AN EFFICIENT IMPLEMENTATION OF GUARD-BASED SYNCHRONIZATION FOR AN OBJECT-ORIENTED PROGRAMMING LANGUAGE

AN EFFICIENT IMPLEMENTATION OF GUARD-BASED SYNCHRONIZATION FOR AN OBJECT-ORIENTED PROGRAMMING LANGUAGE

By

SHUCAI YAO, M.Sc., B.Sc.

A Thesis

Submitted to the Department of Computing and Software

and the School of Graduate Studies

of McMaster University

in Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

Doctor of Philosophy (2020) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	An Efficient Implementation of Guard-Based Synchronization for an Object-Oriented Programming Language
AUTHOR:	Shucai Yao M.Sc., (Computer Science) University of Science and Technology Beijing B.Sc., (Computer Science) University of Science and Technology Beijing
SUPERVISOR:	Dr. Emil Sekerinski, Dr. William M. Farmer
NUMBER OF PAGES:	xvii,167

To my beloved family

Abstract

Object-oriented programming has had a significant impact on software development because it provides programmers with a clear structure of a large system. It encapsulates data and operations into objects, groups objects into classes and dynamically binds operations to program code. With the emergence of multi-core processors, application developers have to explore concurrent programming to take full advantage of multi-core technology. However, when it comes to concurrent programming, object-oriented programming remains elusive as a useful programming tool.

Most object-oriented programming languages do have some extensions for concurrency, but concurrency is implemented independently of objects: for example, concurrency in Java is managed separately with the *Thread* object. We employ a programming model called *Lime* that combines action systems tightly with object-oriented programming and implements concurrency by extending classes with actions and guarded methods. This provides programmers with a unified and straightforward design view for a concurrent object-oriented program.

In this work, using coroutines with guarded methods and actions is proposed as a means of implementing the concurrency extension for objects. Mapping objects to coroutines can result in stack overflow as the number of objects increases. A dynamically segmented stack mechanism, which does not introduce runtime overhead, is implemented to support large-scale concurrency. Since Lime allows guarded methods and actions to "get stuck," a new user-level cooperative scheduler, and a fast coroutine context switch mechanism are implemented to improve the performance.

Compared with the traditional segmented stack mechanisms, the new dynamically segmented stack mechanism gets equal performance for more common scenarios. Besides, it outperforms the contemporary stack mechanisms for deep recursion scenarios. Above all, Lime does not only provide the programmers with a unified and straightforward object-oriented programming model for concurrency, but also accomplishes a better performance than concurrent programming languages such as Erlang and Go, in fine-grained, highly concurrent benchmarks.

Acknowledgements

First and foremost, I would like to express my gratitude to my academic supervisor Dr. Emil Sekerinski for his unwavering support, invaluable guidance, and continuous encouragement throughout the development and advancement of my research studies.

Secondly, I would like to express great appreciation to my co-supervisor, Dr. William M. Farmer, for his excellent comments and suggestions on my thesis and a careful review of the thesis.

My special thanks go to the other members of my supervisory committee: Dr. Wolfram Kahl and Dr. Frantisek (Franya) Franek, for excellent comments and suggestions on supervisory committee meetings and a careful review of the thesis. I also appreciate the help and moral support from all of the other individuals whom I have interacted with throughout my graduate studies.

Furthermore, I want to extend my sincere thanks to the China Scholarship Council, Global Water Futures and McMaster University graduate scholarship, which have provided financial support during my doctoral studies.

Last, but certainly not least, I would like to thank my dear family and my fellow graduate students, Dr. Ronald Eden Burton, Dr. Bojan Nokovic and Dr. Tian Zhang for their constructive feedback in our group meetings.

Contents

List of Tables

List of Figures

Listings

Declaration of Academic Achievement

The following publications cover contributions described in this thesis:

1. Moore-Oliva, J., Sekerinski E. and Yao S. (2014) "A Comparison of Scalable Multi-Threaded Stack Mechanisms." In McMaster University, Department of Computing and Software, Report CAS-14-07-ES, 10 pages.

   My contribution includes identifying the performance gains of the Guard-Page stack mechanism, adding the "callee clean up" mechanism, introducing the stack reuse mechanism and extending the benchmarks accordingly. This work will be discussed in Chapter 5. Moore-Oliva built the C-like compiler, created the benchmarks and compared the performance among different stack mechanisms.

2. Sekerinski E. and Yao S. (2018) "Refining Santa: An Exercise in Efficient Synchronization." Extended Abstract. In REFINE 2018: Refinement Workshop Oxford, UK, July 18, 2018, 16 pages.

   My contributions are the implementation and the experimental part. This work will be discussed in Section 6.4.4. Dr. Sekerinski Emil worked on the theory part.

3. Fadhel M., Sekerinski E., and Yao S. (2018) "A comparison of time series databases for storing water quality data." In International Conference on Interactive Mobile Communication, Technologies and Learning, 11–12 October 2018 at McMaster University, Hamilton, Ontario, Canada, 10 page.

An Efficient Implementation of Guard-Based Synchronization for an Object-Oriented Programming Language

Shucai Yao

July 21, 2020

Chapter 1

Introduction

In this chapter we glance back at the history of processors, from uniprocessors to multicore processors, to see how we got where we are now. Next, we take a close look at the development of programming languages for large-scale applications. Then we examine the discrepancy between models and implementations for concurrency. Finally, this chapter presents the structure of the thesis.

1.1 Processors

1.1.1 Uniprocessors

Early uniprocessor computers allow only one program to be executed at a time. The process model is a key component of an operating system for a uniprocessor system. An operating system consists of several processes: one operating system process and several user processes. All these processes can execute concurrently by a time-sharing mechanism. Conceptually, each process has its virtual CPU with the real CPU switching among processes.

Processes

A *process* is a program in execution, which includes the program code, registers, a stack, a data section and a heap. The stack keeps local data for the program while the heap contains the dynamically allocated data during the running of the program. The data section contains global variables. Each process control block (PCB) represents one process. A PCB includes all information needed by the scheduler. That is, the PCB is the "manifestation of a process in an operating system" (Deitel, 1990). One PCB includes:

- Process ID
- Stack Pointer
- Program Counter
- Process State
- Process's Register Values
- Memory Management Information
- Other Data: Scheduling Information, Process Privileges, Inter-Process Communication Information, I/O Information, Accounting Information

Operating systems that support the concept of process must meet two objectives: to maximize the CPU utilization and to switch the CPU among processes fast. To achieve these two objectives, the scheduler schedules a process to execute the program on the CPU. When an available process is scheduled, the scheduler performs a context save of the current process and a context restore of the selected process. The context of the process is represented in the PCB which contains all the scheduling information of the process in the system.

The Limitations of Processes

First of all, the overhead of context switches can be significant. During a context switch, the system cannot do any useful work. Therefore, the cost of the context switch is considered as pure overhead. The overhead includes that the processor registers (e.g., 16 registers on the X86 architecture) need to be saved and restored, the translation lookaside buffer entries need to be reloaded, and the processor pipeline must be flushed. Second, the overhead for creating, managing, and communicating among processes can also be significant. Third, operating systems need to guarantee some degree of independence and security among concurrently executing processes. Each process has its own address space, and communication between processes relies on inter-process communication facilities such as pipes, which requires time-consuming system calls.

