Pontifícia Universidade Católica
do Rio de Janeiro

Alexandre Rupert Arpini Skyrme

Safe Record Sharing in Dynamic Programming Languages

TESE DE DOUTORADO

Thesis presented to the Programa de Pós Graduação em Informática of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Doutor em Informática

Advisor : Prof. Noemi de La Rocque Rodriguez
Co–Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
March 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Alexandre Rupert Arpini Skyrme

Safe Record Sharing in Dynamic Programming Languages

Thesis presented to the Programa de Pós Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC–Rio, as partial fulfillment of the requirements for the degree of Doutor.

Prof. Noemi de La Rocque Rodriguez
Advisor
Departmento de Informática — PUC–Rio

Prof. Roberto Ierusalimschy
Co–Advisor
Departmento de Informática — PUC–Rio

Prof. Ana Lúcia de Moura
Departamento de Informática — PUC-Rio

Prof. André Rauber Du Bois
UFPEL

Prof. Jean-Pierre Briot
CNRS Brazil

Prof. Renato Fontoura de Gusmão Cerqueira
IBM Research Brazil

Prof. José Eugênio Leal
Coordinator of the Centro Técnico Científico da PUC–Rio

Rio de Janeiro, March 27th, 2015

Alexandre Rupert Arpini Skyrme

Has a BSc in Computers Engineering and an MSc in Informatics from PUC-Rio. His main research interests include concurrent and parallel programming, computer networks and distributed computing. He has extensive professional experience in information systems security.

Bibliographic data

Skyrme, Alexandre Rupert Arpini

Safe Record Sharing in Dynamic Programming Languages / Alexandre Rupert Arpini Skyrme; advisor: Noemi de La Rocque Rodriguez; co–advisor: Roberto Ierusalimschy. — 2015.

v., 77 f: il. ; 30 cm

Tese (Doutorado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. Concorrência. 3. Multithreading. 4. Lua. 5. Comunicação. 6. Compartilhamento. I. Rodriguez, Noemi de La Rocque. II. Ierusalimschy, Roberto. III. Pontifícia Universidade Católica do Rio de Janeiro. Departmento de Informática. IV. Título.

CDD: 004

This thesis is dedicated to my grandfather, Darcy Arpini (*in memoriam*), a truly special person who taught me valuable lessons about life. He was always very proud of even the smallest of my achievements and, despite the lack of knowledge about computers, insisted in coming over for the presentation of my MSc dissertation. I will always be thankful to you, grandpa.

Acknowledgments

To professors Noemi Rodriguez and Roberto Ierusalimschy, who have been an inspiration since my undergraduate classes, for all the knowledge they have shared with me and for making my humble academic career possible.

To professor Markus Endler, with whom I had the pleasure of working with, for the opportunities to publish and to broaden my research field.

To Pablo Musa, for generally helping me out and for the companionship while I was working long days and long hours at LabLua.

To Li Lu, at the University of Rochester, for kindly answering my questions regarding Deterministic Parallel Ruby and for publishing the source-code for the applications used to benchmark it.

To PUC-Rio, for the scholarship that allowed me to pursue my PhD degree and for providing me with a home away from home since 1997.

To Clarisse Siqueira, to whom words cannot do justice, for all the love and support, for being my connection to sanity throughout the last couple of years and for always being there for me. I am really privileged to share my life with you.

To my little Tomás, for bringing me much joy and reminding me of what's important in life.

Abstract

Skyrme, Alexandre Rupert Arpini; Rodriguez, Noemi de La Rocque; Ierusalimschy, Roberto. **Safe Record Sharing in Dynamic Programming Languages**. Rio de Janeiro, 2015. 77p. DSc Thesis — Departmento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Dynamic programming languages have become increasingly popular and have been used to implement a range of applications. Meanwhile, multicore processors have become the norm, even for desktop computers and mobile devices. Therefore, programmers must turn to parallelism as a means to improve performance. However, concurrent programming remains difficult. Besides, despite improvements in static languages, we find dynamic languages are still lacking in concurrency support. In this thesis, we argue that the main problem with concurrent programming is unpredictability – unexpected program behaviors, such as returning out-of-thin-air values. We observe that unpredictability is most likely to happen when shared memory is used. Consequently, we propose a concurrency communication model to discipline shared memory in dynamic languages. The model is based on the emerging concurrency patterns of not sharing data by default, data immutability, and types and effects (which we turn into capabilities). It mandates the use of shareable objects to share data. Besides, it establishes that the only means to share a shareable object is to use message passing. Shareable objects can be shared as read-write or read-only, which allows both individual read-write access and parallel read-only access to data. We implemented a prototype in Lua, called luashare, to experiment with the model in practice, as well as to carry out a general performance evaluation. The evaluation showed us that safe data sharing makes it easier to allow for communication among threads. Besides, there are situations where copying data around is simply not an option. However, enforcing control over shareable objects has a performance cost, in particular when working with nested objects.

Keywords

Concurrency. Multithreading. Lua. Communication. Sharing.

Resumo

Skyrme, Alexandre Rupert Arpini; Rodriguez, Noemi de La Rocque; Ierusalimschy, Roberto. **Compartilhamento Seguro de Registros em Linguages de Programação Dinâmicas**. Rio de Janeiro, 2015. 77p. Tese de Doutorado — Departmento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Linguagens de programação dinâmicas estão cada vez mais populares e já foram utilizadas para desenvolver uma ampla gama de aplicações. Enquanto isso, processadores multi-núcleo se tornaram padrão, mesmo em computadores pessoais e dispositivos móveis. Dessa forma, os programadores precisam recorrer ao paralelismo para aprimorar o desempenho de seus programas. Entretanto, a programação concorrente permanece difícil. Adicionalmente, a despeito de avanços em linguagens estáticas, avaliamos que linguagens dinâmicas ainda carecem de suporte adequado à concorrência. Nesta tese argumentamos que o principal problema da programação concorrente é a imprevisibilidade – comportamentos inesperados de programas, tais como retornar valores descabidos. Observamos que a imprevisibilidade é mais provável quando memória compartilhada é utilizada. Consequentemente, propomos um modelo de comunicação para concorrência que visa disciplinar o compartilhamento de memória em linguagens dinâmicas. O modelo é baseado nos padrões emergentes de concorrência de não compartilhar dados por padrão, imutabilidade de dados e tipos e efeitos (que transformamos em capacidades). Ele demanda a utilização de objetos compartilháveis para compartilhar dados e utiliza troca de mensagens para comunicação entre fluxos de execução. Objetos compartilháveis podem ser compartilhados apenaspara leitura ou para leitura e escrita, o que permite acesso individual de escrita e acessos paralelos de leitura. Implementamos um protótipo em Lua para experimentar com o modelo na prática, bem como para conduzir uma avaliação geral de desempenho. A avaliação demonstra que há benefícios na utilização de memória compartilhada, mas ao mesmo tempo revela que os controles utilizados para assegurar a disciplina ocasionam um impacto de desempenho.

Palavras-chave

Concorrência. Multithreading. Lua. Comunicação. Compartilhamento.

Contents

List of Figures

(. . .) a folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs.

Edward A. Lee, *The Problem with Threads.*

1
Introduction

In this thesis we propose a communication model for dynamic programming languages to support safe record sharing among concurrent execution flows. The proposed model builds on the emerging concurrency patterns of not sharing data by default, data immutability, and types and effects (which we turn into capabilities). It should make concurrent programming less error-prone, which is a necessary step to promote parallelism as a means to increase software performance. Moreover, it should fill a gap in concurrency support for dynamic languages, which despite their popularity, still mostly rely on concurrency constructs that are hard to use correctly.

For over a decade processor manufacturers have been shifting from increasing clock speeds to supporting parallel execution within the same chip. Multi-core processors are now the norm, even for desktop computers and mobile devices. The main consequence of this paradigm shift is that programmers must now turn to parallelism as a means to improve performance, a development that has been referred to as the *concurrency revolution* [69]. An evidence of the importance of parallelism is presented in a survey on parallel programming conducted for Intel [19]: it shows that, among surveyed software developers, 56% considered parallel programming was important to their software and 26% considered it was critical.

However, despite the need for parallelism to improve performance, concurrent programming remains difficult [45, 57, 59, 69] and programmers still struggle with fundamental concurrency problems [23, 48]. Consider, for example, a survey conducted at Microsoft [28] that randomly selected 10% of all employees responsible for development, test and program management, and polled them with regards to concurrency use. The survey shows that 54.3% of respondents had to deal with concurrency bugs on a monthly, or less, basis and that 72.9% of respondents considered concurrency bugs were hard to reproduce. Notorious concurrency bugs have already been responsible for severe real-world consequences: the malfunctioning of a computerized radiation therapy machine called the Therac-25, which resulted in serious injuries and even deaths [46], the NASDAQ Stock Market glitch on Facebook's Initial Pub-

lic Offering (IPO), which resulted in as much as U$13 million in losses [42], and the 2003 northeastern USA power outage, which affected 60 million people [44]. This evidences that to allow programmers to exploit the potential of multi-core processors to improve performance, we must make concurrent programming less error-prone, i.e., we must make it simpler for programmers to write correct, safe concurrent applications.

1.1 Dynamic Programming Languages

To our knowledge, emerging concurrency patterns have been primarily explored in static programming languages. Meanwhile, dynamic programming languages thrive in popularity, as evidenced by their high ranking in programming languages popularity indexes. Despite their popularity, dynamic languages still lag behind in concurrency support. It is true that most of them already include at least some concurrency support. Nevertheless, this support is often limited, not allowing them to benefit from parallel execution, and is based on constructs for synchronization and communication that can be simple to understand conceptually but are hard to use correctly. Moreover, the commitment to promoting safe record sharing certainly has not appealed as much to dynamic languages as it has to static languages.

Creating a concurrency communication model that emphasizes safety for dynamic languages may appear paradoxical. Since dynamic languages commonly use dynamic typing and rely on code interpretation, this could suggest flexibility should always be prioritized over safety in these languages. That is a misconception. Firstly, it is not true that dynamic languages always prioritize flexibility: the wide adoption of automatic memory management, for instance, clearly evidences a concern about safety. Secondly, code interpretation allows for plenty of control over execution, as faults can be handled timely and appropriately by the language's runtime environment. It is often easier to debug programs written in interpreted languages.

Concurrency bugs, however, stand for the exact opposite: unsafety and lack of control. We must face that, on average, they are more troublesome than most common bugs that affect serial programs. They typically depend on specific execution interleavings to appear and can lead to behaviors that are hard to reason about. Therefore, the real paradox is to have concurrency support that lacks safety and control in dynamic programming languages.

1.2 Nondeterminism and Unpredictability

The main characteristic that distinguishes concurrent programming from sequential programming is probably also what troubles programmers the most: the need for communication among execution flows, both for synchronization and for exchanging data, combined with the asynchronous nature of parallel execution. It can be challenging for programmers to reason about the execution of concurrent programs, since they must consider the execution interleaving and the influence of all communication among execution flows. Consequently, developing correct concurrent programs is hard.

The literature commonly points at the inherent *nondeterminism* of concurrency as the culprit [4, 7, 48, 57, 73] for the difficulties in reasoning about, testing and debugging concurrent programs. However, there is no consensus on the semantics of nondeterminism for concurrency. Different definitions and types of nondeterminism are used throughout the literature [4, 10, 20, 49], none of which is widely accepted.

Besides the semantic ambiguity, there is another problem with blaming the difficulties of concurrent programming on nondeterminism: not all instances of behaviors that can be considered nondeterministic are troublesome. For instance, a common definition of determinism is to generate the same output when given the same input [7, 11, 20, 34]. According to that definition, some well-known algorithms can be considered nondeterministic by design [11], such as branch-and-bound solvers and mesh refinement methods. Regardless of not always generating the same output when given the same input, these algorithms still produce results that belong to a well-defined set. For a simple example of an application that can be considered nondeterministic and yet works as expected, consider an airline reservation system that processes, in parallel, requests from multiple customers wishing to reserve seats on the same flights. Depending on the order in which requests are processed, the flights will be filled up differently, i.e., nondeterministically. Nevertheless, as long as reservations are consistent, the nondeterministic behavior is acceptable and expected, due to the asynchronous nature of parallel execution.

Overall, after analyzing the literature, we concluded that the term nondeterminism is used to refer to two different aspects of concurrent programming, in fact, to two different types of nondeterministic behaviors. The first aspect refers to behaviors that are inherent to the asynchronous nature of parallel execution. It essentially encompasses schedule ordering: executing the same concurrent application, with the same input, multiple times, can result in different orderings but yet produce correct output. The second aspect refers to

completely unexpected behaviors that compromise an application's reliability, such as returning out-of-thin-air values. It encompasses, for instance, the behavior of C and C++ programs in the presence of race conditions [12] — according to the C11 and C++11 standards, the results of race conditions are undefined; in practice, this means that "any behavior whatsoever is allowed" [13].

Concurrency discussions commonly mention both types of nondeterministic behaviors together, without any explicit distinction regarding how much each of them contributes to making concurrent programming hard. This lack of distinction may give the impression that these two types are equivalent or that they contribute equally to making concurrent programming hard. However, that is not the case. Conceptually, the two types of behaviors are fundamentally different: while the first type is mostly focused on the ordering of intermediate program steps that do not have observable external effects, the second is focused on reliability and program correctness (both of which are directly related to observable external effects). Even though both types contribute to making concurrent programs hard to reason about, clearly the second type represents a greater evil: while different execution orderings can increase the number of potential interleavings, unexpected behaviors can produce results that make absolutely no sense, are extremely hard to track and can be literally impossible to reproduce. While some concurrent programs could benefit from having some guarantees about their intermediate steps, as evidenced by related research [7, 4, 58], all concurrent programs could benefit from eliminating unexpected behaviors. In this thesis, we consider that the first type of nondeterministic behavior is unavoidable [13] and harmless when compared to the second type. Therefore, we are only concerned with the second type of nondeterministic behavior, the one that makes applications unreliable and is clearly evil.

Since the type of nondeterministic behavior we are interested in is the one related to unexpected behaviors, or behaviors that cannot be predicted by the programmer, we propose using the term *unpredictability* instead. This term better conveys a key concept in our research and its use allows us to avoid the semantic ambiguity associated with nondeterminism. In this thesis, we say a program is unpredictable when it can exhibit unexpected behaviors, such as returning out-of-thin-air-values. An unpredictable program is inherently unreliable and ultimately incorrect.

It is true that, besides unpredictability, concurrency presents other challenges to programmers, such as avoiding deadlocks, starvation and other liveness hazards. However, in this thesis, we choose to focus on unpredictability,

as we understand it is the main reason why concurrent programs are hard to develop, test and debug. Unpredictability must be eliminated to make it simpler for programmers to write concurrent applications correctly.

1.3 The Cause of Unpredictability

Now that we have established our interest in unpredictability, we need to look into what causes it in concurrent programs. We begin this investigation by highlighting an essential safety property in concurrency: *atomicity*. An operation, or set of operations, is said to be atomic when its execution is indivisible and irreducible. No intermediate states or side effects of atomic operations are visible until execution is completed, or in other words, atomic operations provide *isolation*.

Atomicity can be provided at different granularity levels. Database management systems, for instance, ensure that all statements within a transaction are executed as a single atomic operation. Some programming languages also support atomicity. The C11 and C++11 standards, for instance, introduced atomic types as a template class that can be used to provide simple atomic operations for different standard data types. On the lowest level, some processors support atomic instructions, such as compare-and-swap (CAS), which can compare and change the contents of a memory location in a single step.

In theory, programmers should always be aware of the granularity level of the provided atomicity and should implement concurrency controls accordingly. However, in practice, reasoning about atomicity is hard and there is often a mismatch between the atomicity expected by programmers and the atomicity effectively provided. This mismatch leads programmers to implement incorrect synchronization among execution flows, which in turn results in unpredictability when applications are executed.

To further understand how incorrect synchronization is related to unpredictability, we must first recall and briefly explain the two standard models [2] used for communication among execution flows in concurrent programs: *shared memory* and *message passing*. Shared memory is often regarded as having better performance [59, 72, 74], while message passing is often regarded as being less error-prone [45, 59, 69, 74, 71]. Both models can be used as an implementation technique and as a communication abstraction offered to programmers: just as it is possible to offer a shared-memory abstraction in a distributed environment which relies on an underlying message-passing scheme for communication, it is also possible to offer a message-passing abstraction implemented with shared-memory synchronization primitives in a multiprocessed environment.

Message passing can refer either to an architecture model for parallel computers or to a communication model [74] – for the purpose of the present discussion, we are only interested in the latter. In message passing, execution flows communicate with each other by sending and receiving messages. Message addressing usually relies either on unique execution flow identification or on decoupled communication channels (or mailboxes). Message passing has at least two prominent advantages over shared memory: it can easily be used as an abstraction for both local and distributed communication and, as long as explicit operations are used, it makes communication among execution flows localized and explicit in the code. The latter makes reasoning about a program's execution easier, as the points in code where nondeterministic behavior can occur are clear. Despite its advantages, message passing also has its drawbacks. Message passing involves copying data between different address spaces, leading to large overheads as the transmitted data grow in size. The semantics of message transmission, which is related to the performance impact caused by data transmission, are often unclear, for instance with regards to the transmission of complex objects or data structures.

Synchronization and data exchange in message passing are implemented with the same standard operations: send and receive. This means that sending or receiving a message presents an opportunity both for synchronization and for data exchange. Since synchronization in message passing typically occurs on a per-message basis, it is reasonable to assume that the granularity of the expected atomicity is also per-message, i.e., that each message should be handled atomically. This assumption is straightforward and presents an atomicity model that is easy to reason about when implementing concurrency controls. In this case, unpredictability is still possible: consider, for example, an operation that must execute atomically but requires several messages to be exchanged before it can complete. Nevertheless, since unmet atomicity expectations in standard message passing are less common, unpredictability is unlikely.

