[18] A. Silberschatz, P. B. Galvin, and G. Gagne. 2012. *Operating System Concepts* (9th ed.). Wiley Publishing.
[19] D. J. Sorin, M. D. Hill, and D. A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.
[20] W. Stallings. 2008. *Operating Systems: Internals and Design Principles* (6th ed.). Prentice Hall Press.
[21] E. Upton, J. Duntemann, R. Roberts, T. Mamtora, and B. Everard. 2016. *Learning Computer Architecture with Raspberry Pi* (1st ed.). Wiley Publishing.
[22] E. Upton and G. Halfacree. 2014. *Raspberry Pi User Guide.* John Wiley & Sons Ltd.
[23] E. Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation.* Ph.D. Dissertation. Stanford University. Advisor(s) Guibas, Leonidas J.
[24] A. Williams. 2012. *C++ Concurrency in Action.* Manning Publications.