Threads

We have assumed so far that the process model involves only one thread, i.e., a light-weight process. The thread model enables a process to contain multiple threads. In this case, a *thread* becomes a basic unit of CPU utilization. A thread includes program code, registers, a stack and a shared heap. In the operating system, one thread is represented by a thread control block (TCB). The TCB is "the manifestation of a thread in an operating system" (Deitel, 1990). Compared with a PCB, a TCB does

not contain memory management information. That is, a process could have multiple threads, and all these threads share the address space and system resources, such as code section, data section and opened files, of the process. One TCB includes:

- Thread ID
- Stack Pointer
- Program Counter
- Thread State
- Thread's Register Values
- Pointer to PCB

Since creation and management of processes can be expensive, opportunities for sharing local resources are limited, and the communication mechanisms among processes are relatively heavy, the concept of the thread was introduced to make further tradeoffs between autonomy and overhead. The main compromises are:

- *Resource sharing*. By default, all threads of one process share the address space and the resources of the process. Compared with processes, context switching between threads, and creating and destroying threads are more economical. "In Solaris, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower" (Silberschatz et al., 2014, p. 166).
- *Responsiveness*. The multithreaded model provides the opportunity for a program to continue running when some of the threads are blocked or are working on time-consuming tasks. Thus, the system increases responsiveness to the users. For instance, web browsers allow users to watch videos in one thread and download files in another thread at the same time.
- *Communication*. There are more communication options available in the multithreaded model. In addition to the communication mechanisms of the processes, threads can also use other cheaper strategies relying on memory accessing and employing memory-based synchronization facilities such as locks.

There are two kinds of threads in the operating system: kernel-level threads and user-level threads. The kernel-level threads are implemented in the operating system kernel and are scheduled by the operating system's scheduler. The user-level threads are implemented on top of the kernel and managed entirely by the user-level library. That is, user-level threads are "cheaper" than kernel-level threads because user threads have less state to maintain. Because the user-level threads rely on the support of the kernel-level threads, there are three common relationships between kernel-level threads and user-level threads: many-to-one, one-to-one and many-to-many.

The many-to-one model maps many user-level threads to one kernel thread. The drawback to this model is that only one thread can access the kernel at a time. That is, the user-level threads cannot run in parallel. The one-to-one model maps each user-level thread to one kernel thread. The overhead of creating threads can be a significant burden of the system when the number of threads exceeds the threshold value (sometimes it equals to the number of cores). The many-to-many model, also called hybrid model, maps many user-level threads to a fixed number of kernel threads. The number of the kernel threads usually equals to the number of the cores.

The Limitations of Threads

The trade-off made in supporting threads supports a wide range of applications, but cannot always perfectly meet all the needs of a given condition. The overhead of thread creation is still significantly higher than the overhead for lightweight threads, such as fibers (Shankar, 2003), green threads (SunSoft, 1997). Lightweight threads use cooperative multitasking while standard threads use preemptive multitasking. Because the user thread schedulers on Linux and Windows are preemptive, the schedulers still need to save and restore all the registers belonging to the thread during the context switch. When the overhead of thread creation and management becomes a performance concern, we need to make additional trade-offs in autonomy for the sake of the performance: lightweight threads are needed.

Coroutines

Coroutines (Knuth, 1973), a kind of lightweight thread, are computer program components that can be suspended and resumed at specific locations. Compared with threads, coroutines provide a cheaper and more controllable approach to avoid the time-consuming blocking operations, such as file I/O, by suspending and resuming coroutines. Unlike subroutines, coroutines call other coroutines as peers and transfer the control in a symmetric way between coroutines. In this case, coroutines are implemented using continuations, which require allocation of a separate stack for each coroutine. Preallocating or caching stacks can speed up the creation of coroutines.

Coroutines are derived from an assembly language method, but are used to implement the asynchronous mechanisms in more and more programming languages, such as the asynchronous model implemented by C# (Microsoft, 2019) and ECMAScript (Mozilla, 2019).

As for light-weight threads, the performance improvements of coroutines come from three aspects. First of all, coroutines are cooperatively scheduled in the userspace, rather than relying on the kernel to manage their time sharing. So coroutines are more efficient than threads. Second, the switch between coroutines only occurs at distinct points, when an explicit call is made to the runtime scheduler. Therefore, the scheduler for coroutines just needs to handle a small number of registers during the context switches. Third, the distinct points are the places where the coroutine has to wait

until the computing resource is ready, which can reduce the unnecessary blocking time of the coroutines.

The context switch between coroutines happens in the userspace. First, a runtime system is needed for this. Second, the disadvantage of cooperative scheduling is that if a running coroutine does not yield the control, all the other coroutines will never be executed. Explicit transfers need to be inserted by the programmers or by the compiler if some degree of fairness is desirable.

1.1.2 Multicore Processors

The history of parallel processors can be traced back to the Solomon computer of the mid-1960s (Slotnick et al., 1962). Because of the promise of parallelism, parallel computing has caught both researchers' and industry's attention for the last four decades. However, uniprocessor computing prevailed over parallel processors computing before 2005 (Asanovic et al., 2009).

There are many reasons behind this, but the leading one is that the programmers could wait for the uniprocessor's hardware designers to speed their programs up by increasing the frequency. Besides, some sequential programming models have proven useful for the development and maintenance of large software. In contrast, switching from sequential to modestly parallel computing raises the bar for programming and then limits the customer base of parallel processors (Blake et al., 2009).

Due to the *memory wall* (the growing gap of frequency between the CPU and memory), *instruction-level parallelism wall* (the increasing complexity of implementing instruction-level parallelism features and the decreasing of frequency of CPU because of misprediction penalty) and *power wall* (the exponentially increasing temperature and power consumption as the frequency increases), Intel announced in 2005 that "its high-performance microprocessors would henceforth rely on multiple processors or cores" (Geer, 2005), which meant that the computing industry significantly changed course in 2005. Leading manufacturers of processors, including IBM, Sun Microsystems and Intel, started to switch from uniprocessors to multicore processors.

The switch from uniprocessors to multicore processors is a milestone in the history of computing. First of all, performance becomes the programmer's burden. To increase the performance, the programmers must make their programs more parallel. Second, this shift towards increasing parallelism is not based on breakthroughs in programming models for parallelism; instead, this change occurred because there is no other way to continue the expected performance growth from Moore's law. Compared with a uniprocessor system, a multicore system only provides the potential to continue to improve the performance by replacing a computationally efficient, energy-inefficient core with several slower, energy-efficient cores. To realize this full potential, the development of new programming models for parallel computing has become a significant area of interest.

Nowadays general-purpose multicore processors are widely accepted from large-scale cluster to the desktop, from the signal processing system to embedded devices, as the need for more performance and power efficiency has grown. The increased use of multicore processors in various work environments has led to heightened concerns for software programmers with regards to obtaining sustainable performance improvement.

Compared with uniprocessor systems, the raw performance increase of multicore systems comes from increasing the number of cores rather than the frequency, thus avoiding the power wall. However, to give full play to multicore's superiority is a significant challenge because effective parallel programming lags behind our ability to build parallel processors.

1.1.3 Microprocessor Performance and Trends

Trends identified in Figure 1.1 (Rupp, 2018) were: the number of transistors is continually increasing, the frequency is reaching the top, and the single-thread performance is growing slightly. We note two significant changes: First, since 2005, the clock rate increase has been growing at less than 1% per year. Second, the number of cores has doubled with each processor generation since 2005.