In shared memory, execution flows communicate with each other by means of *shared records*[1] that are written to and read from locations in a global address space. Shared memory is conceptually easier to use and understand. It offers a quick communication method, because data does not need to be copied among different address spaces. However, since multiple execution flows can potentially access the same shared record simultaneously, programmers must use available synchronization primitives to control concurrency. These

[1] A record, as defined by Hoare [40], is a computer representation of an object of interest in a model created to solve a problem.

primitives are complex to use and error-prone, making it hard for programmers to pinpoint the proper placement for them. Also, it is hard to determine whether an arbitrary operation accesses a shared record, as programming languages often do not explicitly distinguish these records. In particular, when referential semantics is used, it is difficult to control how and when each execution flow accesses shared records. Thus, we can conclude that shared memory is easier to use but only as long as correctness is not a concern; using shared memory safely is definitely not easy.

Synchronization in shared memory can rely on a variety of mechanisms that implement two basic types of synchronization [2]: *mutual exclusion* and *conditional synchronization*. Unlike in message passing, the granularity of the expected atomicity in shared memory is often unclear. Consider, for instance, a simple operation to increment an integer counter in a standard C program: `x++`. Although it might look like a single operation and thus lead programmers to expect atomic execution, in fact this operation can be broken into at least three lower-level instructions that are executed independently: load the value of `x` from memory into a register, increment the value in the register, store the updated value in memory.

Incorrect synchronization with shared memory can be commonly found in concurrent applications. A study on real-world concurrency bugs [48] that analyzed a total of 105 bugs in open-source software, classified 74 of the bugs as having incorrect synchronization[2] as their root cause. This evidences that when shared memory is used, unmet atomicity expectations are more common and thus unpredictability is more likely. Consider a simple example: a standard C or C++ program, with two threads that both increment by one a global integer counter initially set to 0, without resorting to any synchronization primitives, and prints the result on the screen. The program obviously has a synchronization problem and it would be fair to say it is nondeterministic if the only results it could print would be 1 and 2. However, as we mentioned previously, the C11 and C++11 standards state that such cases have undefined behavior. This means that depending on the compiler and the execution platform, this simple program could output essentially anything [12], such as 42.

It should be clear by now that to eliminate unpredictability we must discipline communication among execution flows to prevent unmet atomicity expectations, i.e., to make it easier for programmers to implement proper synchronization. We have argued that although unpredictability can happen

[2]Although the study defines two bug categories, both of them can be related to the basic types of synchronization: atomicity violations relate to mutual exclusion and order violations relate to conditional synchronization.

in spite of which communication model is used, it is more likely to happen when shared memory is used. An obvious, but perhaps naive, approach would be to simply discard shared memory altogether and concentrate all our efforts in disciplining message passing. The reason why this would not be appropriate is that we would be left with the drawbacks of message passing: performance loss due to overheads for data transmission and potentially intricate semantics for the transmission of complex objects.

An evidence that shared memory should not be entirely discarded is that even programming languages that provide communication models based in message passing, sometimes also include the means to enable data sharing. Examples of such languages include Erlang and Scala. Erlang, which uses message passing for inter-process communication, includes the `ets` (built-in term storage), `dets` [21] (disk-based term storage) and `mnesia` [55] (a distributed database management system) modules, all of which allow for shared data. Scala, which supports the actor model, allows references to be transmitted between actors [63], thus allowing data to be shared.

Instead of discarding shared memory, we propose implementing proper language-level mechanisms to make it easier for programmers to use shared memory safely. In other words, we second the opinion that *disciplined shared-memory* [1, 12] should be promoted to make it easier for programmers to write correct concurrent applications with acceptable performance.

1.4 Disciplined Shared Memory

In this thesis, we propose promoting disciplined shared memory by creating a communication model that can combine the benefits of shared memory and of message passing, as we will explain next. The first and most important property of our communication model is that it should *prevent unpredictability* to facilitate developing, testing and debugging concurrent programs. It should also *keep communication localized and explicit*, like in message passing, to simplify reasoning about program execution. It should allow for data exchange among execution flows with satisfactory performance, like in shared memory, since that is a key concern in concurrency. Last, but not least, it should *provide specific, well-defined constructs for communication* and should *compel programmers to use them properly*, treating violations adequately. In short, the model should provide *structured communication* [67], as that is a necessary step to allow programmers to more easily write correct concurrent applications.

Even more important than what a programming language or library offers to programmers, is what it prevents them from doing. The same survey on

parallel programming conducted for Intel [19] that we cited earlier, shows that while determining where parallelization should occur for maximum benefit is ranked as the first most beneficial feature for parallel programming tools, avoiding incorrect (or unexpected) results when parallelization is used is ranked second.

Such a strict enforcement approach may limit flexibility, as it is common for programmers to try to improve performance by circumventing standard communication patterns [75]. Still, this is a matter of choosing between flexibility and simplicity, a choice programmers are commonly faced with. We can make an analogy with memory management, regarding the choice between manual and automatic memory management [30]. For a long time, programmers despised automatic memory management as too slow for real applications; currently, many programmers and programming languages adopt it, making dangling pointers and memory leaks things of the past. Concurrency must do a similar transition.

The status quo of concurrent programming — namely preemptive multithreading with shared memory — emphasizes the ability to quickly introduce concurrency in a program over safety. Most average programmers would probably agree that while introducing concurrency in an existing sequential program can be a simple task, ensuring the correctness of a concurrent program rarely can. Likewise, most average programmers probably feel it is easier to program in imperative languages than in purely functional languages. Nevertheless, purely functional languages are praised for producing more reliable code [64]. Communication models must *emphasize safety*, even at some cost, to make it simpler for programmers to write concurrent applications correctly.

1.5 Previous Work

In this thesis we are interested in designing a concurrency communication model that allows for safe record sharing while preventing unpredictability. Before we started working on our model, we did an extensive literature survey [67] for recent approaches that were related to safe record sharing and could be built upon. We found at least three promising approaches: *no default sharing*, *data immutability* and *types and effects*.

The first approach, not sharing data by default, is the most fundamental to promote disciplined shared memory and thus to our communication model. When data is not shared by default, it is possible to compel programmers to correctly use constructs that allow for concurrency control and can prevent unpredictability. Also, when data must be shared explicitly, it is easier

to reason about program execution with regards to communication among execution flows. Examples of programming languages that take this approach include Erlang, which uses single assignment variables and message passing for communication, and D, which uses thread-local variables by default.

The second approach, data immutability, has a very straightforward use for concurrency: it allows data to be freely shared among execution flows. Since immutable data is never updated, concurrent accesses do not require synchronization and thus ensuring atomicity is not an issue. Moreover, it allows communication models' implementations to decide whether to pass data by value or by reference [43]. Nevertheless, exclusively using immutability may not be practical as it could impose a performance bottleneck caused by the need to create a new object every time an existing object needs to be changed. On the other hand, the coexistence of mutable and immutable data makes it harder to reason about program execution and allows for incorrect synchronization when there is no enforcement that only immutable data can be shared. Several authors have explored immutability in their research [8, 24, 29, 33]. Examples of programming languages that support immutability include D, Clojure and F#.

The third approach, types and effects, is used by Bocchino et al. [9] to develop the Deterministic Parallel Java (DPJ) project, which employs an effect system to partition memory into regions and then control concurrent access to each region. It is also used by Heumann, Adve and Wang [39], who propose a concurrent programming model based on dynamically created tasks that have programmer-specified effects and run on a scheduler that enforces that tasks with conflicting effects cannot be executed concurrently. Other examples of tasks and effects applied to concurrency include the works of Lu et al. [50], who propose a type and effect system to control what effects may occur in parallel and whether they interfere with each other, and of Flanagan and Qadeer [22], who implement a type system that uses effects to specify and verify atomic methods in multithreaded Java.

1.6 Contributions

To sum up, in this thesis we propose a concurrency communication model for dynamic programming languages that provides structured communication by supporting safe record sharing among multiple execution flows running on the same machine. The proposed model builds on emerging concurrency patterns and should eliminate unpredictability, as this is a fundamental step to make concurrent programming less error-prone and allow for parallelism as a means to improve performance. We experiment with the proposed model

using the Lua programming language, which is an ideal testbed as it has a small code-base and simple, well-defined, semantics. Nevertheless, we have not conceived the model to rely on any exclusive Lua features, therefore it should be possible to use with other dynamic programming languages as well.

The expected contributions of this thesis are the following:

1. A new concurrency communication model for dynamic programming languages designed to allow for safe record sharing and a discussion about what it takes to implement it.

2. A prototype implementation of our model in Lua that allows for safe data sharing among parallel Lua threads.

1.7 Organization

The remainder of this thesis is organized as described next. In Chapter 2 we identify emerging concurrency patterns, besides discussing related implementations and research work pertaining both to static and to dynamic languages. In Chapter 3, we introduce our communication model, explain its key concepts and discuss what it took to implement it in Lua. In Chapter 4 we provide an evaluation of our model using the prototype we implemented in Lua. Finally, in Chapter 5 we present some closing remarks.

2
Emerging Concurrency Patterns

In this chapter we discuss emerging concurrency patterns we have identified in the literature and in practice. Then, we examine how these patterns are supported in static and in dynamic programming languages. Parts of this chapter, in particular the discussion about the emerging patterns and the analysis of how dynamic languages support them, are based in an article [68] we published on the same topic. We start the discussion with a brief introduction about multiprocessing and multithreading.

Conventionally, concurrency support is offered either in the form of multiprocessing, with fork-like functions, or of multithreading. Multiprocessing support ensures that it is technically possible to execute code in parallel. Nevertheless, most implementations offer only a thin layer that calls operating system inter-process communication methods, such as sockets, pipes and shared memory regions. These methods, since they are low-level, usually require more effort from programmers to implement communication among execution flows in applications. Besides, the cost of creating and destroying processes can make multiprocessing unsuitable for applications with large quantity of simultaneous execution flows or dynamic workloads that demand the creation and destruction of execution flows during runtime. Since we want to make it simpler for programmers to write correct concurrent applications and to avoid a performance overhead in our communication model when execution flows are created and destroyed during runtime, multiprocessing is not a suitable choice. Thus, we focus on multithreading.

The most popular form of multithreading is preemptive multithreading with shared memory, which is supported by a variety of programming languages and commonly used in concurrent applications. In this form of multithreading, context switches are not controlled by programmers and can occur at any point during a program's execution. Communication and synchronization among execution flows relies on shared memory and concurrency control is normally implemented with mutual exclusion, condition variables and semaphores. Preemptive multithreading with shared memory is commonly associated with many of the complexities [45, 57, 59, 69] that make concurrent

programming hard. Still, there is a growing interest in developing new, improved, concurrency patterns, especially with regards to safe data sharing. We surveyed recent implementations and research work to identify these patterns, which we talk about next.

The first and most important pattern we have identified is *no-default sharing*. It implies no data is shared among execution flows unless explicitly requested by the programmer. This is a fundamental pattern, as it serves as the basis for safe data sharing. Only when data is not shared by default, a discipline for sharing can be *enforced* by the programming language and thus language-based mechanisms can be used to ensure safety. Moreover, explicit data sharing helps programmers reason about program execution, since it allows them to more easily identify, when analyzing source code, where shared data access can occur.

The second pattern we have identified is *types and effects* [56]. Type systems [25, 62] allow programming languages, by means of type checking, to automatically detect certain program misbehaviors and ensure some invariants are maintained throughout a program's execution. Type and effect systems build on type systems to include effects, which are annotated properties of the semantics of programs that extend type definitions. An effect is normally expressed by an action, which describes what is being done, and by a region, which describes where the action is taking place. Simple examples of effects can include, for instance, memory operations, where the actions would be read, write, allocate and free, and the regions would the points in a program where these operations are performed. In concurrency, types and effects can be used to control concurrent access to shared data. For example, operations on shared data could be mapped as effects and the type system could prevent calls that operate on the same shared data, or calls with *conflicting effects*, to execute in parallel.

Type and effect systems are suitable for static programming languages, which usually have static typing, but not for dynamic languages, which usually have dynamic typing. Still, the same base concept of controlling concurrent access to shared data also resulted in a pattern that is more suitable for dynamic languages: *data ownership*. The data ownership pattern prevents conflicting accesses by assigning an owner to each piece of shared data and enforcing that only the execution flow that owns the data may access it. Ownership of shared data can be transferred to allow multiple execution flows to access the data throughout a program's execution.

The third pattern we have identified is *data immutability*, which is not a concurrency pattern by itself, but represents an important building block

as it provides a simple, yet effective, way to safely share data. Immutability should not be mistaken for constants: while constants are associated with single values that cannot be changed, immutability is associated with data structures or objects that, as a whole, including all contained values, cannot be changed.

No synchronization is needed to access immutable objects, which makes them inherently thread-safe. Also, it is easier to ensure object consistency when working with immutability, since the object essentially only admits a single state after initialization. Immutability makes the choice between passing an object by value or by reference come down to an implementation matter. It allows message passing semantics to be implemented without copying data [43], i.e., it allows references to immutable data be passed as if they were copies of the same data. The main disadvantage of immutability is the need to create a new object each time an existing immutable object must be altered, which can be a performance bottleneck. Also, working exclusively with immutable data requires programmers to change development paradigms employed in imperative languages, where patterns commonly rely on altering data.

The fourth pattern we have identified are *futures*, which explore asynchronous computations that can potentially be executed in parallel. A future [3] works like a read-only view for the resulting value of a computation that will be executed at some point in time during a program's execution. Futures are generally used for asynchronous, non-blocking, parallel computations. Implementations usually allow execution flows to check whether a future's computation is complete or to wait for its completion to retrieve its value, sometimes they also allow the use of callbacks to avoid blocking while waiting for a future to complete. The term future is sometimes used interchangeably with the term *promise* [26]. Still, a promise in fact works like a single-assignment container that completes a future. Futures and promises are not directly related to data sharing, as a language could support either of them and still allow for unsafe data sharing. Still, it is a relevant emerging pattern as it structures a mechanism for asynchronous parallel tasks.

In the following sections, we look at some programming languages and research work that support the emerging patterns we have identified. We consider both static and dynamic programming languages. Since there are no formal, or widely accepted, definitions for static and dynamic languages, we present, at the beginning of the following sections, the characteristics that are generally used to define each of these types of programming languages.

2.1 Static Programming Languages

Static programming languages are generally defined by static typing. Their implementations usually rely on compiling code, rather than interpreting it. Examples of static languages include C and C++, Java, Haskell, F# and D.

C and C++ implementations conforming to the C99 and C++03 standards do not include native concurrency support. Multithreading is commonly supported by means of the POSIX thread library (pthreads), on POSIX-compliant operating systems, and of the Win32 threads API on Microsoft operating systems. However, the latest C and C++ standards, C11 and C++11, include some native concurrency features.

For starters, the C11 and C++11 standards support preemptive multithreading with shared memory, including mutual exclusion (locks) and condition variables for synchronization. They also support atomic data types, which are primitive data types that allow for atomic access. Apart from the atomic access, which is only guaranteed for single instructions, atomic read-modify-write operations are also provided for atomic data types. Among the emerging concurrency patterns we have identified, the C11 and C++11 standards only support futures and promises.

Java supports preemptive multithreading with shared memory. Synchronization in Java can be implemented with standard locks and condition variables, as well as with Java monitors[1]. From the emerging concurrency patterns we have identified, Java only supports futures. Nevertheless, Java has been used as the basis for the exploration of other patterns, as we will discuss next.

In particular, the usefulness of immutability in Java has been discussed by different authors [8, 24, 29]. Still, defining and enforcing immutability in an object-oriented language like Java, which supports referential semantics and allows for both mutable and immutable data, is not a simple matter. Some evidences of the complexities associated with immutability in Java can be found in a number of related research papers [33, 31, 32, 60, 14, 76, 6].

An example of immutability applied to concurrency in Java is presented by Boyapati et al. [15], who propose a type system to prevent data races and deadlocks, using ownership types [17] to associate objects with protection mechanisms. Each protection mechanism is defined as part of a variable's type and may either refer to the lock used to control access to the object pointed by the variable or indicate that the object may be freely accessed by multiple execution flows. The latter case implies that the object is immutable,

[1] Java monitors differ from the original monitor concept [35, 36] as there is no enforcement that shared variables can only be accessed with monitor methods [37].

or it is only accessible by a single execution flow, or the variable holds a unique pointer to the object. Although rather focused on the ordered use of locks, this work is pertinent to our research as it uses immutability as part of its protection mechanisms and it enforces that objects are associated with a protection mechanism.

Apart from immutability, Java has also been used to explore types and effects. Bocchino et al. [9], for instance, developed the Deterministic Parallel Java (DPJ) project, which employs an effect system to partition memory into regions and then control concurrent access to each region. The same effect system for Java is also used by Heumann, Adve and Wang [39]. They propose a concurrent programming model based on dynamically created tasks that have programmer-specified effects and run on a scheduler that enforces that tasks with conflicting effects cannot be executed concurrently. Yet another work that explores types and effects in Java is presented by Flanagan and Qadeer [22], who implement a type system that uses effects to specify and verify atomic methods in multithreaded Java.