Because of the power wall, there are no significant changes in frequency and power. The current trend in hardware is increasing the number of cores based on the technology of shrinking transistors, which can shrink both capacitance and the supply voltage to some degree so that the number of cores can be doubled with each silicon generation. Also, single-thread performance has kept increasing slightly by allowing CPUs to adjust the frequency dynamically based on the actual need (Charles et al., 2009).

According to predictions from the International Technology Roadmap for Semiconductors (ITRS), higher performance is still a priority (Figure 1.2). To achieve higher performance of processors, the manufacturers have started to increase the number of cores, rather than increasing the CPU frequency. Therefore, the ITRS Roadmap has predicted that there will be processors with upwards of 100 cores by 2022 (ITRS, 2007). In 2019, AMD (2019) released a processor with 128 cores and Intel (2019) released a processor with 112 cores.

1.2 Object-Oriented Programming Languages

In this section, we focus on the object-oriented programming languages and discuss why the object-oriented programming model is useful in sequential programming but problematic in concurrent programming.

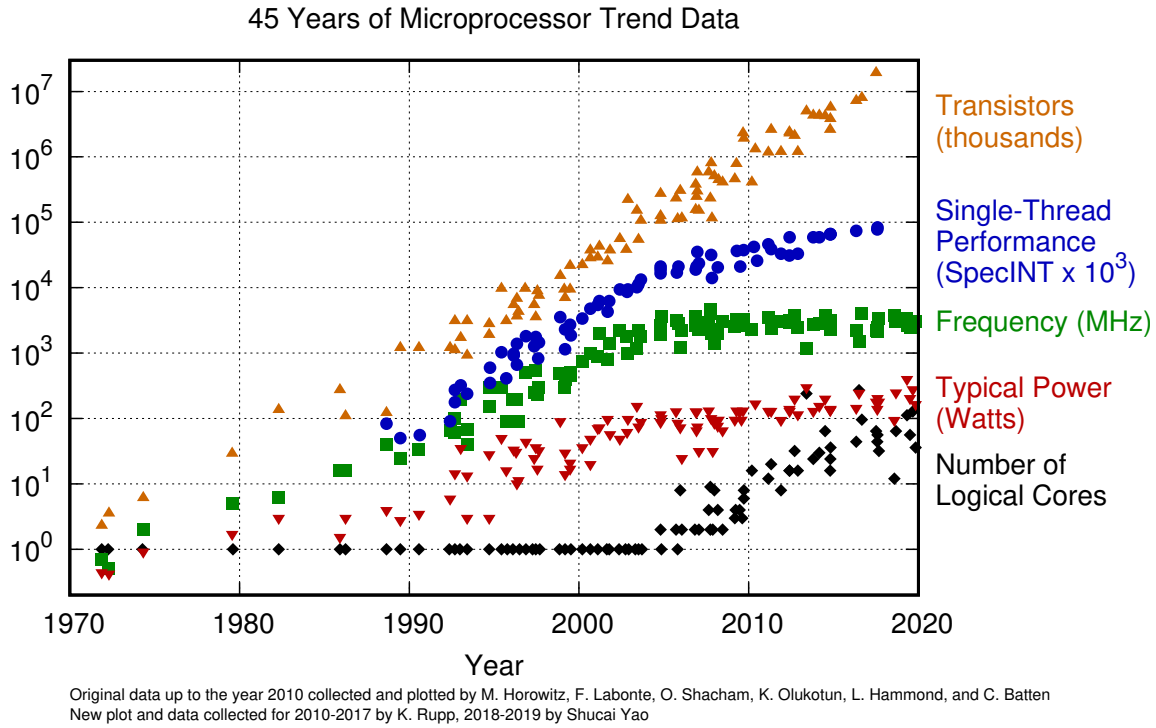

Figure 1.1: 45 Years of Microprocessor Trend Data (Rupp, 2018)

1.2.1 Sequential Programming

Object-Oriented Programming Languages (OOPLs) are becoming increasingly popular because they combine numerous techniques that have proven useful for the development, maintenance, and reusability of software. OOPLs have had a significant impact on software development because they provide programmers with a clear structure for a large system. From the programmer's point of view, OOPLs reduce the complexity of the construction of large computer software.

Consider the four principles of OOPLs:

- *Encapsulation* combines data and operations on that data, which provides programmers with an external interface that contains various services, without worrying about how those services are implemented.

- *Abstraction* hides all but the relevant data about an object, which allows programmers to concentrate on the essential characteristics and ignore what is not relevant.

- *Polymorphism* allows operations to behave differently on objects depending on

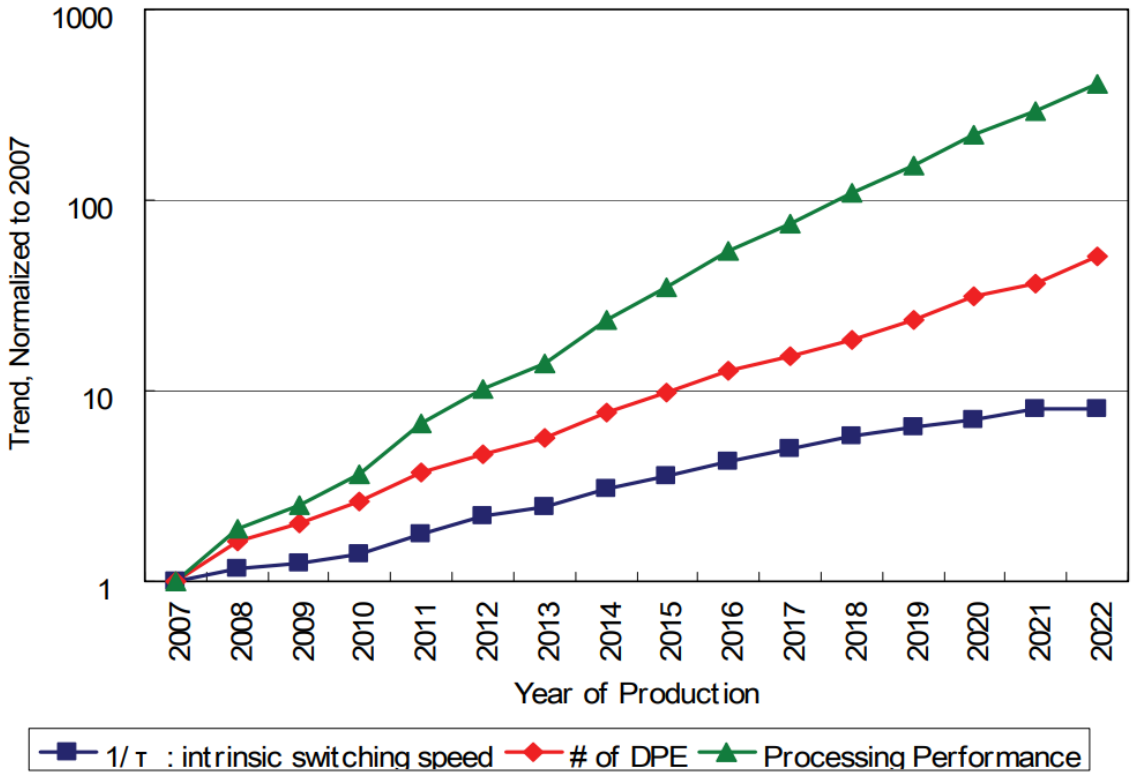

Figure 1.2: SOC Consumer Stationary Performance Trends (ITRS, 2007)

the data type or class of the objects, which enables programmers to ask for a service without worrying about how different objects might provide it.

- *Inheritance* empowers new objects to obtain the properties of existing objects which allows programmers to reuse an interface or an implementation, providing only the pieces that are different from what was provided before.

In addition to the concepts mentioned above, OOPLs allow certain design principles, the open-closed principle: "[Classes] should be open for extension, but closed for modification" (Meyer, 1997) and the Liskov substitution principle: "If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T" (Liskov, 1988). These principles decrease the complexity in the development, maintenance, and extension of the computer applications.

1.2.2 Concurrent Programming

The more widespread and demanding uses of multicore processors in various areas have increased the number of applications involving concurrent programming. The

concurrent OOP model has become a hot area of research recently (West et al., 2015; Heußner et al., 2015; Faes and Gross, 2018).

There are two reasons for this. First, the demand for concurrency in OOP models stems from the real world. In real-world systems, many things are naturally parallel, and hardware is typically parallel. Concurrency in OOP models can help programmers to partition a complex system into easy-to-understand concurrent software components. A good concurrent OOP model could provide the programmer with an efficient, scalable, correct and concurrent solution for the complex system.

Second, with the domination of multicore processors, application developers now have to explore concurrent programming to take full advantage of multicore technology. Parallel computing can substantially speed up the computational work if multicore processors are available. Even within a single processor, multitasking can significantly make the program run faster by preventing unnecessary blocking; for example, one task blocks another task while waiting for I/O.

However, when it comes to concurrent programming, OOP remains elusive as a useful programming tool because OOP handles the objects and the concurrency separately. Most OOPLs do have some extensions for concurrency, but concurrency is implemented independently of objects: for example, concurrency in Java is managed separately with the *Thread* object. While syntactically threads are objects, they introduce a new control structure that has to be supported by the underlying runtime system. The programmers have to handle two different design views for concurrent object-oriented programs: one is a static view of classes with inheritance, the other is a dynamic structure of communicating threads.

It has been argued that an object should be regarded as a natural "unit" of concurrency (Ishikawa and Tokoro, 1984). A concurrent OOP model should offer programmers a unified and straightforward concurrent object-oriented programming model and make concurrency transparent to the programmer. It permits concurrency to be considered as an implementation issue in the same way as the choice of an algorithm. In this case, objects can be used to manage resources, very much like processes can, so no expressiveness is lost. A programming model called *Lime* (Sekerinski, 2003) is employed in this work: it provides the programmers with a unified and straightforward design view for a concurrent object-oriented program, which decreases the difficulty of developing, maintaining, and extending the concurrent object-oriented program.

1.3 Problems This Thesis Addresses

In this section, we first discuss the problems this thesis addresses in general, and we will elaborate their technical aspects in Section 4.2.

Lime, developed by Dr. Emil Sekerinski's research group, is a object-oriented action-based programming model. Lime has a notion of correctness of programs that is based on the weakest precondition calculus and has a notion of program development based on the refinement calculus (Sekerinski, 1996; Büchi and Sekerinski, 2000;

Sekerinski, 2002). The unified and simple design view in Lime stems from the combination of the OOPL model and action systems. However, this combination requires a simple and efficient implementation of Lime, which includes the guard implementation and the Lime runtime system. As a concurrent object-oriented programming model, three problems need to be addressed: 1) how to reduce the complexity of the implementation, 2) how to increase the efficiency of the Lime runtime system, and 3) how to improve the stack mechanism for Lime, which should start with a small size stack and grow on demand.

1.4 Contributions

This research is based on Lime which will be discussed in Chapter 3. My contributions to the research are:

1. First efficient implementation of guard-based synchronizations for Lime, which has a better performance than concurrent programming languages such as Erlang and Go, in fine-grained, highly concurrent benchmarks
2. First implementation of a cooperative scheduling for Lime
3. First implementation of user-level coroutines for Lime
4. First implementation of Lime runtime system which maintains a local object queue for worker thread and a global object queue
5. First lock-free implementation of the local object queue for Lime
6. Evaluation of segmented stack mechanisms for highly concurrent objects

1.5 Structure of the Thesis

The thesis is organized as follows:

- Chapter 2 gives an overview of existing concurrency models and implementations;
- Chapter 3 discusses in detail the previous work on *Lime*;
- Chapter 4 lists the main contributions of this thesis;
- Chapter 5 discusses and compares the performance of stack mechanisms;
- Chapter 6 describes the guarded commands implementation;
- Chapter 7 concludes the thesis with a discussion.

Chapter 2

Concurrency Models

The research of concurrency in computer science begins with the mutual exclusion problem, introduced by Dijkstra (1986), which is used to prevent a shared resource from being accessed simultaneously. All kinds of formal models, languages, and algorithms have been proposed over the last four decades. Concurrency has become essential in the modern programming world. Nevertheless, despite the diversity of the forms, much of the work on concurrency can be classified into two fundamental paradigms:

- Shared Variables: Communication via shared memory (e.g., Concurrent Pascal (Brinch Hansen, 1975), UNITY (Chandy, 1989)): concurrent modules interact by reading and writing shared variables in memory.
- Message Passing: Communication via message passing (e.g., CSP (Hoare, 1978) and Actors (Agha, 1985)): concurrency models based on message passing provide channels and primitives for communication between processes via sending and receiving messages. These vary in the way how the message passing is defined and applied, and how the communication is synchronized.

The thread model we have considered in Chapter 1 is a shared memory model. The shared memory model comes with the notion of atomicity, which is implemented by atomic memory access instructions, for example, *CompareAndSwap* instruction, or guaranteeing the atomicity of the whole code section, such as the action systems (Back and Kurki-Suonio, 1989) and transactional memory (Harris and Fraser, 2003). If an operation on the shared memory cannot be completed atomically, the atomicity of the whole code section is needed. In the shared memory model, multiple threads can access a shared resource at the same time. "The output of execution depends on the order in which the access takes place is called a *race condition*"(Andrews, 1991). Thus, to avoid a race condition, *mutual exclusion* is needed. The shared resource must be protected either by locks or by synchronization constructs, such as semaphores and monitors.

In this chapter, we summarize existing concurrency models and implementations. First, we examine the synchronization constructs and the inherent concurrency models which can be combined with object-oriented concepts. Second, we introduce the concurrency model that supports message passing. Last, we discuss the implementation of these concurrency models.

2.1 Shared Variables

2.1.1 Synchronization

The processes in a concurrent program have to work together to solve one problem correctly via communication. To complete the communication, one process should be capable of exchanging information with another process. That information can be stored in shared memory or exchanged in a communication channel, which means communication can be completed by writing and reading shared memory or sending and receiving messages on the channel.

Communication gives rise to the need for synchronization. In general, concurrent programs employ two kinds of interactions: mutual exclusion and condition synchronization. Mutual exclusion prevents simultaneous access to a shared resource. Mutual exclusion is concerned with combining atomic actions that are implemented directly into sequences of actions called *critical sections*. Dijkstra first introduced the critical section problem in 1965. It was the first problem to be studied extensively and remains of interest since most concurrent programs have critical sections of code. There are n processes in the critical section problem and all of the processes alternatingly execute a critical section then a noncritical section. Each process must obtain the permission to enter the critical section based on the entry protocol. An exit protocol follows the critical section. Thus, the processes have the following form:

Listing 2.1: Critical Section Problem (Andrews, 1991)

```
process CS [i := 1 to n]
   while true do
      entry protocol
      critical section
      exit protocol
      noncritical section
```

We specify atomic actions using angle brackets $\langle$ and $\rangle$. For example, $\langle e \rangle$ indicates that expression e is to be evaluated atomically. We specify synchronization using the await statement, which is a useful statement since it can be used to specify arbitrary, coarse-grained atomic actions. The **await** statement can also specify condition synchronization as follows:

$\langle$**await** B **then** S **end**$\rangle$

This conditional atomic action ⟨ **await** *B* **then** *S* **end** ⟩ delays the executing process until the Boolean expression B is true.