While immutability is still finding its way to imperative programming languages, like Java, it has been long used as a standard in functional languages. One of the most popular Haskell compilers (the Glascow Haskell Compiler – GHC), for instance, includes a concurrency extension that exploits immutability. Concurrent Haskell [61] provides primitive types and operations that allow concurrent execution flows to be created, synchronized and to communicate. Since data is immutable by default in Haskell, execution flows can share data simply by using the same variables. An underlying synchronization mechanism ensures lazy expressions are only evaluated by one execution flow; if other execution flows try to evaluate an expression that is already being evaluated, they are blocked until the first execution flow finishes evaluating and overwriting the expression with its value. Concurrent Haskell also provides a mutable state variable, by means of monads, to allow execution flows to share data — Peyton Jones et al. [61] outline the reasons why such mechanism is needed, which evidence some of the limitations of concurrent programming exclusively with immutable data.

Even programming languages that are not purely functional, but are strongly inspired in functional programming, have exploited immutability. Consider, for instance, the F# programming language. It is built on the .NET framework, which includes a number of concurrency features, such as standard preemptive multithreading, parallel asynchronous tasks and parallel data structures. All variables in F# are immutable by default, which allows them to be freely shared. Nevertheless, F# does not preclude mutable variables, which

must be declared by using the `mutable` keyword. Using immutable variables by default and requiring programmers to explicitly declare variables as mutable can help minimizing the risk of improper shared data manipulation, as it sets a safe standard. Still, it is up to programmers, once a variable is declared as mutable, to ensure proper synchronization for concurrent access.

Another example of programming language that, inspired by functional languages, explores immutability is the D programming language. Although it supports preemptive multithreading, it does not rely on global variables and standard shared memory synchronization constructs for communication among threads. All variables in D, by default, are local to threads. There are two ways to share data among threads in D: message passing and explicitly shared variables. Message passing is the preferred way to share data in D, it allows a thread to send data to another thread, which is specified by a unique thread identifier (tid). However, D places a restriction on what data can be sent in messages: only non-pointer, explicitly shared or immutable variables are allowed. Explicitly shared variables must be declared with the `shared` type modifier and can only be manipulated with a special set of atomic operations, provided by D. In practice, this means D enforces safe data sharing. Moreover, it shows that, besides immutability, D supports the no-default sharing emerging pattern.

2.2 Dynamic Programming Languages

Dynamic programming languages are generally defined by dynamic typing and by the existence of an eval-like function, i.e., by the ability to dynamically execute code in the same environment of the program itself. Their implementations usually rely on interpreting code, rather than compiling it. Examples of dynamic languages include PHP, Python, Ruby, Lua, Perl, JavaScript and Clojure.

PHP includes, among its Process Control Extensions, a POSIX Threads (pthreads) extension that provides an object-oriented API for userland multithreading. It supports mutual exclusion and condition variables, as well as Java-inspired synchronization constructs like synchronized code blocks and the `wait` and `notify` methods. Notes included in the PHP manual warn users against sharing resources among contexts; they also state that restrictions and limitations are necessary to provide a stable environment when pthreads is used. Among popular dynamic languages, PHP is the one that offers the most basic concurrency support.

The reference implementations for Python (CPython) and Ruby (MRI) support userland threads that are mapped to system threads. They include

standard shared memory synchronization constructs such as mutual exclusion and condition variables (Ruby and Python), as well as semaphores (Python).

Despite their multithreading support, parallel execution in Python and Ruby is severely hampered, because both of them include a *Global Interpreter Lock* (GIL). The GIL is a mutual exclusion lock used by the interpreter which only allows a single userland thread to execute at a time. It simplifies implementation, as all code is inherently thread-safe, but it can be a performance bottleneck, resulting in performance degradation when more threads are used to run a script. The only way to circumvent the GIL is to use alternative language implementations, such as Jython and IronPython (for Python), and JRuby and Rubinius (for Ruby). These alternative implementations, though, do not seem to have mainstream adoption.

Python, in addition to standard preemptive multithreading, supports futures, one of the emerging patterns we have identified. It does so with its `concurrent.futures` module, which allows the asynchronous execution of callables using either a pool of threads or a pool of processes.

Ruby, in addition to standard preemptive multithreading, supports cooperative (or collaborative) multithreading by means of *fibers*. This form of multithreading is not designed to support parallel execution, but rather to allow for explicitly coordinated threads which take turns to execute. Fibers are lightweight userland threads scheduled cooperatively that cannot be executed in parallel.

Lu, Ji and Scott used Ruby to develop Deterministic Parallel Ruby (DPR) [51], a library for Ruby that provides parallel constructs with a focus on ensuring determinism. Among provided constructs are *isolated futures*, which are futures with a deterministic behavior. To ensure determinism, DPR requires that isolated futures represent pure functions (i.e., a function that always generates the same outputs when given the same inputs and has no observable side effects) and that the arguments to these functions are passed by deep copy, to prevent concurrent changes. The motivation for DPR closely relates to our own, i.e., to make it simpler for programmers to write correct concurrent applications. Nevertheless, the approach taken by DPR is to ensure determinism, which is a strong property that can be costly, in terms of usability and performance, to obtain. In our research, as we discussed previously, we strive for predictability, which is cheaper to obtain and will make it significantly easier for programmers to write correct concurrent applications.

The Lua programming language is another example of a dynamic language that, like Ruby, supports cooperative multithreading. Lua supports *coroutines*, which follow the same paradigm as fibers in Ruby, i.e., they are

lightweight userland threads scheduled cooperatively that cannot be executed in parallel.

Lua allows a somewhat obvious approach to implement the no-default sharing pattern: not sharing anything at all. For a C application to interact with Lua, it must first create a *Lua state*, a data structure that defines the state of the interpreter and keeps track of global variables, among other interpreter-related information. States are independent, i.e., they do not share data. The Lua C API allows C applications to create and manipulate multiple states at a time. In addition, when Lua code is loaded in a Lua state, its execution can be controlled just like a coroutine: it can be suspended (yielded) and resumed. The combination of multiple independent states and system threads allows for the implementation of Lua libraries for concurrent programming that support parallel execution.

Examples of such libraries include luaproc [66] and Lua Lanes [53]. The luaproc library implements the concept of *Lua processes*, which are lightweight execution flows of Lua code able to communicate only by message passing. Each Lua process runs in its own Lua state and thus no data is shared by default. A set of workers, system threads implemented in C with pthreads, are responsible for executing Lua processes. There is no direct relation between workers and Lua processes, they are independent from each other (N:M mapping). Workers repeatedly take a Lua process from a ready queue and run it either to completion or until a potentially blocking operation is performed, in which case the corresponding process can be suspended and placed in a blocked queue. The only operations that are potentially blocking in luaproc are the standard message passing primitives: send and receive.

Message addressing in luaproc relies on communication channels and Lua processes have no unique identifiers. Each message carries a tuple of values with basic Lua data types. More complex or structured data can be transmitted either by serializing it beforehand or by encoding it as a string of Lua code that will be later executed by the receiving process.

The Lua Lanes library, like luaproc, uses multiple Lua states to host independent execution flows of Lua code (or lanes). However, unlike luaproc, in Lua Lanes each execution flow of Lua code is associated with a single system thread, i.e., there is a 1:1 mapping.

Communication among lanes is based on Linda [27], i.e., on tuple spaces. Each tuple in Lua Lanes must have a number, string or boolean as its key. Basic Lua data types, with the exception of coroutines, can be used as values in tuples. Two operation sets are available for lanes to interact with tuple spaces. The `send` and `receive` operations are analogous to the `out` and `in` operations

in Linda. The `send` operation queues a tuple; the `receive` operation consumes a tuple specified by a key. The `set` and `get` operations are used to access a tuple without queuing or consuming it. The `set` operation writes a value to a tuple specified by a key; it overwrites existing values and clears queued tuples with the same key. The `get` operation is analogous to the `rd` operation in Linda, i.e., it reads the value of a tuple specified by a key without consuming the tuple.

Another example of dynamic language that does not share data by default is Perl 5, which supports multithreading through its `threads` module. It provides interpreter-based threads (or simply ithreads), which are userland threads that are mapped 1:1 to system threads and scheduled preemptively. For each new thread, Perl creates a new interpreter instance and copies all the existing data from the current thread. This results in no data being shared among threads, i.e., all variables are local to threads by default. It also results in a higher cost to create threads, since all thread data must be copied.

Despite not sharing data by default, Perl 5 allows variables to be explicitly shared through the `threads::shared` module. It allows programmers to create shared variables with `shared` attribute or with the `share` function. The module supports sharing scalars, arrays, and hashes, as well as references to these data types. Only scalars and references to shared variables can be assigned to shared arrays and hash elements.

Nevertheless, the only guarantee that Perl provides for shared variables is that their internal state will not become corrupt in case of conflicting accesses. Applications are still vulnerable to data races when multiple threads access shared data without proper synchronization. The `threads::shared` module provides two constructs for synchronization: locks and condition variables. Locks are implemented with the `lock` function, which places an *advisory lock* on a shared variable. An advisory lock does not prevent other threads from accessing the shared variable unless they first try to lock the same variable, in which case their execution is suspended until the variable is unlocked. There is no unlock function, a locked variable is automatically unlocked when the lock goes out of scope, i.e., when the code block from where it was called finishes executing. The Perl 5 tutorial on threads cautions programmers against standard synchronization constructs (such as locks) and recommends instead the `Thread::Queue` module, which provides thread-safe queues and frees programmers the complexities associated with the synchronization of shared data access.

A remarkable exception to conventional concurrency support in dynamic languages is JavaScript. Albeit a popular language, it does not support

multiprocessing or traditional multithreading. Unlike other dynamic languages cited in this section, JavaScript was developed for client-side scripting, which probably explains why it lacks such low-level concurrency support.

Concurrency in JavaScript is based on *web workers*, which are defined in a draft HTML5 specification. Web workers are intended to run scripts in the background while the main (user-interface) thread responsible for handling visual elements and user interaction is executed. They cannot directly interfere with the user-interface, as that could lead to race conditions with the user-interface thread. For each new web worker, a new JavaScript Virtual Machine (VM) instance running on a new system thread is created. This results in no data being shared among web workers, i.e., in no default data sharing, an emerging concurrency pattern we have identified. Still, it also results in a higher cost to create them.

Web workers can communicate by means of message passing and JavaScript employs data ownership, another emerging concurrency pattern, to control concurrent access to objects whose references are shared through messages. Messages can be exchanged among web workers by means of message channels, and between web workers and their spawning scripts simply by posting messages directly to each other. The default method to send a message among web workers is to use *cloning*, i.e., to create a copy of the data comprised in the message. Alternatively, instead of cloning, messages can be sent by *transferring ownership* (sometimes also referred to as *transferable objects*), i.e., by sending a reference to an object. This approach has the benefit of avoiding the overhead to copy objects. Since data cannot be owned by more than one execution flow at a time, the sending web worker loses access to data once it transfers its ownership and any access attempt results in an exception.

JavaScript was used as a platform for the development of the River Trail data-parallel programming API [38], which explores a particular kind of immutability. River Trail introduces a new parallel array data type (`ParallelArray`) with a set of methods that define array processing patterns (`map`, `combine`, `reduce`, `scan`, `filter` and `scatter`). Operations in individual elements in a `ParallelArray` can be executed in parallel by specifying a processing pattern and an elemental function, which defines the operation to be performed. To increment by one all elements of a `ParallelArray` in parallel, for instance, a programmer could use the `map` processing pattern and an elemental function that receives a single array element and returns its value incremented by one. Elemental functions cannot communicate directly with each other, but they have read-only access to the global state. When elemental functions are executing, the parent execution flow is suspended and thus

it does not change its own (global) state. Since neither the parent execution flow, nor child execution flows spawned to run elemental functions, change the global state while elemental functions are running, River Trail calls this approach *temporal immutability*.

Clojure is a dialect of LISP, it runs on the Java Virtual Machine and builds on the Java concurrency support to offer a robust infrastructure for multithreaded programming. Although Clojure is a compiled language, it has features commonly found in dynamic languages and so it can be considered (and promotes itself) as a dynamic language. More interestingly, Clojure was designed to support concurrency. Both futures and promises are supported in Clojure, which uses an API similar to the one included in Java's package `java.util.concurrent`. However, Clojure is better known for its strong support to data immutability.

Clojure is inspired in functional programming languages and, as such, provides strong support for immutability. It offers a range of immutable persistent data structures (lists, vectors, sets, and maps). Only references mutate in Clojure, and they do so in a controlled way. Clojure supports four different types of mutable references to its immutable data structures: `Vars`, `Refs`, `Agents` and `Atoms`. They differ in how they are changed. Changes to `Vars` are isolated on a per-thread basis. Changes to `Refs` must be executed within transactions, using Clojure's Software Transactional Memory (STM) system. `Refs` allow for synchronous, coordinated state changes. `Agents` must receive functions (called actions, in Clojure), with message passing style semantics, to perform changes. Actions are executed asynchronously and can alter an agent's state. Each agent works by sequencing operations to a data structure instance. `Agents` allow for asynchronous, independent state changes. Finally, changes to `Atoms` must be executed by using simple atomic operations (`swap` and `compare-and-set`). Atoms allow for synchronous, independent state changes.

3
A Communication Model for Safe Record Sharing

We have argued that disciplined data sharing can make it easier for programmers to write correct concurrent applications. In this chapter we discuss a concurrency communication model that we propose to allow for safe record sharing in dynamic programming languages. We begin by explaining the model itself and then discuss how we implemented a prototype in Lua to experiment with the model in practice.

3.1 Model Description

Although emerging concurrency patterns are finding their way to scripting languages, there is still room for improvement. As we have shown in the previous chapter, even scripting languages that embrace emerging concurrency patterns have their limitations. Besides, recent research such as River Trail (for JavaScript) and Deterministic Parallel Ruby (DPR) indicate that promoting safe data sharing in scripting languages is a pressing matter.

In this chapter we present a brief rationale for how we have combined the emerging concurrency patterns discussed in the previous chapter to create our concurrency communication model. We also explain the model itself and examine its key concepts.

(a) Combining Emerging Concurrency Patterns

The first and most important pattern that programming languages should adopt to allow for safe data sharing is no-default data sharing. However, not sharing data by default should not be confused with not sharing data at all. As we mentioned in Chapter 1, concurrent applications can benefit from data sharing. In particular, data sharing provides better performance (because data does not need to be copied) and it is easier[1] to use.

[1]We use the term "easier" here with the same meaning as we defined in Chapter 1: posing no difficulty as long as correctness is not a concern.

No-default data sharing allows programmers to more easily identify where shared data is accessed and it facilitates the implementation of language-based concurrency control mechanisms to access shared data. Ultimately, no-default data sharing diminishes the chances of programmers inadvertently introducing data races in applications. Nevertheless, it does not eliminate data races. Take, for instance, Perl 5: by default data is local to each thread, but programmers may still declare shared data and access it without proper synchronization.

We propose that, in addition to requiring programmers to explicitly declare shared data, communication among execution flows should be localized and explicit. This combination allows programmers to more easily identify, in source code, which data is shared and how it is accessed. It simplifies reasoning about programs execution, as we have mentioned in 1.4, which is an important step towards making it easier to write correct concurrent applications.

One way to make communication explicit and localized that is consistent with the dynamic nature of scripting languages is to combine data sharing with message passing, as JavaScript does. More than simply using messages to transfer data, JavaScript uses it to transfer ownership rights: web workers can transfer the ownership of a shared object, via message passing, to allow another web worker to write to that object. Ownership rights, a form of data ownership, uses the same base concept of types and effects that we mentioned in Chapter 2. It works well for read-write accesses, but is not suitable for the common case of read-only access. To allow multiple web workers to read a shared object, a programmer must either sequentially transfer the object's ownership, precluding parallelism, or clone the object multiple times, impacting performance.

A simple and effective approach to allow multiple execution flows to read shared data is to use immutability. Recall that immutable objects are inherently thread-safe and can be freely shared. Therefore, we propose using immutability as a complement to data ownership. Combining immutability with data ownership allows both for parallel read-only access and for safe, disciplined, read-write access to shared data.

Recall, as we discussed in Chapters 1 and 2, that using data immutability by itself may not be a viable choice. Working exclusively with immutable data can cause a performance bottleneck due to the need to create new objects to alter existing objects. Also, it requires programmers to change programming patterns commonly employed in imperative languages which rely on altering data. That is why it is beneficial to support both mutable and immutable data sharing.

(b) Safe Record Sharing

Our model combines the benefits of no-default data sharing, ownership rights and immutability to allow for safe data sharing in dynamic languages. Because we adhere to no-default data sharing, by default all data is local to execution flows. The only means to share data among execution flows is to use *shareable objects*. Regular (local) objects can never be shared.

The only means to share a shareable object is message passing. We keep communication localized and explicit as only one execution flow at a time can write to a shareable object and allowing other execution flows to access changed objects always requires sharing the object. This simplifies reasoning about program execution. We avoid the performance cost usually associated with message passing, as we do not need to copy data between different address spaces. Moreover, since the only means to share data is to use shareable objects with message passing, we are effectively providing specific, well-defined constructs for communication, while compelling programmers to use them.

We build on ownership rights by combining it with immutability to improve flexibility. We propose expressing this combination by means of dynamically assigned *capabilities* [18]. Capabilities express the set of rights that execution flows have over shareable objects. Each reference to a shareable object has an associated capability. We define two basic capabilities: *read-write* (mutable) and *read-only* (immutable). The lack of capabilities means the execution flow has *no access* to an object. When an execution flow creates a shareable object, by default it has a reference with the read-write capability for the new object.