For the critical section problem, we can use one Boolean variable *lock* to indicate when a process is in a critical section.

Listing 2.2: Critical Section Solution Using Locks (Andrews, 1991)

```
process CS [i := 1 to n]
   while true do
      ⟨await not lock then lock := true⟩
      critical section
      lock := false
      noncritical section
```

For the condition synchronization, the fundamental mechanism is to implement the **await** statement. The naive way is to implement **await** by busy waiting or spinning. The unconditional atomic section can be implemented by using the critical section problem's solution. We use **CSenter** and **CSexit** to represent the critical section's entry protocol and exit protocol, respectively. Then we can choose a random delay time within an acceptable range to avoid the collisions, called the *binary exponential back-off protocol*, to implement the **await** statement as follows (Andrews, 1991):

Listing 2.3: An Await Statement Implementation (Andrews, 1991)

```
CSenter
while not B do
   CSexit Delay CSenter
S
CSexit
```

Implementation

In general, the critical section problem can be solved by using a simple tool — a lock, which must be acquired before the critical section and be released after the critical section. Many modern computer systems provide atomic instructions, such as `CompareAndSwap` and `TestAndSet` instructions, to implement synchronization in a relatively simple manner. These atomic instructions perform one operation on a memory location and are guaranteed to succeed or fail in their entirety. We illustrate this solution by using the `TestAndSet` instruction which can be used to solve the critical section problem. The `TestAndSet` instruction works as follows: it reads the data from the memory location into a register and then writes a non-zero value to that location. All the operations of this instruction are guaranteed to be invisible — the other processes cannot access the data until this instruction finishes (Andrews, 1991). If the *lock* is held by another process, the `TestAndSet` instruction returns *true*. Otherwise, it returns *false*.

Listing 2.4: Critical Section Solution Using *TestAndSet* (Andrews, 1991)

```
process CS [i := 1 to n]
   while true do
      while TestAndSet(lock) do skip
      critical section
      lock := false
      noncritical section
```

2.1.2 Semaphores

The semaphore concept was first invented by Dijkstra (1962) and has been extensively employed in operating systems. A semaphore is a shared non-negative counter that is manipulated only by two atomic operations, P and V, and is used to control access to the shared resources. The V operation is used to signal the occurrence of an event, so it increments the value of a semaphore by one. The P operation waits until the value of a semaphore is positive then decrements the value by one. That is, the P operation can block a process until an event has occurred (which is represented by the V operation) (Dijkstra, 1967).

Listing 2.5: Semaphore (Andrews, 1991)

```
   var s : sem
   P(s) = ⟨await s > 0 then s := s − 1⟩
   V(s) = ⟨s := s + 1⟩
```

First, semaphores can be used for access control when the semaphore is set to the number of resources. Second, semaphores also can be used to solve various synchronization questions when the semaphore is declared as a binary semaphore: the value can only be zero or one. Compared with locks, semaphores are more natural for the designers to understand and verify any synchronization problems since the power of semaphores results from the fact that P operations might have to delay.

Implementation

"Semaphores can be implemented by using busy waiting"; however, there is no reason to waste CPU cycles in the operating system kernel (Andrews, 1991). When semaphores are added to the kernel, rather than using busy waiting, the P operation of semaphores can *block* itself. Therefore, processes have one more state that needs to be handled: blocked on a semaphore. In particular, a process is blocked if it is waiting to complete a P operation. To keep track of blocked processes, each semaphore descriptor contains a linked list of the descriptors of processes blocked on that semaphore. The implementation of semaphore primitives is in Listing 2.6.

A semaphore descriptor contains the value of one semaphore and two linked lists: a blocked list and a ready list. The descriptors for ready processes are stored on the

ready list. To keep track of blocked processes, each semaphore descriptor contains a linked list of descriptors of processes blocked on that semaphore. It is common for each linked list to be implemented as a First-In-First-Out (FIFO) queue since this ensures that the semaphore operations are fair. The *executing* variable contains the index of the descriptor of the executing process. When the *dispatcher* is called at the end of a primitive, it checks the value of *executing*. If it is zero, dispatcher removes the first descriptor from the ready queue and sets *executing* to point it. If *executing* is not zero, the current process continues to execute.

Listing 2.6: Semaphore Primitives for A Single-Processor Kernel

```
procedure createSem(value: int, name: int*)
  get an empty semaphore descriptor
  initialize the descriptor
  set name to the name(index) of the descriptor
    dispatcher()
procedure P(name: int*)
  find semaphore descriptor of name
  if value > 0 then
    value := value - 1
  else
    insert descriptor of executing at end of blocked list
    executing := 0
  dispatcher()

procedure V(name: int*)
  find semaphore descriptor of name
  if blocked list empty then
    value := value + 1
  else
    remove process descriptor from front of blocked list
    insert the descriptor at end of ready list
  dispatcher()
```

However, when semaphores are used as synchronization tools to construct a sizable concurrent program, semaphores become difficult to use and error-prone (Andrews, 1991). First of all, semaphores are a low-level mechanism, which means it is easy to make errors when using them. For example, a programmer must make sure all P and V operations correct. One missing P or V operation can crash the whole program. Second, semaphores are global to all processes. Thus, to see how a semaphore is used, one must examine the entire program. Third, both mutual exclusion and condition synchronization use the same pair of primitives: P and V. Without checking all the related semaphores, it is difficult for the programmers to ascertain the purpose of a given P and V.

2.1.3 Conditional Critical Regions

Conditional critical regions (CCRs) are proposed by Hoare (1972) to solve the problems discussed in the previous section when using semaphores. CCRs provide a structured way to define synchronization. With CCRs, shared variables that need to be accessed with mutual exclusion are declared together in resources. Mutual exclusion guarantees that the execution of the region statement is not interleaved. Boolean conditions in region statements provide condition exclusion.

Listing 2.7: CCRs (Hoare, 1972)

```
resource r (variable declarations)
region r when B → S end
```

The CCRs notation employs two mechanisms: resource declarations and region statements. The CCR resource contains one or more variable declarations. In the region statement, B is a Boolean guard, and S includes one or more statements. Both local variables and the variables declared in r can be referenced by B and S.

Implementations

The key to implementing CCRs is to implement region statements. Since region statements are very similar to await statements, they can be implemented in the same way using busy waiting (Listing 2.3). Harris and Fraser (2003) "map CCRs onto software transactional memory (STM) which groups together series of memory accesses and makes them appear atomic. CCRs allow programmers to indicate what groups of operations should be executed in isolation. The programmer can also guard the region by an arbitrary Boolean condition." The CCR conditions are re-evaluated only when the related variables have been updated. Because a native method may access arbitrary memory, the system throws a runtime exception when a native method is called within a CCR.

Listing 2.8: CCRs Implementation Using STM (Harris and Fraser, 2003)

```
bool done := false;
while ¬done
   STMStart()
   try
      if B then
         S
         done := STMCommit()
      else
         STMWait()
   catch
      done := STMCommit()
```

Listing 2.9: Monitor Declaration

```
monitor mName
   declarations of permanent variables
   initialization statements
   procedures
```

However, compared with fine-grained lock-based systems, STM suffers a performance hit on multicore processors. The overhead primarily comes from maintaining the log and committing transactions.