An execution flow can share an object by sending it to another execution flow via message passing. When it does so, the sender must choose whether the object should be sent as read-write or as read-only. When sending a shareable object, the following rules apply:

- when sending an object as read-write, the sender loses access to the object and the receiver gains read-write access to the object, as shown in figure 3.1;
- when sending an object as read-only, the object becomes immutable; the sender loses write access and the receiver gains read-only access, as shown in figure 3.2.

Sending a shareable object as read-write in our model leaves the sender with an invalid reference to the object that cannot be used anymore. Observe that multiple objects in the same thread might include a reference to a shareable object and they will all be unable to use it after the object is sent as

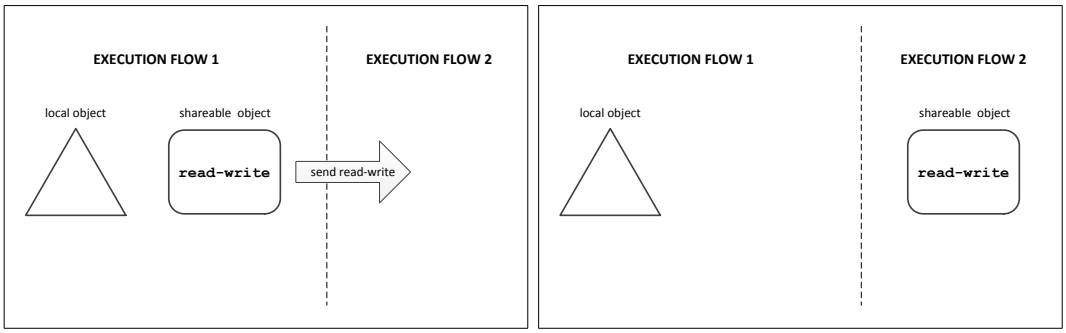

Figure 3.1: Sending a shareable object as read-write causes the sender to lose access and the receiver to gain read-write access.

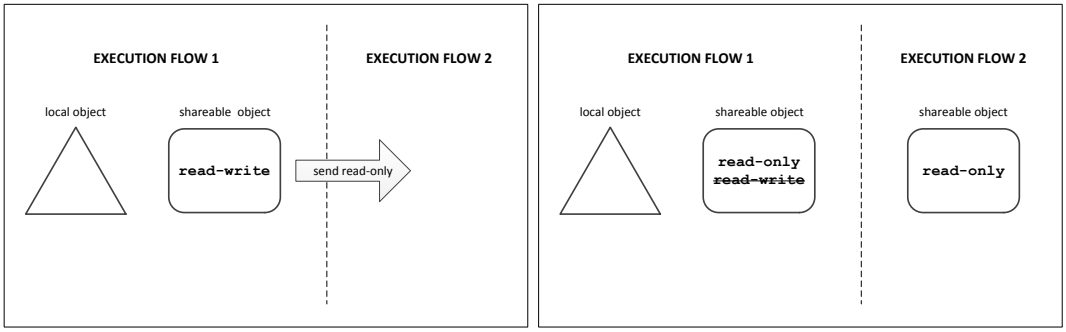

Figure 3.2: Sending a shareable object as read-only causes the sender to lose write access and the receiver to gain read-only access.

read-write, as illustrated in figure 3.3. Invalidating the reference to an object after it is sent as read-write is necessary to prevent the sender from accessing the object while the object is potentially being written by the receiver. Also, observe that in our model immutability cannot be reversed. Once an object becomes immutable, it cannot become mutable again. Using messages to assign capabilities is aligned with the nature of dynamic languages, as access control is performed during runtime.

Shareable objects can hold immutable values (strings, numbers and booleans) and other shareable objects. Nesting shareable objects does not cause capabilities to be inherited. Moreover, when shareable objects are nested, all of their capabilities must be considered when attempting to share them. Therefore, the following rules apply:

- to send an object as read-write, the object and all its inner objects must have the read-write capability;
- to send an object as read-only, the object and all its inner objects can either have the read-only capability or the read-write capability; all objects with the read-write capability will become read-only after the object is sent, as shown in figure 3.4;

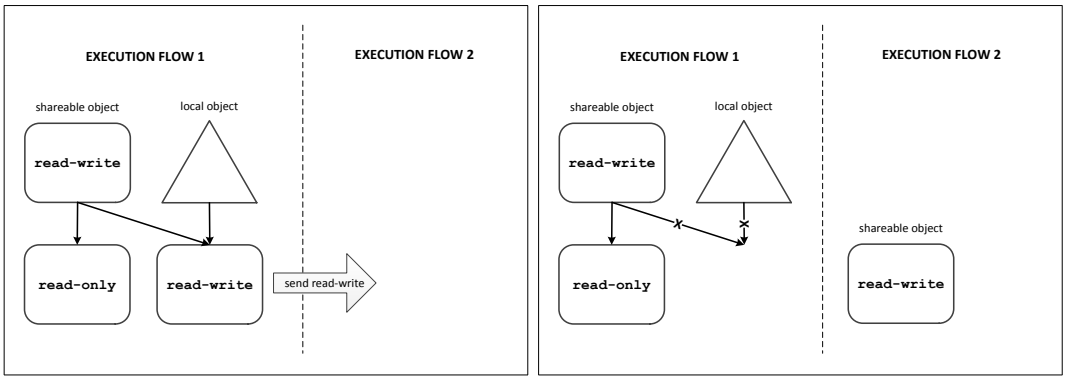

Figure 3.3: Sending a shareable object as read-write invalidates its reference in the sender. Other objects that might include that reference will not be able to use it anymore.

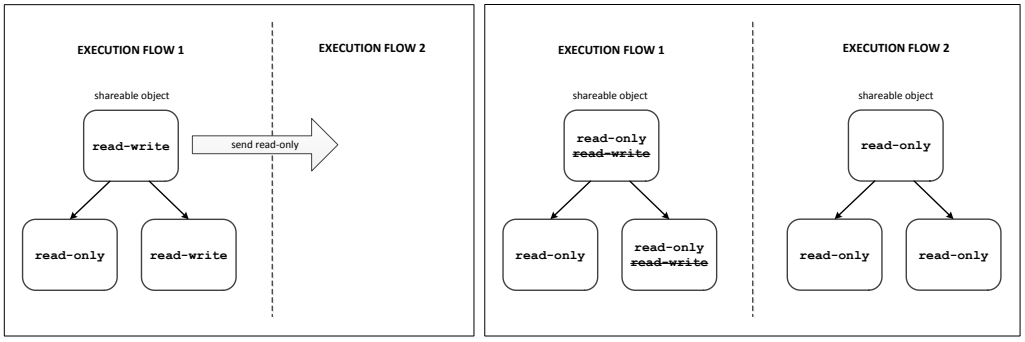

Figure 3.4: When shareable objects are nested, sending an outer object as read-only makes all objects read-only.

Overall, our model provides the means for disciplined data sharing. It keeps communication localized and explicit; it defines shareable objects, a specific construct for data sharing that works according to well-defined rules; and last, but not least, it compels programmers to use capabilities to ensure safe concurrent access to shared data. In addition, our model can be used as a building block for higher-level concurrency patterns.

3.2 Model Implementation

As part of our research, we implemented a prototype to experiment with the proposed concurrency communication model using the Lua [41] programming language. Lua is a lightweight multi-paradigm dynamic programming language. It is dynamically typed, supports collaborative multithreading and performs automatic memory management. Lua also provides a C API that allows programmers to use Lua code to extend C applications, as well as to use C code to create new functions for Lua.

The prototype is called *luashare*. Its implementation is divided in two

parts, one inside the interpreter and another outside the interpreter. The prototype allows multiple threads in Lua to be executed in parallel and to safely share data. It uses the following API:

`newthread(string code)`

Creates a new Lua thread that will execute the specified string of Lua code. Returns `true` when it is successful and `nil` plus an error message when it fails.

`getnumworkers()`

Returns the number of active workers (system threads).

`setnumworkers(number n)`

Sets the number of active workers (system threads). Raises an error when it fails.

`send(string chan_name, string capability,`
`immutable_values|shareable_object, ...)`

Sends shareable objects with specified capability and immutable values (strings, numbers and booleans) to the specified channel. Multiple shareable objects and immutable values can be sent at once. Returns `true` when it is successful and `nil` plus an error message when it fails. Blocks when there is no matching `receive` on the same channel.

`receive(string chan_name, boolean asynchronous)`

Receives shareable objects and immutable values from the specified channel. When the optional argument `asynchronous` is `true`, it does not block waiting for a matching `send` on the same channel. Returns shareable objects and immutable values sent to channel when it is successful and `nil` plus an error message when it fails.

`newchannel(string chan_name)`

Creates a new channel with the specified name. Returns `true` when it is successful and `nil` plus an error message when it fails.

`delchannel(string chan_name)`

Deletes the channel with the specified name. Returns `true` when it is successful and `nil` plus an error message when it fails.

`newobj(table t)`

Creates a shareable object using the values in the specified table. Returns a reference to the object when it is successfully and `nil` plus an error message when it fails.

`getcapability(shareable_object obj)`

Returns the capability associated with a reference to a shareable object. Raises an error when it fails.

`wait()`

Waits until all Lua threads have finished executing. Should only be called from the main Lua thread.

As a starting point for the implementation outside the interpreter, we used luaproc. We presented a brief overview of luaproc in Section 2.2, when we discussed examples of dynamic languages that employ the no-default sharing concurrency pattern. Luaproc allows parallel execution of Lua code and provides a communication model that closely matches the one that we are proposing. However, it lacks the support for data sharing and capabilities, which are central to our communication model. Before we discuss how we modified luaproc to suit our needs, we must recall how it works and explain its architecture in more detail.

(a) Luaproc

Luaproc is a concurrency library, developed in C, for Lua. It is implemented entirely outside the interpreter, allowing it to be used with the standard Lua interpreter. The key concept luaproc implements is that of *Lua processes*. Each Lua process is an execution flow of Lua code and multiple Lua processes can run in parallel. Like operating system processes, Lua processes are independent from each other, they do not share data.

Luaproc implements Lua processes with *Lua states*. A Lua state is a data structure that defines the state of the Lua interpreter, and keeps track of global variables and other interpreter-related information. Luaproc explores two properties of Lua states to implement Lua processes. The first property is that the Lua C API allows C applications to create and manipulate multiple states at a time. The second property is that Lua states do not share data among them.

Each Lua process in luaproc has its own Lua state, i.e., there is a one-to-one relation between Lua processes and Lua states. Using independent Lua states is a natural design choice for luaproc, as states do not share data and the communication model implemented by luaproc precludes shared data. Other than a design choice, using independent Lua states actually makes it very difficult to support data sharing, as it would require implementing a distributed garbage collector. Observe that if a Lua state was allowed to reference data in

another state, a garbage collection cycle in the state where the data is stored could lead to data corruption.

Lua processes are executed by a set of workers, which are system threads implemented with the POSIX thread library (pthreads). Workers and Lua processes are independent from each other. This N:M mapping between system threads and user threads is a common concurrency pattern [70, 65]. Typically it relies on a smaller number of system threads (usually just enough to allow all processor cores to be used) and a greater number of user threads (as many as needed by the application). Each worker in luaproc continuously retrieves a Lua process from a ready queue and executes it until it completes, blocks or yields. Even though workers can execute multiple Lua processes in parallel, only a single worker at a time can access an individual Lua process and its corresponding Lua state.

Lua processes communicate by using message passing. Because Lua processes are implemented with independent Lua states, which do not share data among them, luaproc must supply its own functions to allow for communication. Therefore, luaproc provides the `send` and `receive` functions. These functions work by copying immutable values (strings, numbers and booleans) between Lua states. Messages are addressed to communication channels. Each channel is identified by a distinct name and a Lua process only needs to know a channel's name to be able to use it.

The `send` and `receive` functions are the only potentially blocking operations provided by luaproc. When a Lua process tries to send a message to a channel where there are no Lua processes waiting to receive a message, its execution is suspended until there is a corresponding receive operation on the channel. The same happens when a Lua process tries to receive a message from a channel where there are no Lua processes waiting to send a message.

Figure 3.5 presents an architecture overview of luaproc. The figure shows six Lua processes (A-F) and four workers (1-4). While four of the Lua processes (A, B, D and F) are being executed by workers, the remaining two are waiting to be executed in a ready queue. Moreover, two Lua processes (A and B) are communicating by using channel 'ch'. This figure illustrates the key points that we have explained in this section and that are essential for understanding how we used luaproc to implement the part of our prototype which is outside the interpreter.

(b) Implementation Outside the Interpreter

Part of our prototype is implemented outside the interpreter. We used luaproc as a starting point to implement it. However, recall that luaproc does

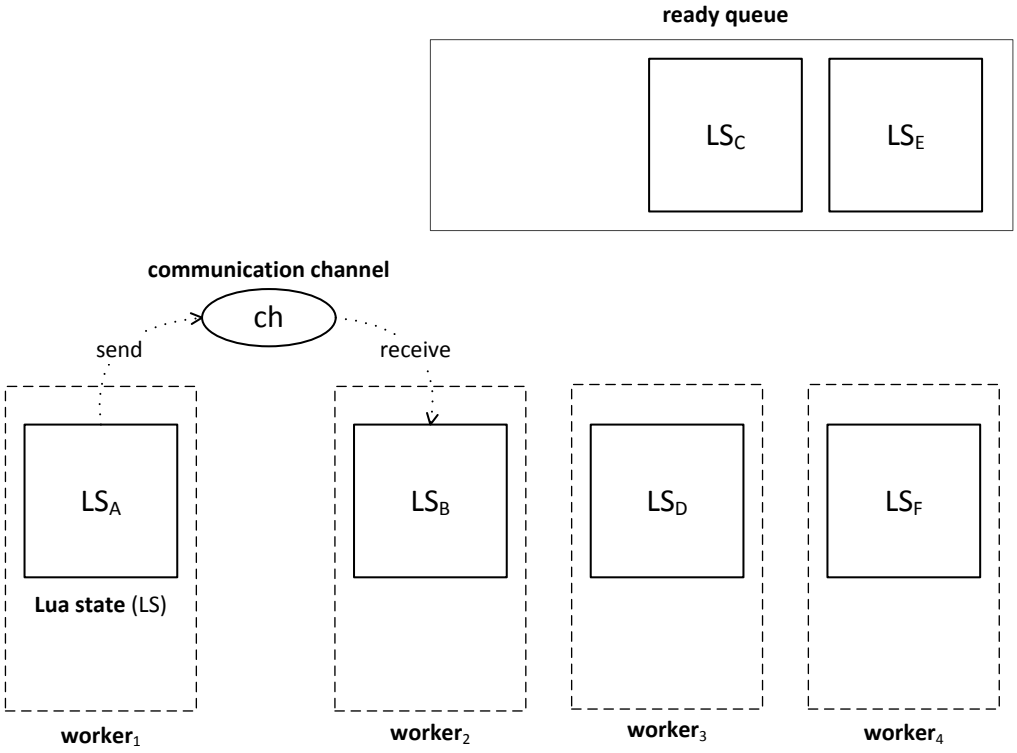

Figure 3.5: An architecture overview of luaproc.

not support data sharing, which is central to the communication model we are proposing. Consequently, our first concern was how to allow for data sharing with luaproc.

Lua supports multithreading with *coroutines*, which we briefly talked about in Section 2.2. Coroutines are lightweight userland threads scheduled cooperatively; they cannot be executed in parallel. Each coroutine in Lua has its own runtime stack. A coroutine can suspend its execution only by explicitly calling a yield function. Once a coroutine starts executing, it will run until it terminates or yields. Global variables in Lua are shared among coroutines. Therefore, despite the lack of support for parallel execution, we looked into coroutines as a promising base to introduce data sharing in luaproc.

To understand how multithreading in Lua works, we must take a closer look at the implementation of Lua states. Each Lua state comprises a *global state* and one or more *local states*. A local state stores the runtime stack of a coroutine — since the runtime stack is essentially what defines a coroutine, a local state is a coroutine. The global state stores all data that is shared by coroutines created inside the same Lua state: garbage collection control variables, a hash table to intern strings, a table that holds global variables, among others. Having a shared global state allows coroutines inside the same Lua state to share global variables in Lua.

We modified luaproc to use coroutines inside the same Lua state instead of independent Lua states. While luaproc implements Lua processes with independent Lua states that do not share data, luashare implements threads inside the same Lua state so that it has a shared global state. This was the most fundamental change we performed in luaproc. It satisfies a central property of our model, which is to support data sharing, while retaining the benefits of the original communication model, based on message-passing.

This fundamental change, however, introduced a fundamental problem. Observe that threads in luashare are executed by workers, just like Lua processes in luaproc. Nevertheless, unlike processes in luaproc, threads in luashare are inside the same Lua state (as we want to share data). This means that there are multiple local states (one of each thread) and a single global state shared among threads. Therefore, while in luaproc each worker could only access a single Lua state at a time, in luashare all workers access the same shared global state. That is why we had to provide concurrency control to the global state.

Having modified luaproc to support data sharing, we then continued to use it for message transmission in luashare. While luaproc has to copy data between Lua states, luashare can simply exchange data between threads using the shared global state. Thus, we changed the behavior of the `send` and `receive` functions in luashare accordingly. Nevertheless, to make communication in luashare comply with our model, we needed to implement shareable objects and capabilities.

We implemented shareable objects with tables in Lua. The `table` type defines an associative array that can be indexed with any Lua value and can hold values of any type. Because tables are the only data-structuring mechanism provided by Lua, they were an obvious choice for us to implement shareable objects. Luckily, tables proved to be flexible enough to suit our needs. A shareable object is simply a table that holds the values that compose the object and an associated capability. Recall that shareable objects can only hold immutable values (strings, numbers and booleans) and other shareable objects, as we explained in the previous section.