2.1.4 Monitors

With CCRs, mutual exclusion is implicit, and condition synchronization is programmed explicitly. Compared with semaphores, it is easier for programmers to understand and use CCRs. Besides, it leads to a more straightforward proof system. However, region statements in CCRs are more expensive to implement than semaphore operations because guards have to be re-evaluated either every time a shared variable is updated. Because CCRs are relatively inefficient, they have not been as widely used as semaphores.

Monitors (Hoare, 1974) are program modules that provide not only more structure than CCRs but also the same efficiency as semaphores. Monitors are a data abstraction mechanism. A monitor contains its state in variables and provides procedures that modify these variables of the object. All variables of a monitor are private while all procedures are public:

Mutual exclusion in monitors is provided implicitly by allowing at most one process to execute. Condition synchronization is provided explicitly by employing condition variables. A condition variable is used to block a process until some Boolean expressions become true. The declaration of a condition variable has the form:

cond *cv*

The content of a condition variable *cv* is a queue of delayed processes (Andrews, 1991). Initially, this queue is empty. The value of *cv* is not directly visible to the programmer. Instead, it is accessed indirectly by several special operations:

- *wait*(*cv*) blocks the executing process at the tail of *cv*'s queue.

- *signal*(*cv*) awakens the process at the front of *cv*'s queue if this queue is not empty. Otherwise, it has no effect.

Implementations

Monitors can also be readily implemented in the operating system kernel (Andrews, 1991). We assume that procedures execute with mutual exclusion and condition synchronization uses the *Signal and Continue* discipline. That is, signalling does not cause the signalling thread to lose occupancy of the monitor. To implement monitors in the kernel, the following primitives are added: monitor creation, monitor entry, monitor exit, and the operations, such as signal and wait.

Each monitor contains a lock and three queues: an entry queue, a ready queue and a condition variable waiting queue. The entry queue is used to maintain processes waiting to enter the monitor. When the monitor exits, one waiting process of the entry queue is moved to the ready queue if the ready queue is not empty. The *signal_all* operation is implemented by moving all the elements of the condition variable waiting queue to the ready queue. The condition variable contains the pointer of the first waiting process of the waiting queue. Every process — except an executing process — is linked to either the ready queue, the monitor entry queue, or the condition variable queue.

Listing 2.10: Monitor Kernel Primitives (Andrews, 1991)

```
procedure enter(mName: int)
  find decriptor for monitor mName
  if mLock = true then
    insert descriptor of executing at end of entry queue
    executing := 0
  else
    mLock := 0
  dispatcher()
procedure exit(mName: int)
  find descriptor for monitor mName
  if entry queue not empty then
    move process from front of entry queue to rear of ready list
  else
    mLock := 0
  dispatcher()
procedure wait(mName: int, cName: int)
  find descriptor for condition variable cName
  insert descriptor of executing at end of delay queue of cName
  executing := 0
  exit(mName)
procedure signal(mName: int, cName: int)
  find descriptor for monitor mName
  find descriptor of condition variable cName
  if delay queue not empty then
    move process from front of delay queue to rear of entry
       queue
```

```
dispatcher()
```

When a concurrent program uses a monitor for communication and synchronization, this makes the concurrent program more straightforward to design and understand than semaphores, to some extent. There are two critical benefits (Andrews, 1991): first, the programmer who uses a monitor does not need to know how the procedures of the monitor are implemented; second, the developers of a monitor can focus on how the monitor is implemented, as long as they keep the public procedures' interface and functionality the same. That is, there is relative independence between program and monitors.

Turing Plus is designed (Holt and Cordy, 1985) as a concurrent programming language. The concurrency is supported by implementing processes and monitors at the programming language level. As we discussed above, the independence between processes and monitors simplifies the design of a concurrent program in the single-core era. The novel feature of *faithful execution* proposed in Turing (Holt and Cordy, 1988) ensures that the program either executes entirely according to the mathematical specification or generates system exceptions. The keywords *checked* and *unchecked* are introduced to allow the programmers to verify the correctness of the program explicitly, for example, array subscripts and initialization of variables. Faithful execution provides the programmers with a practical approach to check both language constraints and implementation constraints of a concurrent program.

When a concurrent object-oriented program uses a monitor as a synchronization tool, the typical structure of the method is like:

Listing 2.11: Monitor Use Case

```
enter(cv)
   while count = 0 do wait(cv)
   method body
   signal(cv)
exit(cv)
```

However, this independence introduces one more design view for the programmers when they develop the concurrent object-oriented program. Programmers have to handle two different design issues: building the program with objects and handling concurrency of the program by using condition variables, which increases the difficulty of developing, maintaining, and extending the concurrent object-oriented program.

2.1.5 Futures

The concept of *futures*, first introduced by Baker and Hewitt (1977), was used for synchronizations in concurrent programming languages. A somewhat similar concept, called *promise* was proposed by Friedman and Wise (1978). They describe a value that may not exist now but can be used subsequently because the computation has

not finished yet.

future e

The expression e is some computation that may take some time to complete. Future indicates computation e may run concurrently with its parent. The computation e can be done more flexibly, starting either eagerly when the future is created, or lazily when its result is first needed.

Compared with the concepts discussed in the previous sections, futures are a lightweight concept for concurrent programming. That is, futures cannot be used to implement arbitrary synchronization and communication. The implementation of futures depends on the runtime system. Futures can be implemented implicitly as a library or explicitly as part of the programming language.

For example, the following *add* operator expects two integer arguments, thus needs to wait until the computation of $factorial(100000)$ is finished:

$26 +$ **future** $factorial(100000)$

This problem can be solved by using coroutines or generators, to create a new coroutine that executes $factorial(100000)$. The runtime system decides how to schedule this coroutine.

Futures can easily be implemented by using the message passing mechanism (will be discussed in Section 2.2). The problem can be solved by sending a message to *factorial* with the argument, and then a message of "+ *26*" to the receiver. The receiver adds *26* to the result of $factorial(100000)$ and sends the final result back. By using channels, futures can also easily be implemented by using a buffered channel. For example, in the Go programming language, factorial can be mapped to a goroutine which sends the result back via a buffered channel.

2.1.6 Action Systems

Guarded Commands

The Guarded Command Language was defined by Dijkstra (1975) with predicate transformer semantics. Guarded commands were introduced as a design notation for writing programs more abstractly. Nondeterminism is introduced when guarded commands combine with alternative or repetitive constructs. Historically, nondeterminism was first connected to concurrent programs.

The guarded command $G \rightarrow S$ consists of a command S, proceeded by a Boolean expression G, known as a *guard*.

The execution rules state that a guarded command can only be executed when the guard is true. When control reaches an alternative or repetitive construct in

a language with guarded commands, a nondeterministic choice is made among the guards that hold, and the command list following the chosen guard is executed.

It is illuminating to compare guarded commands with traditional clauses. Take Euclid's GCD algorithm as an example: the implementation of the guarded commands in Listing 2.13 "has been reduced to its bare essentials" (Dijkstra, 1975). The repetitive constructs of **do ... od** executes repeatedly until none of the guards are true. For the traditional clauses, programmers need to decide which construct to be used, *if* statement or *while* statement in Listing 2.12. The structure `[]` denotes nondeterministic choices.

Listing 2.12: Euclid's GCD Algorithm Using Traditional Clauses

```
while x ≠ y do
   if x > y then x := x - y
   else x := y - x

while x ≠ y do
   while x > y then x := x - y
   while y > x then x := y - x
```

Listing 2.13: Euclid's GCD Algorithm with Guarded Commands

```
do
   x > y → x := x - y
[]
   y > x → y := y - x
od
```

In addition to an expressive programming notation, guarded commands also provide programmers with a more natural way to prove the correctness of the programs. Combined with Hoare logic, guarded commands have evolved from a programming element into a formal language for modelling concurrent systems. Further, guarded commands become a calculus for studying systems of communicating processes rather than a language for writing application programs.

> The *weakest precondition* of statement list S and predicate Q, denoted $wp(S, Q)$, is a predicate characterizing the largest set of states such that, if the execution of S is begun in any state satisfying $wp(S, Q)$, then the execution is guaranteed to terminate in a state satisfying Q (Dijkstra, 1975).

Weakest preconditions are closely related to correctness assertions `{P} S {Q}` in Hoare logic, which express that under precondition P statement S terminates in postcondition Q:

$$\{P\}\ S\ \{Q\} = (P \implies wp(S, Q))$$

The *wp* function is defined over the structure of statements:

$$wp(skip, R) = R$$
$$wp(abort, R) = false$$
$$wp(S, false) = false$$
$$wp(x := E, R) = R[x \backslash E]$$
$$wp(S_1; S_2, R) = wp(S_1, wp(S_2, R))$$

Weakest preconditions emphasize the calculation aspect as a precondition can be systematically derived for a given postcondition. Following properties follow from the definition:

$$wp(S, P) \wedge wp(S, Q) = wp(S, P \wedge Q)$$
$$wp(S, P) \vee wp(S, Q) \implies wp(S, P \vee Q)$$

The last implication can be strengthened to an equivalence if S is deterministic.
For the alternative construct, let IF denote the following:

if $B_1 \rightarrow SL_1$ [] ... [] $B_n \rightarrow SL_n$ **fi**

If BB denotes:

$$\exists i : 1 \leq i \leq n : B_i$$

then, by definition:

$$wp(IF, R) = (BB \wedge (\forall i : 1 \leq i \leq n : B_i \implies wp(SL_i, R)))$$

"The first term BB requires that the alternative construct will not lead to abortion on account of all guards being *false*; the second term requires that each guarded expression in the list eligible for execution will lead to an acceptable final state" (Andrews, 1991).

For the repetitive construct: let DO denote the following:

do $B_1 \rightarrow SL_1$ [] ... [] $B_n \rightarrow SL_n$ **od**

Let

$$H_0(R) = (R \wedge \neg BB)$$

and for $k > 0$

$$H_k(R) = (wp(IF, H_{k-1}(R)) \vee H_0(R))$$

Then, by definition:

$$wp(DO, R) = \exists k : k \geq 0 : H_k(R)$$

"Intuitively, $H_k(R)$ can be interpreted as the weakest pre-condition guaranteeing proper termination after at most k selections of a guarded list, leaving the system in a final state satisfying R" (Dijkstra, 1975).

Implementations of Guarded Commands

OPS5: OPS5, developed by Forgy (2018), is a rule-based computer language. In OPS5, guarded commands are mapped to production rules. Each production rule contains a LHS and an RHS. The LHS includes one or more condition elements, and the RHS includes one or more actions. Taking the Euclid's GCD algorithm as an example, compared with guarded commands, OPS5's implementation is inefficient. Koa and Hwang (1987) criticize OPS5 because "in some tasks that using OPS5 only results in a mass of inefficient, awkward, and tedious rules".

Listing 2.14: OPS5 Guarded Commands (OPS5, 2013)

```
(literalize number value id)
...
(p gcd-step
  (number ^id <id-1> ^value <larger-value>)
  (number ^id <> <id-1> ^value { <smaller-value> > 0 <= <larger-
     value>} )
  -->
  (modify 1 ^value (compute <larger-value> % <smaller-value>)))
...
```

Occam: *ALT* is introduced in Occam to specify a list of guarded commands. The guards are a combination of Boolean conditions and an input expression. One of the successful commands is selected to execute when the relevant guards hold. That is, the Boolean expression is true, and the input channel is ready.

The program in Listing 2.15 (Talla, 1990) implements an semaphore which is shared among N processes. The P operation is completed by executing `P ! any` while the V operation is achieved by executing `V ! any`. The P operation must be blocked until the value of the semaphore (*val*) is positive.

Listing 2.15: Occam Guarded Commands (Talla, 1990)

```
PROC semaphore ([N]CHAN OF BYTE P, V, end, VAL INT N)
   BYTE any:
   INT i, val, tot.users.term:
   SEQ
      val := 0
      tot.users.term := 0
      WHILE ( tot.users.term < N )
         SEQ
```

```
ALT i = 0 FOR N
   V[i] ? any
      val := val + 1
ALT i = 0 FOR N
   (val > 0) & P[i] ? any
      val := val - 1
ALT i = 0 FOR N
   end[i] ? any
      tot.users.term := tot.users.term + 1
```

In Occam, output processes are not allowed in guards in an ALT constructor. This makes some algorithms harder to program but dramatically simplifies the implementation of the *transputer* (Whitby-Strevens, 1985), which used an unusual hardware architecture to achieve higher performance. Occam is based on the Communicating Sequential Processes model which will be discussed in the Section 2.2.2.

Ada: Communication in Ada is defined by declaring "entries" in the task declaration. Tasks have *accept* alternatives in a selective wait statement. The execution of any alternatives can be controlled by using a guard. In Ada, an alternative with a *true* guard is called *open*. If at least two alternatives are open, one of them would be chosen to execute by arbitrary decision of the implementation, which introduces nondeterminism in the program.

Listing 2.16: Ada Guarded Commands (Andrews, 1991)

```
protected type Semaphore is
   entry P;
   procedure V;
   private Value : Integer := 0;
end Semaphore;
protected body Semaphore is
   entry P when Value > 0 is
   begin
      Value := Value - 1;
   end P;
   procedure V is
   begin
      Value := Value + 1;
   end V;
end Semaphore;
task body Semaphore is
begin
   loop
      accept P;
      accept V;
   end loop;
```

```
end Semaphore;
```

Performance is also the bottleneck of Ada's guarded commands implementation. Besides, the combination of concurrency and object-oriented programming in Ada needs much work (Moore, 2010).

CIVL: CIVL, developed by Siegel et al. (2015), stands for *Concurrency Intermediate Verification Language* and encompass a C-based programming language, called CIVL-C, verification and analysis tools and the translators which translate commonly used languages and APIs to CIVL-C. A guarded commands is defined by using a $*when* statement:

Listing 2.17: CIVL-C Guarded Commands

```
$when (expr) stmt;
```

Stmt is enabled when *expr* is *true* and *stmt* is disable when *expr* is *false*. A enabled *stmt* will be scheduled for execution. The evaluation of *expr* and the first atomic action of *stmt* are executed atomically. However, there is no guarantee that the execution of *stmt* is atomic if *stmt* contains more than one action.

Action Systems

Action systems were first introduced by Back and Kurki-Suonio (1989), where it was applied to the step-wise refinement of distributed algorithms. This formalism stems from an extended version of guarded commands language, which introduced the concept of guards and non-determinism.