Because shareable objects, as their names imply, are intended to be shared among threads, we must control access to ensure concurrency safety. To do so, we used a Lua feature that allows the behavior of operations over individual tables to be changed. In particular, we defined our own functions to handle the indexing access operation, which happens when a value in a table is accessed, and the indexing assignment operation, which happens when a value in a table is assigned. Our functions enforce capabilities and prevent unsafe

concurrent access.

Besides controlling access to shareable objects, we must also control how they are shared. The modified `send` function in luashare sends only shareable objects and it requires programmers to specify how objects should be shared (either read-write or read-only). The function checks whether objects and nested objects have the necessary capabilities before sending them.

Recall that in luashare multiple workers access the shared global state. That is why we must implement concurrency controls to prevent data corruption in the Lua state that holds it. However, that cannot be done outside the interpreter, as it requires access to the internal data that defines the global state in Lua. That is why we needed to modify the Lua interpreter, which is what we discuss in the next section.

Before we move on, though, we present an architecture overview of luashare in figure 3.6. The figure shows three Lua threads (A-C) being executed by three workers (1-3). It also shows that all threads (L) are inside the same Lua state and thus share the same global state (G).

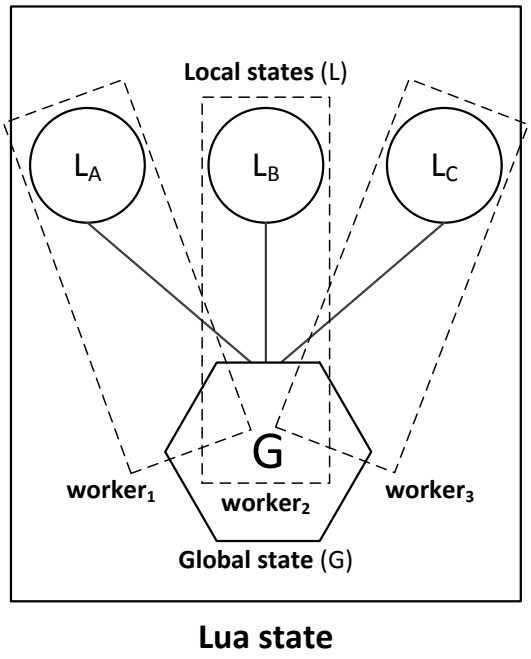

Figure 3.6: An architecture overview of luashare.

(c) Implementation Inside the Interpreter

Luashare creates multiple threads inside the same Lua state. Recall that threads inside the same Lua state share the same global state, which is what allows them to share data and thus explains why we chose this implementation

strategy. To ensure that we had proper concurrency control, we had to look into the global state, which is implemented by the Lua interpreter.

The global state is a graph. Each node in this graph represents some data in C. More importantly, most of the nodes have a one-to-one relation with objects in Lua. We exploited this relation to implement concurrency control for the global state. To understand how, we must first consider garbage collection in more detail.

In garbage collection parlance, systems are divided in two parts. The first part, called the *collector*, executes the garbage collection code; it discovers unused memory and reclaims it. The second part, called the *mutator*, executes the user code[2]; it allocates objects and mutates (changes) them.

In Lua, the mutator is the Lua application being executed by the interpreter. Consequently, the only objects it mutates are Lua objects. Because our implementation outside the interpreter controls concurrent access to these objects (using shareable objects and capabilities) and there is a direct relation between data in C and data in Lua, concurrency control for the corresponding data in C is not necessary. Therefore, for all nodes in the global state that have a one-to-one relation with objects in Lua, luashare provides the concurrency control.

We need to provide concurrency control in C for the collector, whose inner workings we cannot control from Lua, and for any global state data that does not have a one-to-one relation with objects in Lua. When not performing a collection cycle, the garbage collector uses a number of fields in the global state to control its activities, in particular to keep track of object creation. We used mutual exclusion to control concurrent access these fields, as we will explain later in this section. In addition, while performing a collection cycle, the garbage collector can potentially access any object in Lua; therefore, it should not be executed in parallel with other threads. To prevent data corruption, we implemented a synchronization barrier to stop all threads before performing a collection cycle.

Once a thread decides it must perform a collection cycle, it stops at the barrier and waits for all other threads. After all threads have stopped at the barrier, a single thread then resumes its execution and performs a collection cycle. Once the thread finishes the cycle, it notifies the remaining threads that were waiting at the barrier that they can resume execution. The notified threads skip the garbage collection cycle, which has already been performed, and continue executing. We do not expect the barrier to have a significant

[2]From the point of view of the garbage collector, the mutator contains "all the rest" of the system, including the user code.

performance impact. However, we are aware that in case a thread is blocked executing an I/O operation, it will hold back all the other threads in the barrier. As this is a very particular scenario, we explicitly chose not to deal with it in the scope of this work.

Besides providing concurrency control for the garbage collector, we also had to provide concurrency control for the data in the global state that had no direct relation with objects in Lua. We analyzed each of the fields that compose the global state structure, which defines the header of the graph, to determine how they were used. For each field, we took one of the following three approaches:

1. if the field was written to only once (during the Lua state initialization) then we did nothing about it;

2. else, if the field did not need to be global, then we moved it from the global state structure to the local state structure;

3. else, we used mutual exclusion to control concurrent access.

Overall, we moved three fields from the global state structure to the local state structure, we did nothing about eight fields that were written to only once and we had to use mutual exclusion to prevent concurrent access to the remaining 23 fields. Out of these 23 remaining fields, 22 were used to control garbage collection and one was used to store a hash table.

We provided concurrency control to these 23 fields with three standard POSIX mutual exclusion variables (`pthread_mutex_t`). The first mutual exclusion variable (mutex) controls the concurrent execution of functions that handle Lua objects and thus update fields used to control the garbage collector. It is complementary to the synchronization barrier which we talked about earlier in this section. These functions concentrate accesses to most of the 23 fields of the global state that needed concurrency control. There were only two exceptions, that we handled with the other two mutexes. The second mutex controls concurrent access to a hash table that is used to store the hashes of strings created in Lua. Finally, the third mutex controls concurrent updates to a counter that the Lua garbage collector uses to determine when it should perform a collection cycle.

(d) A Simple Example Application

Now that we have explained how we implemented luashare, we can present a simple example of a how an application that uses it looks like.

Listing 3.1 shows the source code for a simple application, along with some inline comments.

As the listing shows, we begin by loading the part of luashare that is implemented outside the interpreter (line 2) and by increasing the number of workers, or system threads, to two (line 5). Next, we create a communication channel (line 8) and two shareable objects (lines 11 and 12). Then we create a new thread (lines 17 to 31) that will receive the two objects (lines 21 and 23), print the values stored in each of them (lines 25 and 26), update a value in one of the objects (line 28) and then send the updated object back (line 30). To create the thread, we pass a string of Lua code that will be executed as an argument to the `newthread` function – the `[[` and `]]` symbols can be used to define literal strings in Lua.

After creating the communication channel, the shareable objects and a thread, we begin to share objects. We share the first shareable object as read-only (line 34) and then, right after, we try to change one of its values (line 37) just to check we are not allowed to. Next, we share the second shareable object as read-write (line 41) and, right after, we try to read one of its values (line 44), again to check that we are not allowed to. Finally, we receive the updated shareable object (line 48) and print its values (line 51).

Listing 3.2 shows the output that results from executing the application with luashare. Observe that we receive errors when trying to update a shareable object as read-only and when trying to access a shareable object sent as read-write (lines 2 and 3). Also, observe that the value in the shareable object that was shared as read-write was correctly updated (line 6).

(e) Development Experience

During the development of our prototype implementation we were faced with common difficulties that are typical of concurrency with preemptive multithreading and shared memory. In this section we present a brief and informal report on some of these difficulties, as well as how we handled them, as they evidence some of the problems that we are trying to solve with our research.

The first and most time-consuming problem we had to deal with were data races. Throughout the development of our prototype, we were faced with a *multitude of unexpected program behaviors* that would appear at different points in code, often resulting in segmentation faults. Most of the time it would be nearly impossible to reproduce these behaviors consistently. We used standard debugging tools, such as the GNU debugger (GDB) and the tools

```
-- load luashare
luashare = require "luashare"

-- create an additional worker
luashare.setnumworkers(2)

-- create a communication channel
luashare.newchannel("a_channel")

-- create two shareable objects
a_so = luashare.newobj{x=31, y=33, z=7}
another_so = luashare.newobj{"ping"}

-- create a new lua thread that will receive a read-only shareable object
-- and try to write to it and then will receive a read-write object, write
-- to it and send the updated object back.
luashare.newthread([[
  -- load luashare
  luashare = require"luashare"
  -- receive the first object
  a_recv_so = luashare.receive("a_channel")
  -- receive the second object
  another_recv_so = luashare.receive("a_channel")
  -- print received values
  print("a_recv_so:\t", a_recv_so.x, a_recv_so.y, a_recv_so.z)
  print("another_recv_so:",another_recv_so[1])
  -- change a value in the shareable object received as read-write
  another_recv_so[1] = "pong"
  -- send the updated object back
  luashare.send("a_channel", "rw", another_recv_so)
]])

-- share the first inner object as read-only
luashare.send("a_channel", "ro", a_so)

-- try to change the object sent as read-only
ret, err = pcall( function() a_so.x=32 end )
if (not ret) then print(err) end

-- share the second inner object as read-write
luashare.send("a_channel", "rw", another_so)

-- try to read the object sent as read-write
ret, err = pcall( function() print(another_so[1]) end )
if (not ret) then print(err) end

-- receive the updated object that was shared as read-write
an_updated_so = luashare.receive("a_channel")

-- read the updated value
print("an_updated_so:\t", an_updated_so[1])

-- wait until all threads have finished
luashare.wait()
```

Listing 3.1: A simple example application implemented using luashare.

```
$ luashare simple.lua
cannot change an immutable object
cannot access shareable object (sent rw)
a_recv_so:              31      33      7
another_recv_so:        ping
an_updated_so:          pong
$
```

Listing 3.2: The output that results from executing the simple application implemented using luashare.

included in the Valgrind framework, as well as a lot of print debugging, to try to understand and deal with these behaviors as best as we could.

Although debugging is an integral part of programming, what differentiates concurrency bugs from regular bugs is the time it takes to understand and fix them. While making the changes to allow multiple threads in Lua to run in parallel took a few days, understanding and fixing all the data races we encountered took weeks. Besides, even after we implemented a potential fix, it was difficult to ensure that all possible execution interleavings that could lead to the race were covered.

Other than data races, we also had to deal with many deadlocks. The advantage of deadlocks over data races is that deadlocks are always immediately visible when they occur. Most deadlocks we identified were caused by multiple functions in a call path trying to lock the same mutex. Although conceptually simple to understand, dealing with this problem in practice proved to be demanding. Even when working with a single mutex, we had to reason about multiple function call paths and most functions in Lua are recursive, which makes matters worse.

A common workaround we employed when dealing with deadlocks was to implement two versions of the same function: while one version locked the mutex before executing the function's body, the other version simply executed the function's body without locking the mutex. The first version could be used by functions that did not lock the mutex beforehand, while the second version could be used by functions that did lock the mutex beforehand.

Another cause for deadlocks were multiple mutexes being locked in different orders. In this case, not only the problem is simple to understand, but so is the conceptual solution: to ensure that mutexes are always locked in the same order. Nevertheless, applying the solution in practice was not simple, mostly because each mutex was locked by a different set of functions. Besides, reasoning about deadlocks that involve multiple mutexes is generally harder than reasoning about those that only involve a single lock. In our experience, even when working with a small number of mutexes, complexity escalates quickly.

4
Evaluation

In this chapter we evaluate the proposed communication model in practice, using luashare. The evaluation consisted of the implementation and execution of an artificial benchmark application, as well of a few real-world applications, using luashare. Our goal was to have an idea of how using shareable objects facilitates safe data sharing and to analyze the performance of our implementation. In the first section we present some general and more informal remarks about the evaluation and then, in the following section, we detail the results of the performance analysis.

4.1 General Analysis

Overall, our experience with luashare as users was better than our experience as its developers. While we had to deal with a number of intricate data races that were time consuming to debug when developing luashare, all of the concurrency errors we had to deal with when using luashare were straightforward and quick to debug. Common errors we had to deal with when using luashare included unmatched send and receive operations, forgetting to define a capability when sending objects, mistakenly sending incorrect values and using incorrect channel names. Observe that all of these errors revolve around, or at least are evidenced by, calls to the send and receive operations, which are used for communication among threads. Because these calls are explicit and localized in code, it is easier to debug errors such as the ones we encountered.

The ability to use strings to index values in shareable objects makes it intuitive to access specific values in objects. Besides, being able to use shared data makes it easier to implement communication among execution flows, as there is no need to serialize and deserialize data every time it needs to be shared. More importantly, because shared data does not need to be copied among threads, luashare has better scalability when working with large data sets that need to be accessed by multiple threads.

Still regarding communication, we found that the use of capabilities

to control access to shared data was intuitive. Moreover, we found that even though capabilities strictly define how shared data can be accessed, they still provide enough flexibility to allow for different communication strategies. When using domain decomposition, for example, at least the following alternatives for communication among threads can be used:

- each thread receives the whole input data set as a read-only shareable object, creates and fills a new shareable object with its partial results and then sends back this object as read-only (or read-write);
- each thread receives a part of the input data set as a read-write shareable object, fills this shareable object with its partial results (by changing its values in-place) and sends it back as read-write (or read-only).

Likewise, when using functional decomposition, for example to implement a pipeline, each thread in the pipeline can receive a read-write shareable object with the results of previous stages of the pipeline, fill this shareable object with its results (by changing its values in-place) and send it as read-write to the next thread in the pipeline.

Enforcing disciplined data sharing in luashare, however, comes at a cost. In particular:

- to create a shareable object, we must check the types of all values it is supposed to hold; if there are tables among these values, we must traverse them;
- to send a message, we must check whether all values contained in the message are shareable objects; we must also check whether each shareable object has the appropriate capability to be shared as specified and, when necessary, change capabilities accordingly; if there are nested shareable objects among these values, we must traverse them.

Observe that after a shareable object is created and before it is shared, we do not control how the object is manipulated. Therefore it is possible, for instance, to insert a value of an unsupported type in a shareable object after the object is created. However, luashare will not allow such object to be shared.

The lack of control over how shareable objects are manipulated before they are shared demands stricter controls for effectively sharing them. That is why, besides checking messages to ensure they only contain shareable objects, we must also check values inside each shareable object before allowing them to be shared. Having stricter controls to share objects while relaxing control over how they are locally manipulated by each thread means that it is comparatively

more expensive, in terms of performance, to share an object than it is to manipulate it.

We could have taken a different approach and implemented stricter controls for local object manipulation while relaxing controls to effectively share them. Nevertheless, our experience during the evaluation showed us that the number of times a shareable object is accessed typically outweighs the number of times it is shared. Thus, from a performance point of view, it makes more sense to make accesses as fast as possible and to transfer the burden of control to the point when objects are effectively shared.

Regardless of the fact that it makes sense to have stricter controls when sharing objects, this approach has a drawback. When sharing a nested shareable object, all its inner objects must be traversed, which can be slow depending on the number of objects and their sizes. Since nesting is a common technique to implement data structures such as graphs and trees, this represents a limitation in luashare.

A possible work-around for that limitation is to use regular Lua tables to create data structures and then use shareable objects to hold coarser-grained parts of the structures that will be shared. Consider, for example, a square matrix with N rows. Intuitively, a programmer would probably store each row in a different Lua table and store all the row tables in a single table. However, instead of using a shareable object for each row, the programmer could use a single shareable object to hold multiple rows, reducing the number of shareable objects to make communication faster.

Independently of where controls are applied, the overall performance of luashare is invariably very sensitive to how these controls are implemented. Consider, for example, a scenario where every time a shareable object is accessed a function must be called to check the object's capability. The function can potentially be called many times during the execution of a program that uses large shareable objects. Therefore, changing the function to include or exclude a single operation or function call, for instance, can have a significant impact in the performance of the program.

In practical terms, throughout the evaluation, we managed to reduce the times to read values from and write values to shareable objects by up to 10 times with very simple changes to the routines that enforce capabilities. Examples of such changes include removing string concatenation operations and storing references within functions instead of using function calls to retrieve them. In the end, access times to shareable objects were, in order of magnitude, comparable to access times to regular Lua tables. We discuss performance in more details in the next section.

4.2 Performance Analysis

We conducted a performance evaluation to assess the performance of our implementation and to verify that our model can be used in practice. To do so, we implemented and executed the following applications with luashare:

Black and Scholes — a financial application, ported from the PARSEC [5] benchmark suite, that calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation;

FASTA Parser — an application to search a FASTA[1] sequence for patterns;

CAPTCHA Filter — an application to filter CAPTCHA[2] images to make it easier to perform automatic optical character recognition (OCR).

We selected these applications because they explore different parallelization and data partitioning patterns. Black and Scholes explores sharing structured data as read-only among threads that read different pieces of shared data. The FASTA Parser explores sharing unstructured data as read-only among threads that read the same piece of shared data. The CAPTCHA filter explores sharing data as read-write among threads that work as pipeline.

While searching for real-world applications to implement, we analyzed applications from two benchmark suites: PARSEC [5] and the Java OpenMP (JOMP) version of the Java Grande Forum Benchmark Suite (JGF) [16]. We ported the Black and Scholes application from PARSEC. It is the simplest application included in the suite. We chose not to use the remaining applications in PARSEC because they were too complicated to port. We chose not to use any applications from the JGF suite because they all use the same parallelization strategy, which is to partition data and have threads access disjoint sets of data, and we already explore that case with the Black and Scholes application.

Apart from the real-world applications, we also implemented an artificial benchmark application to specifically analyze the performance of sharing objects. This application, which we called ping-pong, reads data from a file and uses it to create a shareable object. Then, it creates two threads that send the object to each other, back and forth, a number of times.

For each application, except ping-pong because it would not make sense, we implemented a serial version using standard Lua and parallel versions using both luashare and luaproc. Unless explicitly stated, we always took the same

[1]The FASTA format is text-based format, used in bioinformatics, for representing either nucleotide sequences or peptide sequences

[2]CAPTCHA stands for "Completely Automated Public Turing test to tell Computers and Humans Apart". CAPTCHA images show distorted texts that users must type to prove they are humans to a computer system.

approach to decompose data or functions when parallelizing applications with luashare and luaproc. When we needed to serialize data in luaproc, we used the luabins library [54].

We used the serial version of the applications to compare the standard Lua interpreter with the modified interpreter used in luashare. During our evaluation, we observed no significant differences in execution times. This suggests that, as we predicted, the barrier we used to synchronize garbage collection generally does not imply a significant performance impact.

We executed all applications in a server with four Intel® Xeon® Processors E5-2690 v2 (25M Cache, 3.00GHz), for a total of 40 cores, 128GB RAM and 600GB SA-SCSI 15K RPM hard drives in RAID 1. The server had Ubuntu Linux 14.04 LTS Trusty Tahr (64 bit) installed with only essential services running. Each application was executed five times and we used the means to consolidate results. We also calculated the relative standard deviations. In the following subsections we discuss the results for each application.

(a) Ping-pong Benchmark

The ping-pong application is an artificial benchmark application we implemented to evaluate the performance of communication among threads in luashare and processes in luaproc. It reads data from an input file then creates two execution flows that send input data to each other continuously for a number of times. Because only one execution flow at a time is active, we executed this application using a single system thread.

We implemented two versions of the ping-pong application. The first version reads input from an ASCII file and uses the file contents to define the data that will be used for communication among execution flows. In luashare, we place input data in a shareable object, together with some metadata (such as the number of times data should be shared among threads and what capability should be used for sharing). In luaproc, we simply send a string with the input data. We executed this first version using as input two different files, a small one with 1,022 bytes and a larger one with 4,139,780 bytes. Also, when using luashare, we shared data as read-only and as read-write, to check whether there are significant differences in execution times.

We executed the application gradually increasing the number of times data should be exchanged among execution flows. We estimated the data throughput using the total execution times, input sizes and number of times data was exchanged among execution flows. Table 4.1 presents the results for luashare and luaproc.

Suite	Capability	Input size (bytes)	Throughput (GB/s)	σ (%)
luashare	read-only	1,022	0.11	0.64
		4,139,780	460.12	0.25
	read-write	1,022	0.07	1.00
		4,139,780	289.69	1.27
luaproc	-	1,022	0.94	1.12
	-	4,139,780	5.65	0.87

Table 4.1: Throughput for communication among execution flows in luashare and luaproc.

As we can observe, for the small file, luaproc has a higher throughput. That is explained by the fact that creating a shareable object has an overhead which, in this case, is probably creating a bottleneck. For the large input file, luashare has a much higher throughput than luaproc. The throughput difference reflects the cost of copying, instead of sharing, data. We can also observe that, when using luashare, sharing data as read-only results in higher throughput. That difference can be explained by looking at how we implemented sharing.

To share an object, we must first traverse all its values to check that there are only supported data types. To share an object as read-only we must also set the capability to read-only and block write access to the object. To share an object as read-write, we must also check whether the object has the read-write capability and then invalidate the reference to that object. Setting the read-only capability and blocking write access essentially takes less operations to perform than checking for write access and invalidating references. Since sharing is executed many times in this benchmark, even a few operations more or less can impact performance. Therefore, because sharing an object as read-write is comparatively more expensive, throughput is lower.

The second version of the ping-pong application reads input from a structured ASCII file. To keep things simple, we reused code from the Black and Scholes application to read options data from a file. Input data, just like the original Black and Schole application, is stored in shareable objects. Recall that options data is represented in a matrix, therefore we nest shareable objects to store it. We executed the second version with the same input set as we used for the first version. However, instead of measuring file sizes in bytes, we look at how many options each file describes: the small (1,022 bytes) file describes 16 options, while the larger file (4,139,780 bytes) describes 65,536 options. We did implement this second version for luaproc, as we understood it would not make sense to compare the throughout for sending shareable objects and for

sending serialized tables.

Like we did with the first version, we gradually increased the number of times data should be exchanged among execution flows. Then, we estimated the data throughput using the total execution times, input sizes and number of times data was exchanged among execution flows. This time, though, we measured throughput in objects shared per second instead. Table 4.2 presents the results.

Capability	Input size (shareable objects)	Cycles (#send/recv)	Throughput (objects/s)	σ (%)
read-only	16	1,000	63,670	1.03%
		10,000	62.661	1.30%
		100,000	62.763	0.65%
	65,536	1	21.644	1.92%
		10	45,610	1.40%
		100	51.352	2.74%
read-write	16	1,000	49.133	1.21%
		10,000	49.941	1.04%
		100,000	49.804	2.33%
	65,536	1	17.892	1.27%
		10	33.756	1.51%
		100	35,590	2.43%

Table 4.2: Throughput for sharing shareable objects among threads in luashare.

As we can observe, once more the throughput for sharing data as read-write is lower than the throughout for sharing it as read-only. We have already explained, previously in this subsection, why that happens. We can also observe that, as expected, the throughput is higher when working with a smaller number of objects. Moreover, we can observe that as we increase the number of cycles when working with the large input, throughput does not change significantly.

Besides evaluating the performance of communication, we took advantage of the ping-pong application to evaluate the performance cost of enforcing safe concurrent access to shareable objects. To do so, we developed a fork of luashare using the same implementation inside the kernel, but changing the implementation outside the kernel to remove all controls over shareable objects. More precisely, we stopped checking whether shareable objects only held immutable values and shareable objects, as well as we stopped checking capabilities to control access to shareable objects (although we do set them accordingly). Then, just like in the previous test, we executed the ping-pong

application that reads data from a structured ASCII file and estimated the data throughput in objects shared per second. Table 4.3 presents the results.

Capability	Input size (shareable objects)	Cycles (#send/recv)	Throughput (objects/s)	σ (%)	Speedup
read-only	16	1,000	566.667	0.00%	8.90
		10,000	532.915	1.31%	8.50
		100,000	529.101	2.61%	8.43
	65,536	1	97.186	1.49%	4.52
		10	321.102	2.64%	7.04
		100	458.621	2.34%	8.93
read-write	16	1,000	177.083	5.71%	3.60
		10,000	175.983	2.82%	3.52
		100,000	178.216	2.90%	3.58
	65,536	1	54.478	2.13%	3.04
		10	122.134	2.71%	3.62
		100	150.857	3.32%	4.24

Table 4.3: Throughput for sharing shareable objects among threads in luashare with no controls to prevent unsafe concurrent access to shareable objects.

As we can observe, throughput increases significantly when controls are removed. The last column in table 4.3 shows that speedups relative to the standard luashare, with all controls enabled, range from just over three times to almost nine times. Speedups are higher when sharing objects as read-only. That can be explained by the fact that sharing an object as read-only is cheaper, in terms of performance, than sharing as read-write – when we share objects as read-only we reuse references, while when we share as read-write we create new references for each object. Therefore, the cost of control when sharing objects as read-only contributes more significantly to the total execution time than when sharing as read-write. Overall, the results in table 4.3 evidence, as we have stated before, that control comes at a cost.

(b) Black and Scholes Benchmark

We ported the Black and Scholes application implemented in C in the PARSEC benchmark suite to Lua. The application prices a portfolio of options with the Black-Scholes partial differential equation (PDE). It uses a synthetic input, provided with PARSEC, that is based on the replication of 1,000 real options. The input consists of a structured ASCII file, where each line provides information divided in nine columns about an option.

After reading the input, the application then calculates the prices for each option 100 times[3]. In the serial version, the application simply iterates over all options and calculates their prices individually. In the parallel version, the application divides options among threads and each thread calculates the prices for a different range of options. Despite the division of responsibilities, all threads receive all input data.

In luashare, input data is read by a master thread into shareable objects which are then shared with worker threads that calculate the prices. Results are also returned from worker threads to the master thread by using shareable objects. Both input data and results are shared as read-only. In this case, the master thread only prints results, so it really does not make any difference whether results are shared as read-only or read-write. In luaproc, we simply used regular Lua tables instead of shareable objects and serialized them for communication among processes.

We built our own input set, based on the input provided with PARSEC, containing three files with 10,000, 100,000 and 1,000,000 options. Observe that PARSEC uses a file with 65,536 options as its big data set[4]. For each input file in our set, we executed the serial application and the parallel applications with different number of workers.

Figure 4.1 presents the speedups observed in luaproc and luashare. All of the execution times we measured had relative standard deviations from 0.66% to 6.65%.

As we can observe, increasing the number of workers in luashare yields speedups from around 2x up to just over 3.5x. Meanwhile, increasing the number of workers in luaproc yields speedups from around 2x up to just over 9x. So, while increasing the number of workers up to a point does yield a speedup in luashare, using more workers progressively slows down execution times. To understand why that happened, we must analyze how threads communicate in the application.

The initial input data, which consists of one line per option with nine columns each, is read as a whole from a file. However, after being read, part of the data is separated to be shared in six different shareable objects. Each shareable object corresponds to a columns in the input file. Therefore, there are six shareable objects, each with a number of values equal to the number of options in the input file (10,000, 100,000 or 1,000,000 in our input set). The master thread must share each of these objects with each worker thread. Recall that the options are divided among worker threads, so as we increase the

[3]This number is hard-coded in the PARSEC implementation.
[4]In PARSEC input data sets are divided according to their size.

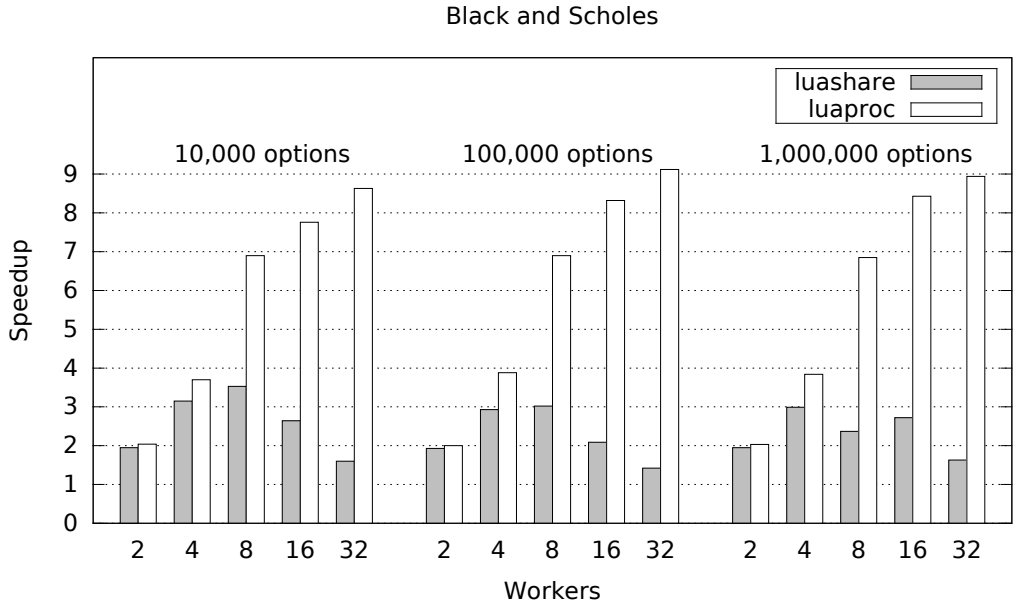

Figure 4.1: Speedups observed when running the Black and Scholes application with multiple workers and different inputs using luashare and luaproc. Higher results are better.

number of workers, we also increase the number of times objects are shared. Consider, for example, an input size of 1,000,000 options and 32 workers. The master thread will share six objects, each with 1,000,000 values, with 32 threads.

As we have discussed previously in this section, we chose to implement stricter controls to share objects, while relaxing controls to access objects. Remember that before sharing an object we must check all its values to ensure there are only supported types. The results of this benchmark evidence the cost of such approach. When using more than 8 workers, the master thread takes too long to distribute data to worker threads. Therefore, worker threads delay their execution waiting for data and the master thread becomes a bottleneck as it cannot share data fast enough to keep worker threads busy.

(c) FASTA Parser Benchmark

The FASTA Parser is an application inspired by similar existing applications to search for patterns in FASTA sequences. It essentially works like a pattern matching application: it reads input data from one file and patterns to be searched from another file, then it searches the input for the patterns. Sequences are very long strings containing letters and patterns, in our implementation, are any text matching patterns supported by Lua.

We built our own input set based in real sequences. It comprises the

complete genome of a bacteria, a string with 5,801,598 characters, a shotgun sequence of the genome of a rodent, a string with 77,654,946 characters, and finally a shotgun sequence of a human chromosome, a string with 250,522,664 characters. We chose a random sequence (TTAGGAAA) that yielded matches in all three sequences in our set. When experimenting with multiple patterns, we simply repeated the same pattern multiple times.

In our implementation using luashare, a master thread reads the search patterns and the entire input sequence as a whole. Then, it places the sequence in a shareable object that has two values: the name of the sequence (a short string), and the sequence itself (a very long string). Next, the master thread creates worker threads and shares the sequence a read-only with each of the worker threads. It then continuously creates a new searcher thread and sends a single pattern to the created thread. Once a searcher thread receives a pattern, it searches for it in the input sequence and stores results in a local table. After it finishes searching for a pattern, it places consolidated search results in a shareable object which is shared with the master thread.

In our implementation using using luaproc, we work with the sequence and its name as two separate strings. The master thread sends these strings and then a pattern to each worker thread. After searching for the pattern, each worker sends the matching results back as a serialized table.

Figures 4.2, 4.3 and 4.4 present the speedups observed in luaproc and luashare. All the execution times we measured had relative standard deviations of up to 10.02%.

As we can observe, increasing the number of workers in luashare yields speedups up to just over 14x when searching the small (bacteria) sequence, 9x when searching the medium (rodent) sequence and 8x when searching the large (human) sequence. Meanwhile, increasing the number of workers in luaproc yields speedups up to just over 5x when searching the small sequence, just over 7x when searching the medium sequence and almost 8x when searching the large sequence.

The speedups for luashare are clearly higher than luaproc when searching the small sequence, in particular as we increase the number of patterns. Recall that we create one thread per pattern, so when searching for 500 patterns we create 500 threads. In Lua, creating a thread (as luashare does) is faster than creating a new Lua state (as luaproc does to create its Lua processes). Therefore, that contributes to the higher speedups when using luashare. Notice that, as we increase the size of the input sequence, the difference between luashare and luaproc reduces as the time needed to search for patterns influences more significantly the total execution time.

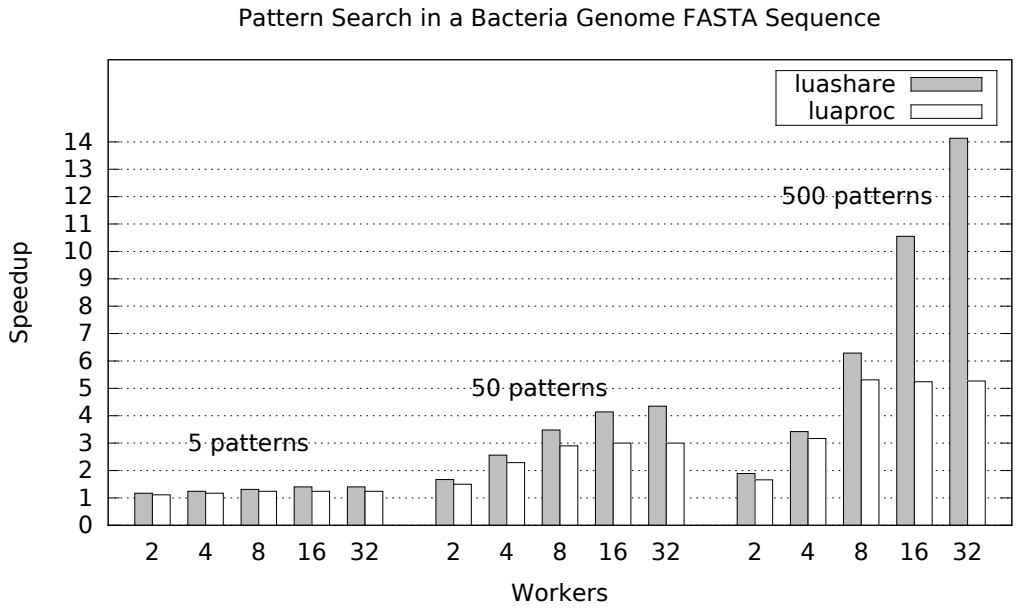

Figure 4.2: Speedups observed when running the FASTA Parser application with multiple workers and searching for a different number of patterns in a bacteria genome sequence using luashare and luaproc. Higher results are better.

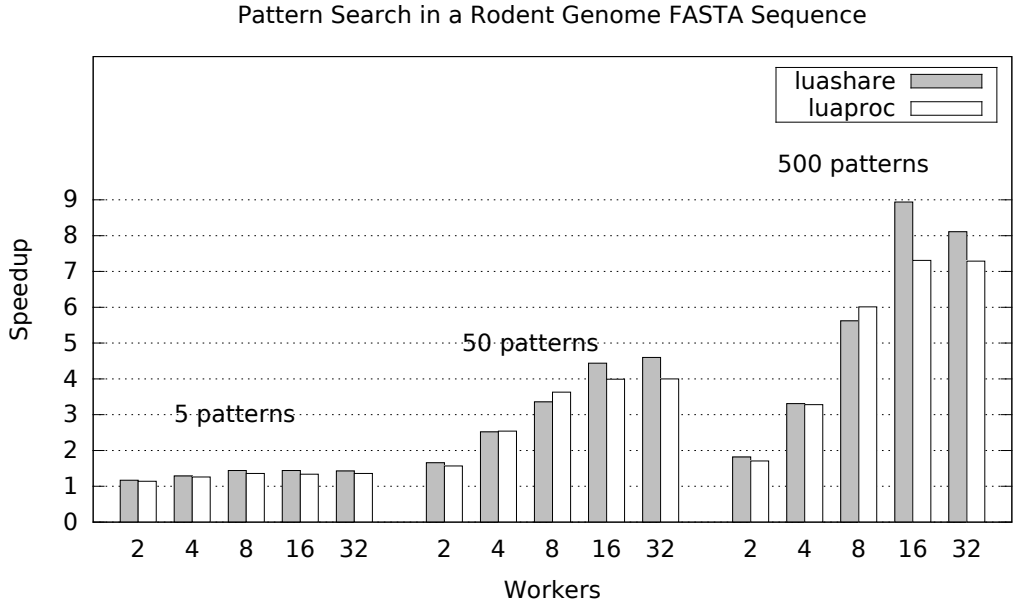

Figure 4.3: Speedups observed when running the FASTA Parser application with multiple workers and searching for a different number of patterns in a rodent genome sequence using luashare and luaproc. Higher results are better.

In this test we worked with large inputs and a higher number of threads (one per pattern), therefore besides measuring total execution times we also measured memory use. Figure 4.5 presents the memory use observed in luaproc and luashare. The relative standard deviations were up to 5.97%.

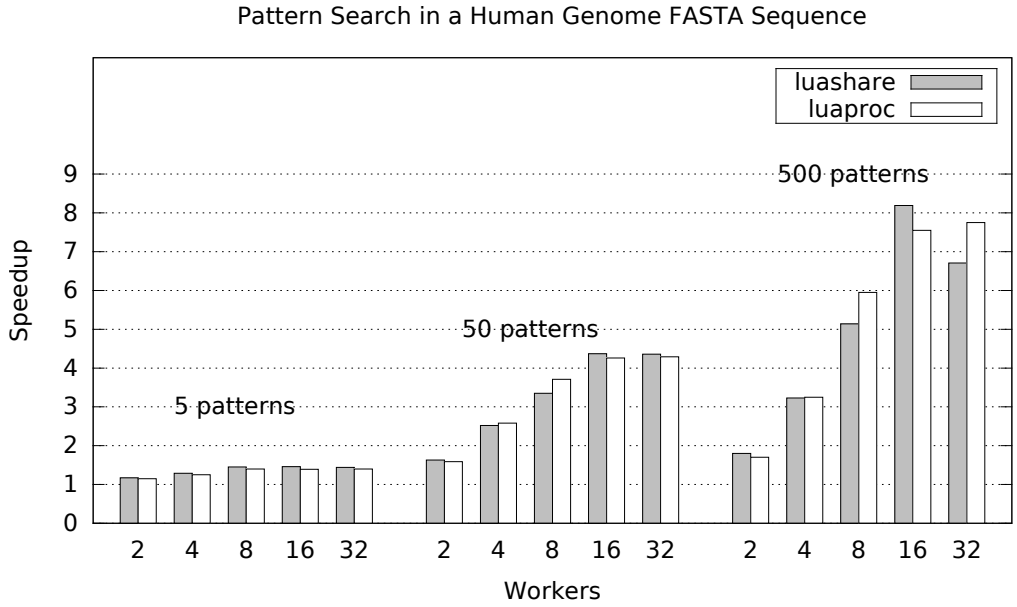

Figure 4.4: Speedups observed when running the FASTA Parser application with multiple workers and searching for a different number of patterns in a human genome sequence using luashare and luaproc. Higher results are better.

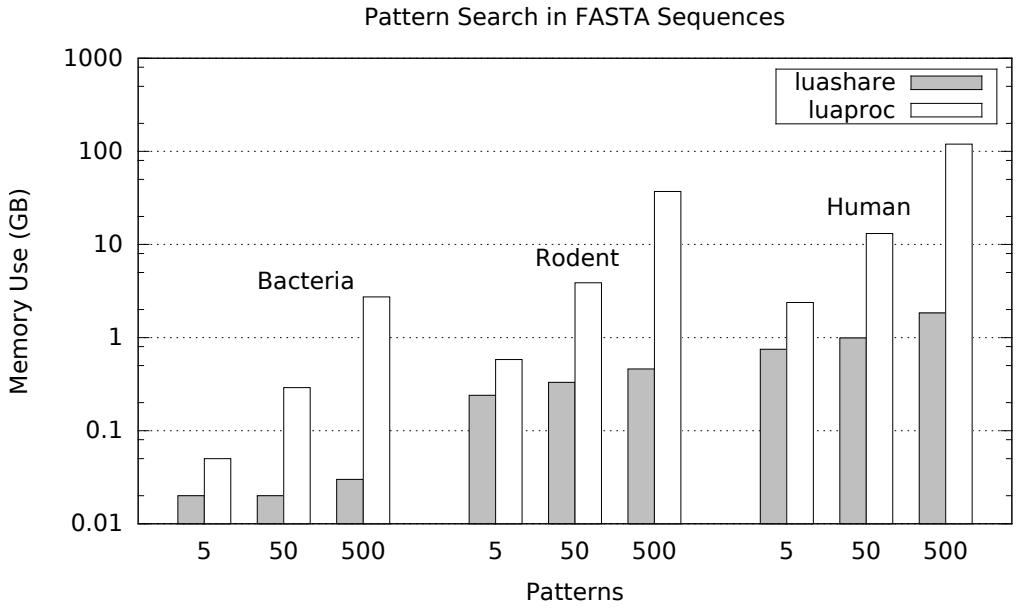

Figure 4.5: Memory use observed when running the FASTA Parser application to search for a different number of patterns in genome sequences using luashare and luaproc. Lower results are better.

As we can observe, luaproc consistently uses more memory than luashare. Increasing by 10x the number of patterns in luashare barely increases memory. Meanwhile, in luaproc it increases memory use by a factor that ranges from 5.5x to 9.5x. In addition, even when working with a small number of patterns, luashare uses less memory than luaproc. The higher memory use in luaproc

can be explained by the fact that when it copies the input sequence to send it to each worker thread. Therefore, since each worker thread searches for a single pattern, luaproc must copy the input sequence a number of times equal to the number of patterns. When searching for 500 patterns in the human (large) genome sequence, this results in around 120GB of used memory.

We can conclude that luashare has better scalability than luaproc when working with large input data sets that must be shared among multiple threads. Although we did not exceed the available memory (128GB) in the server with our benchmark, we were close. Based on what we observed, we can safely infer that if we had less available memory or a combination of larger input data sets and more search patterns, we would have run out of memory and execution times would increase significantly (due to thrashing). Therefore, more generally, we can conclude that there are cases where copying data around is simply not an option – the programmer must either share data or change the parallelization strategy (for instance to use a limited number of worker threads that search for multiple patterns).

(d) CAPTCHA Filter Benchmark

The CAPTCHA Filter is an application that applies different filters to a CAPTCHA image to make it easier to process the text contained in the image with optical character recognition (OCR). It uses a pipeline to apply filters sequentially to images. In our implementation we apply the following filters, respectively:

1. grayscale, which converts the image colors to a range of shades of gray, preparing it for the next filters;
2. binary threshold, which converts the image colors to either black or white according to the brightness of each pixel, to eliminate noise;
3. Gaussian blur, which clouds the image and makes it appear as if it is viewed through a translucent screen, to reduce detail;
4. binary threshold, same as above, applied a second time to eliminate more noise;
5. invert, which converts black to white and white to black, to change contrast.

Figure 4.6 shows an example of how an input image looks like after passing each of the filters in our application.

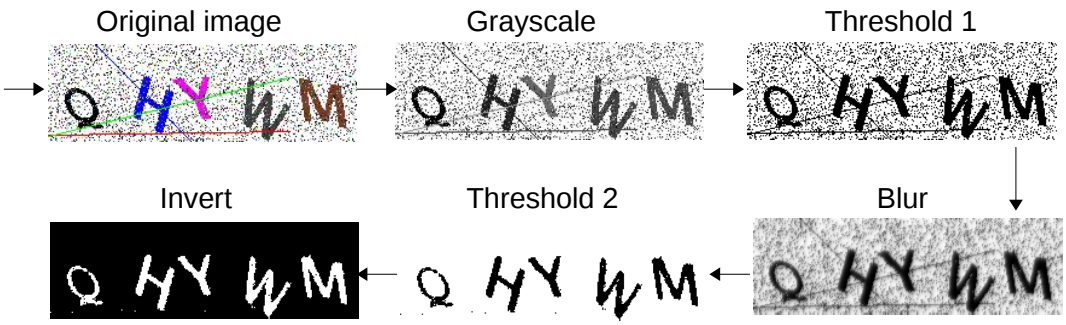

Figure 4.6: Sample results from sequentially applying each of the filters implemented in the CAPTCHA Filter application.

The parallel implementation of the application creates a worker thread per filter, as well as an additional worker thread that saves images after they have been processed, for a total of six worker threads. A master thread reads input images and send them, sequentially, to the first worker thread. Each worker applies a filter to the image it receives and sends a modified image to the next worker.

We work with a very simple image format, the portable pixmap format (PPM). A PPM file essentially contains a small ASCII header file which is followed by binary data that defines the color of each pixel in the image. We chose this format due to its simplicity, which allows images to be manipulated with simple Lua code, i.e., without external libraries.

In luashare, we used shareable objects to store images. Each image object had among its values image metadata (such as height, width and type) and a nested shareable object with the values of the red, green and blue color components of each pixel in the image. Each worker thread receives an image as read-write, modifies it and then shares the modified image as read-write with the next thread.

Figure 4.7 presents the speedups observed in luaproc and luashare. The maximum relative standard deviation we observed was 9%, although most standard deviations were below 4%.

As we can observe, luashare was actually slower than the serial version of the application. Meanwhile, luaproc showed speedups of around 1.5x, peaking at almost 2x for 32 workers processing 1,000 images. It may appear counter-intuitive that increasing the number of workers does not increase speedups significantly. However, in our experience, it is difficult to obtain good speedups with pipelines, except in very particular cases.

We attribute the slowdown in execution times cause by luashare to the overhead associated with creating shareable objects to store images and with sharing these objects among worker threads. Observe that two shareable

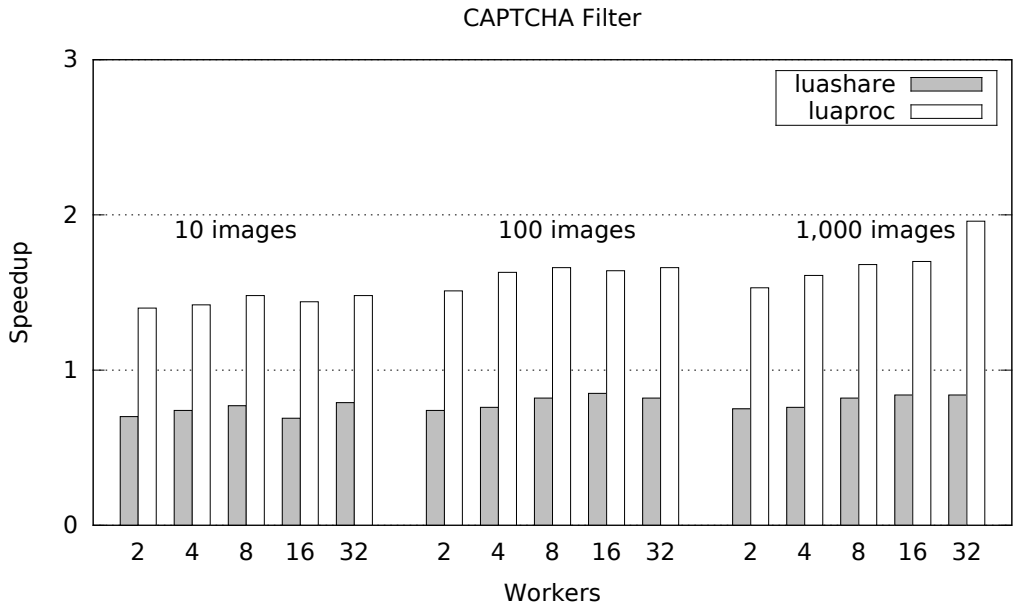

Figure 4.7: Speedups observed when running the CAPTCHA Filter application with multiple workers and processing a different number of input images using luashare and luaproc. Higher results are better; results lower than 1 mean execution times slower than the serial version of the application.

objects must be created for each image and that each image must be shared at least six times.

5
Conclusion

In this thesis we have proposed a concurrency communication model for safe record sharing in dynamic languages. The model defines shareable objects as the only means to share data among threads and message passing as the only means to share a shareable object. Each reference to a shareable object has an associated capability that defines the set of rights a thread has over the object. The model defines two mutually exclusive capabilities: read-only and read-write. When an object is shared as read-only, it becomes immutable and cannot be written anymore. When an object is shared as read-write, references to the object in the thread that is sharing the object are invalidated. A read-write shareable object can only be accessed by a single thread at a time; a read-only shareable object can be accessed simultaneously by multiple threads. In other words, a shareable object admits either a single valid read-write reference or multiple valid read-only references.

Concurrency control in our model is implemented entirely during runtime and does not rely on any type information. When a thread shares a shareable object with another thread, a new reference to the shareable object is created and a capability is dynamically assigned to that reference. Each thread has its own reference and corresponding capability to access a shareable object. Using dynamically assigned capabilities during runtime to control concurrency is the key to make the model well-suited for dynamic languages.

The main benefit of our model is that it provides structured communication, which is a necessary step to make it easier for programmers to write correct, safe concurrent applications. It prevents unpredictability by taking away from programmers the burden of synchronizing access to shared data. It keeps communication localized and explicit, by requiring programmers to explicitly declare shared data and controlling concurrent access to shared data. It also provides specific, well-defined constructs for communication, while compelling programmers to use them. Recall that the only means for threads to communicate is to use message passing and the only means to share data is to use shareable objects.

Using our model to implement a concurrent application requires reason-

ing about how information can be structured in terms of shareable objects, as they are the fundamental building block for communication among threads. Moreover, it requires considering whether, or rather which, shareable objects will need write access. Because message passing is the only means to share a shareable object, and thus for threads to communicate, synchronization in our model must be performed by matching send and receive operations. Examples of how a concurrent application can be structured using our model include: a main thread reads a large data set into a shareable object, shares it as read-only with other threads and receives the results of data processing back in shareable objects; a main thread reads a large data set into multiple shareable objects, shares each object as read-write with other threads and waits until the threads have finished writing back the results of data processing; or multiple threads organized as a pipeline where each thread processes a shareable object and then sends it as read-write to the next thread.

We implemented a prototype of the proposed concurrency model using Lua. The prototype, which we called luashare, was implemented both inside and outside the Lua interpreter. As part of our research, we are interested in studying how the choice between implementing concurrency with a library or a programming language is made [57]. In luashare, however, there was not really a choice. We had to modify the Lua interpreter as we needed to provide concurrency control for data that was only visible from within the interpreter. In particular, we had to provide concurrency control to the Lua garbage collector.

Introducing concurrency in an existing language is never as elegant as designing a language from scratch with concurrency in mind. Nevertheless, designing a language that provides proper concurrency support and makes it easier for programmers to write correct concurrent applications is not a simple task. In our experience, using a library is the preferable approach to introduce concurrency in an existing language. Consider, as an example, the luaproc concurrency library for Lua. It is implemented entirely outside the Lua interpreter and allows for multithreading in Lua.

While implementing luashare we were faced with many problems commonly associated with the combination of preemptive multithreading and shared memory. In particular we had to deal with a number of data races, a typical concurrency problem. What sets data races apart from typical problems programmers face when programming is how difficult and time consuming it is to debug them. Indeed, to make it easier for programmers to write correct concurrent applications we must get rid of data races. And to do so, we must eliminate unpredictability.

We used luashare to evaluate the proposed model from the point of view of a user. During the evaluation our perception was that, despite the difficulties to implement luashare, we managed to successfully abstract the common complexities associated with data sharing from the user. When implementing applications with luashare, the problems we had to deal with were mostly inherent to message passing (such as inadvertently sharing incorrect data or not matching send and receive operations). Moreover, we found that capabilities were intuitive to use and did not restrict parallelization patterns.

The ability to share data makes communication among threads more straightforward when using structured data. While in luashare we simply used shareable objects to hold structured data, in luaproc we had to serialize data before sending it. Besides, we observed that sometimes copying data around is simply not an option. When working with large data sets that must be accessed by multiple threads, it is imperative to be able to share data among threads. During the evaluation, we observed that allowing multiple threads to access large input data as read-only was a common pattern.

The biggest limitation we found during the evaluation of luashare was the performance cost of enforcing concurrency controls in shareable objects. As we have discussed, deciding where and how to implement such controls is a complex issue. On the one hand, having stricter control over how objects are locally manipulated by threads increases access times but allows for faster sharing. On the other hand, having stricter control over sharing, the approach that we took, makes sharing slower but access times are lower. Because the routines used to implement such controls can potentially be executed many times as objects are accessed and shared, adding or removing a single operation can have a significant performance impact on overall execution time. Unfortunately, all controls come at a price. We cannot expect to make correct concurrent programming easier if we are not ready to pay it.

Another important limitation, which is related to how shareable objects are controlled, is how nesting works in luashare. Because programmers can keep references to objects within shareable objects, we must assume they can use them to store unsupported data types. Therefore, when we share a nested object, we must check all the object's values, as well as all the inner objects values, to ensure they hold only supported data types. This delays sharing nested objects, which is a concern since using nested objects is a common technique to store structured data. During the evaluation we clearly saw a performance impact caused by sharing nested objects.

Observe that the limitations we identified are related to how we implemented luashare. They do not invalidate the proposed communication model

or suggest the model needs changes in the short-term. In fact, we think the model should be further researched and the implementation could be improved. Therefore, in the next section we provide some suggestions for future work.

5.1 Future Work

The first and perhaps more pressing matter that should be analyzed, regarding the implementation, is how shareable objects are controlled. This involves evaluating where and how controls should be implemented. One approach that should be investigated is splitting the control cost between local object manipulation and object sharing. In this approach, there would be strict controls for writing values to shareable objects. Therefore, we would be able to prevent users from storing unsupported data types. This would make writing values slower, but would make sharing faster, as we would not need to check the type of every value before sharing.

Improving how shareable objects are controlled also relates to nesting. As we explained earlier, currently users can keep references to objects within shareable objects. Because we cannot trust that objects were safely manipulated locally, we must check all of their values before sharing them. One way to remove the need to check nested objects would be to allow objects to be created within objects without allowing the user to have inner object references. In other words, it would become possible to create shareable objects that only exist within other objects. Therefore, nested objects could be shared as a whole, without the need to check every inner object. Observe that splitting control, as proposed in the previous paragraph, would also improve nesting, as when sharing an object we would be sure it only contains supported data types. In this case, since we would have to change capabilities recursively anyway, we could store a list of inner shareable objects to remove the need to iterate over all values of the (outer) object looking for them.

Another limitation of the implementation that could be looked into is the lack of support to share the userdata type. This type represents a raw block of memory and is used to store arbitrary C data in Lua variables. It is commonly employed by Lua libraries. For example, a socket in LuaSocket [52] and a pattern in LPeg [47] are both userdata. Therefore, it would be useful to evaluate whether it is possible to share userdata among threads by using a scheme similar to the one we used in luashare implement shareable objects.

Another potential problem with the implementation which we observed and briefly talked about is blocking during I/O operations. Recall that we use a barrier to synchronize threads before executing a garbage collection cycle. Therefore, if a thread blocks executing an I/O operation, all other threads

will have to wait at the barrier and the entire application will ultimately be delayed. It would be opportune to look into the possibility of implementing asynchronous I/O operations in luashare, to prevent blocking threads.

Last, but not least, we consider the proposed communication model high-level enough that it abstracts data sharing complexities from users, but yet low-level enough that it lends itself to being used as a building block for higher-level concurrency abstractions. Although during the evaluation we only worked with the basic constructs provided by luashare, further research could go into evaluating if and how the model can be used to build other concurrency constructs. It would also be interesting to see how well it can be implemented in dynamic languages other than Lua.

Bibliography

[1] S. V. Adve and H.-J. Boehm. **Memory models: A case for rethinking parallel languages and hardware**. *Communications of the ACM*, 53(8):90–101, August 2010. 1.3

[2] G. R. Andrews and F. B. Schneider. **Concepts and notations for concurrent programming**. *ACM Computing Surveys*, 15:3–43, March 1983. 1.3

[3] H. C. Baker, Jr. and C. Hewitt. **The Incremental Garbage Collection of Processes**. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM. 2

[4] T. Bergan, N. Hunt, L. Ceze and S. D. Gribble. **Deterministic process groups in dOS**. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, *OSDI'10*, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. 1.2

[5] C. Bienia, S. Kumar, J. P. Singh and K. Li. **The PARSEC benchmark suite: Characterization and architectural implications**. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008. 4.2

[6] A. Birka and M. D. Ernst. **A practical type system and language for reference immutability**. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, *OOPSLA '04*, pages 35–49, New York, NY, USA, 2004. ACM. 2.1

[7] G. E. Blelloch, J. T. Fineman, P. B. Gibbons and J. Shun. **Internally deterministic parallel algorithms can be fast**. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, *PPoPP '12*, pages 181–192, New York, NY, USA, 2012. ACM. 1.2

[8] J. Bloch. **Effective Java (2nd edition) (the Java series)**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008. 1.5, 2.1

[9] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve and M. Snir. **Parallel programming must be deterministic by default**. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, *HotPar'09*, Berkeley, CA, USA, 2009. USENIX Association. 1.5, 2.1

[10] R. L. B. Jr., S. Heumann, N. Honarmand, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, M. Vakilian, S. V. Adve, V. S. Adve, D. Dig and M. Snir. **A language for deterministic-by-default parallel programming**. In *Proceedings of the 15th Workshop on Compilers for Parallel Computing (CPC 2010)*, 2010. 1.2

[11] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc and T. Shpeisman. **Safe nondeterminism in a deterministic-by-default parallel language**. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL '11*, pages 535–548, New York, NY, USA, 2011. ACM. 1.2

[12] H.-J. Boehm and S. V. Adve. **You don't know jack about shared variables or memory models**. *Communications of the ACM*, 55(2):48–54, February 2012. 1.2, 1.3

[13] H.-J. Boehm. **Position paper: Nondeterminism is unavoidable, but data races are pure evil**. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, *RACES '12*, pages 9–14, New York, NY, USA, 2012. ACM. 1.2

[14] J. Boyland, J. Noble and W. Retert. **Capabilities for sharing: A generalisation of uniqueness and read-only**. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, *ECOOP '01*, pages 2–27, London, UK, UK, 2001. Springer-Verlag. 2.1

[15] C. Boyapati, R. Lee and M. Rinard. **Ownership types for safe programming: preventing data races and deadlocks**. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, *OOPSLA '02*, pages 211–230, New York, NY, USA, 2002. ACM. 2.1

[16] L. A. Smith, J. M. Bull and J. Obdrŷálek. **A Parallel Java Grande Benchmark Suite**. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001. 4.2

[17] D. G. Clarke, J. M. Potter and J. Noble. **Ownership types for flexible alias protection**. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, *OOPSLA '98*, pages 48–64, New York, NY, USA, 1998. ACM. 2.1

[18] J. B. Dennis and E. C. Van Horn. **Programming semantics for multiprogrammed computations**. *Communications of the ACM*, 9(3):143–155, March 1966. 3.1.2

[19] **The parallel programming landscape: Multicore has gone mainstream – but are developers ready?**, 2012. UBM TechWeb survey sponsored by Intel and conducted with Dr. Dobb's readers. 1, 1.4

[20] P. A. Emrath and D. A. Padua. **Automatic detection of nondeterminacy in parallel programs**. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, *PADD '88*, pages 89–99, New York, NY, USA, 1988. ACM. 1.2

[21] Ericsson AB. **Erlang standard library (stdlib) user's guide**, December 2014. Erlang STDLIB User's Guide 2.3. 1.3

[22] C. Flanagan and S. Qadeer. **A type and effect system for atomicity**. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, *PLDI '03*, pages 338–349, New York, NY, USA, 2003. ACM. 1.5, 2.1

[23] P. Fonseca, C. Li and R. Rodrigues. **Finding complex concurrency bugs in large multi-threaded applications.** In *Proceedings of the Sixth European Conference on Computer Systems*, *EuroSys '11*, pages 215–228. ACM, 2011. 1

[24] N. Ford. **Functional thinking: Immutability – make Java code more functional by changing less**, July 2011. 1.5, 2.1

[25] J. S. Foster, M. Fähndrich and A. Aiken. **A theory of type qualifiers**. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, *PLDI '99*, pages 192–203, New York, NY, USA, 1999. ACM. 2

[26] D. P. Friedman and D. S. Wise. **The impact of applicative programming on multiprocessing**. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 263–272. IEEE, august 1976. 2

[27] D. Gelernter and N. Carriero. **Coordination languages and their significance**. *Communications of the ACM*, 35:97–107, February 1992. 2.2

[28] P. Godefroid and N. Nagappan. **Concurrency at Microsoft: – an exploratory survey**. Technical Report MSR-TR-2008-75, Microsoft Research, May 2008. 1

[29] B. Goetz. **Java theory and practice: To mutate or not to mutate? – Immutable objects can greatly simplify your life**, February 2003. 1.5, 2.1

[30] D. Grossman. **The transactional memory / garbage collection analogy**. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, *OOPSLA '07*, pages 695–706, New York, NY, USA, 2007. ACM. 1.4

[31] C. Haack, E. Poll, J. Schäfer and A. Schubert. **Immutable objects in Java**. Dept. of Computer Science ICIS-R06010, Radboud University Nijmegen, 2006. 2.1

[32] C. Haack, E. Poll, J. Schäfer and A. Schubert. **Immutable objects for a Java-like language**. In R. D. Nicola, editor, *ESOP'07*, volume 4421 of *LNCS*, pages 347–362. Springer, 2007. 2.1

[33] C. Haack and E. Poll. **Type-based object immutability with flexible initialization**. In *ECOOP 2009*, volume 5653 of *LNCS*, pages 520–545. Springer, 2009. 1.5, 2.1

[34] R. H. Halstead, Jr. **MULTILISP: A language for concurrent symbolic computation**. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985. 1.2

[35] P. B. Hansen. **An outline of a course on operating system principles**. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 29–36, London, 1972. Academic Press. 1

[36] P. B. Hansen. **Operating system principles**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973. 1

[37] P. B. Hansen. **Java's insecure parallelism**. *ACM SIGPLAN Notices*, 34:38–45, April 1999. 1

[38] S. Herhut, R. L. Hudson, T. Shpeisman and J. Sreeram. **Parallel programming for the web**. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX. 2.2

[39] S. T. Heumann, V. S. Adve and S. Wang. **The tasks with effects model for safe concurrency**. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, *PPoPP '13*, pages 239–250, New York, NY, USA, 2013. ACM. 1.5, 2.1

[40] C. A. R. Hoare. **Record handling**. *A series of lectures to be delivered at The NATO Summer School*, September 1966. 1

[41] R. Ierusalimschy, L. H. de Figueiredo and W. C. Filho. **Lua – an Extensible Extension Language**. *Software – Practice and Experience*, 26(6):635–652, 1996. 3.2

[42] J. Jackson. **NASDAQ's Facebook glitch came from 'race conditions'**. *Computerworld – Financial IT*, May 2012. `http://www.computerworld.com/s/article/9227350/Nasdaq_s_Facebook_glitch_came_from_race_conditions_`. 1

[43] E. Johansson, K. Sagonas and J. Wilhelmsson. **Heap architectures for concurrent languages using message passing**. In *Proceedings of the 3rd International Symposium on Memory Management*, *ISMM '02*, pages 88–99, New York, NY, USA, 2002. ACM. 1.5, 2

[44] R. LaPedis. **Lessons learned from the 2003 northeastern blackout**. Technical report, Hewlett-Packard (HP), May 2004. `http://h20223.www2.hp.com/NonStopComputing/downloads/Lessons_NEBlackoutArticle.pdf`. 1

[45] E. A. Lee. **Disciplined message passing**. Technical report, EECS Dept. University of California Berkeley, January 2009. 1, 1.3, 2

[46] N. G. Leveson and C. S. Turner. **An investigation of the Therac-25 accidents**. *Computer*, 26(7):18–41, July 1993. 1

[47] R. Ierusalimschy. **LPeg: Parsing Expression Grammars for Lua**, 2014. `http://www.inf.puc-rio.br/~roberto/lpeg`. 5.1

[48] S. Lu, S. Park, E. Seo and Y. Zhou. **Learning from mistakes: A comprehensive study on real world concurrency bug characteristics**. In *Proceedings of the 13th International Conference on Architectural Support*

for Programming Languages and Operating Systems, *ASPLOS XIII*, pages 329–339, 2008. 1, 1.2, 1.3

[49] L. Lu and M. L. Scott. **Toward a formal semantic framework for deterministic parallel programming.** In *Proceedings of the 25th International Symposium on DIStributed Computing (DISC)*, volume 6950 of *Lecture Notes in Computer Science*, pages 460–474. Springer, 2011. 1.2

[50] Y. Lu, J. Potter, C. Zhang and J. Xue. **A type and effect system for determinism in multithreaded programs**. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, *ESOP'12*, pages 518–538, Berlin, Heidelberg, 2012. Springer-Verlag. 1.5

[51] L. Lu, W. Ji and M. L. Scott. **Dynamic enforcement of determinism in a parallel scripting language**. In *Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation*, *PLDI '14*. ACM, 2014. 2.2

[52] D. Nehab. **Luasocket: Network support for the Lua language**, 2007. `http://w3.impa.br/~diego/software/luasocket`. 5.1

[53] A. Kauppi. **Lualanes – multithreading in Lua**, 2009. `http://kotisivu.dnainternet.net/askok/bin/lanes/`. 2.2

[54] A. Gladysh. **luabins – Lua Binary Serialization Library**, 2011. `https://github.com/agladysh/luabins`. 4.2

[55] Ericsson AB. **Mnesia user's guide**, December 2014. Mnesia's User Guide 4.12.4. 1.3

[56] F. Nielson and H. R. Nielson. **Type and effect systems**. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer Berlin Heidelberg, 1999. 2

[57] M. Odersky. **Tackling concurreny – language or library?** `http://lampwww.epfl.ch/~odersky/talks/intel06.pdf`, November 2006. Talk presented at Intel Berkeley Research Center. 1, 1.2, 2, 5

[58] M. Olszewski, J. Ansel and S. Amarasinghe. **Kendo: Efficient deterministic multithreading in software**. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS XIV*, pages 97–108, New York, NY, USA, 2009. ACM. 1.2

[59] J. Ousterhout. **Why threads are a bad idea (for most purposes)**. *Presentation given at the 1996 USENIX Annual Technical Conference*, January 1996. 1, 1.3, 2

[60] I. Pechtchanski and V. Sarkar. **Immutability specification and its applications**. *Concurrency and Computation: Practice and Experience*, 17(5–6):639–662, April/May 2005. Special Issue: Java Grande/ISCOPE 2002. 2.1

[61] S. Peyton Jones, A. Gordon and S. Finne. **Concurrent Haskell**. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL '96*, pages 295–308, New York, NY, USA, 1996. ACM. 2.1

[62] B. C. Pierce. **Types and programming languages**. MIT Press, Cambridge, MA, USA, 2002. 2

[63] M. Odersky, L. Spoon and B. Venners. **Actors and concurrency**. In *Programming in Scala*, chapter 30, page 619. Artima Press, 1st edition, 2008. 1.3

[64] B. O'Sullivan, D. Stewart and J. Goerzen. *Real World Haskell*, pages XXIV–XXVII. O'Reilly Media, 1st edition, December 2008. 1.4

[65] D. C. Schmidt and S. Vinoski. **Comparing alternative programming techniques for multithreaded CORBA servers: Thread pool.** *C++ Report*, April 1996. 3.2.1

[66] A. Skyrme, N. Rodriguez and R. Ierusalimschy. **Exploring Lua for concurrent programming**. *Journal of Universal Computer Science*, 14(21):3556–3572, dec 2008. 2.2

[67] A. Skyrme, N. Rodriguez and R. Ierusalimschy. **A survey of support for structured communication in concurrency control models**. *Journal of Parallel and Distributed Computing*, 74(4):2266–2285, 2014. 1.4, 1.5

[68] A. Skyrme, N. Rodriguez and R. Ierusalimschy. **Scripting multiple CPUs with safe data sharing**. *Software, IEEE*, 31(5):44–51, September 2014. 2

[69] H. Sutter and J. Larus. **Software and the concurrency revolution**. *ACM Queue*, 3:54–62, September 2005. 1, 1.3, 2

[70] H. Sutter. **Use thread pools correctly: Keep tasks short and nonblocking**. *Dr. Dobb's Journal*, April 2009. Effective Concurrency column. 3.2.1

[71] W. Gropp, E. L. Lusk and A. Skjellum. **Using MPI: Portable parallel programming with the Message Passing Interface**. MIT Press, 2nd edition, 1999. 1.3

[72] R. von Behren, J. Condit and E. Brewer. **Why events are a bad idea (for high-concurrency servers)**. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems – Volume 9*, pages 19–24, Berkeley, CA, USA, 2003. USENIX Association. 1.3

[73] S. West, S. Nanz and B. Meyer. **Demonic testing of concurrent programs**. In *Proceedings of the 14th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, *ICFEM'12*, pages 478–493, Berlin, Heidelberg, 2012. Springer-Verlag. 1.2

[74] S. E. Zenith. **Process interaction models**. PhD thesis, *Ecole Nationale Supérieure des Mines de Paris*, Centre de Recherche en Informatique – 35 rue Saint-Honore 77305 – Fontainebleau – France, 1992. 1.3

[75] Z. B. Rui Zhang and W. N. S. III. **Composability for application-specific transactional optimizations**. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, *TRANSACT 2010*, April 2010. 1.4

[76] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun and M. D. Ernst. **Object and reference immutability using Java generics**. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, *ESEC-FSE '07*, pages 75–84, New York, NY, USA, 2007. ACM. 2.1