System behaviours in action systems focus on coordinating the actions, rather than executing the sequential code of the action. There are two different execution models for action systems: a sequential execution model and a concurrent execution model. The sequential execution model provides programmers with a more straightforward approach to design, understand, and reason about large concurrent programs, whereas the concurrent execution model may exploit the inherent concurrency of the programs.

In general, actions are guarded commands of the form $G \rightarrow S$ where G is a Boolean expression and S is a statement. An action is enabled if the guard is true; otherwise, the action is disabled.

Listing 2.18: Action System

```
A = |[ var l := l0; g := g0
      do Ai[] ... [] Am od
    ]|
```

An action system A is a set of actions which operates on local and global variables. Action system A first creates and initializes the local variables to l_0 and global variables to g_0, respectively. Then, A describes computations on these variables.

The behaviour of an action system is the same as a single repetitive construct in a guarded command language or a production system, for example, an OPS5 program (Kurki-Suonio and Järvinen, 1989). Compared with the other language mechanisms, actions systems are designed for concurrent execution, rather than sequential execution. Additionally, action systems allow infinite computation.

The local and global variables in action systems are distinct. An enabled action A_i is chosen for execution nondeterministically after the initialization. Actions are executed atomically without any interference. In this case, actions which operate on disjoint sets of variables can be executed concurrently. So if two actions are enabled and do not have any read-write conflicts, they can be executed in any order. The execution terminates if there is no enabled action; otherwise, it continues infinitely.

Action systems that model concurrency by combining actions with objects can help programmers to simplify both the specification and design of concurrent programs. However, this is still theoretical. There are two programming models which combine the object-oriented concepts with action systems: Seuss and OO-action.

As a programming notation, an action system can be extended from the original form to object orientation. Over the last couple of decades, many action-based object-oriented concurrent models have been proposed and described in the literature (Lamport, 1994; Chandy and Misra, 1988; Back et al., 1997).

Seuss: Seuss, developed by Misra (2001), is a programming model that integrates action systems with object-based programming. It supports features for object definition and instantiation and allows designated methods of the object instances to be executed concurrently. Seuss does not provide any particular communication or synchronization mechanism, except procedure call. Objects in Seuss can encode the traditional schemes for communication, synchronization, interfaces among processes, and accesses to shared memory. Since dynamic creations of objects can lead to an inefficient implementation, one can only declare a fixed number of objects in Seuss. Seuss requires the execution of an action, when started in a state where its guards hold, to terminate. An action in Seuss is viewed as an atomic "unit that is either executed to completion or not executed at all" (Misra, 2001).

At runtime, a program in Seuss contains a set of objects and their states are initialized before execution. Seuss has two different execution modes: a *tight* mode (sequential) and a *loose* mode (concurrent). In the *tight* execution mode, "one action is executed at a time", while in the *loose* execution mode, "actions may be executed concurrently" (Misra, 2001). The *tight* execution provides programmers with an easy approach to reason about a program. The *loose* execution may improve the performance of the program in Seuss.

OO-action: Bonsangue et al. (1998) proposed an approach to integrate object-orientation concepts with action systems, so-called the OO-action system. The extended action system formalism provides the objects with the support of dynamical

creation. "Communication between objects takes place via remote procedure calls and shared variables" (Bonsangue et al., 1998).

Discussion of Action Systems

The enabled actions in the OO-action system can be selected and executed concurrently. For the actions of the same active object, they can be executed simultaneously if they work on the disjoint sets of local and shared attributes and object fields. For the actions from different active objects, they can be executed concurrently as long as they operate on disjoint sets of shared attributes. The communication in the OO-action system is achieved by using shared attributes and method calls.

In Seuss and the OO-action system, the concurrency of objects can be defined as autonomous because the objects could be active even if there is no method call. The communication between objects is through synchronous method calls. Condition synchronization is achieved by using guards. Seuss can only declare a fixed number of objects while the OO-action system allows dynamic object creation.

The OO-action system only provides a theoretical model that combines the action concepts and object-oriented concepts. The developers follow the theory of action systems in their implementation of the OO-action system. An action in the OO-action system should be viewed as "a unit that is either executed to completion or not executed at all" (Misra, 2001). In this case, if there are multiple method calls in one action and one of the method calls is disabled, the action is therefore not enabled. A rollback mechanism, which requires an elaborate implementation, has to be added to the OO-action system.

Seuss not only provides a theoretical model but also discusses the outline of the implementation strategy. To avoid the rollback in the OO-action system, Seuss decides to allow a call to a guarded method to be only the first statement in an action or a method. Besides being inconvenient, this forbids that an unguarded method is refined by a guarded one because a guarded method in Seuss cannot be called by an unguarded method body.

The models, such as Seuss and the OO-action system, aim to combine concurrent concepts with object-oriented ones. The notion of a guard provides the programmers with a unified and straightforward design view for the large concurrent program: they do not need any synchronization constructs, such as P and V operations of semaphores, or wait and signal operations of monitors. The actions are suspended or resumed implicitly based on the value of the guards. However, "the price that must be paid for this automatic scheme is performance" (Briot et al., 1998).

When there are multiple method calls in the actions and methods, the rule of the atomicity of the actions and methods varies. Suppose an unguarded action contains two method calls: the first one is calling method m of object x and the second one is calling method n of object y. The method n of object y is not enabled. In the OO-action systems, following the theory of action systems, the whole action is therefore not enabled; thus, the whole action has to roll back when the second method call is

disabled. These restrictions of the execution model make it awkward for programmers to construct concurrent programs. Therefore, a new atomicity rule of the actions and methods is preferred.

Implementation of Seuss

Krüger (1996) designs and implements a compiler and a runtime system for Seuss. The compiler translates the Seuss programs into C++ code, which may be distributed over computer networks using Parallel Virtual Machine (Geist et al., 1994). Based on Krüger's work, Joshi (1998) designs a language Seuss for Java, which is an adaptation of the Seuss notation for use with Java. For both Java and C++, the overhead of context switch becomes significant when the number of objects exceeds the available number of cores.

2.2 Message Passing

The synchronization constructs we have discussed so far are based on shared variables. In this section, we have a look at another mechanism: message passing. With message passing, processes share *channels* which provide communication paths between processes and two primitives on the channel: *send* and *receive*. Compared with semaphores, message passing not only provides synchronization but also conveys data.

The sender process releases a message to a channel, and the receiver process acquires the message from the channel. Communication is accomplished since data flows from the sender to the receiver. Communication can be asynchronous or synchronous.

2.2.1 Asynchronous Message Passing

With asynchronous message passing, communication channels are unbounded message queues. The *send* statement appends the message to the tail of the queue while the *receive* statement acquires a message from the head of the queue. In this case, the send statement does not get blocked because the queue is unbounded. The receive statement may get blocked when the channel is empty.

Asynchronous message-passing primitives have been included in several programming languages and operating systems. We illustrate the asynchronous message passing by using the actor model.

The actor model (Agha, 1985) stems from Hewitt (1971)'s work on the AI system *Planner* in the early 1970's. The actor model uses an explicitly concurrent notion, and actors are used as concurrent primitives that can have different behaviours based on receiving messages:

- Send messages to itself or other actors;
- Create new actors;

- Specify a replacement behaviour which takes effect when the next message comes.

In the actor model, a communication event is called a *task*, which has three parts:

- A unique tag of this task in the system;
- A target, the mail address of the receiver;
- The data conveyed by this communication event.

Actors send messages asynchronously, which means that messages will arrive in the receiver's mailbox eventually but may take an uncertain length of time. Also, messages in the actor model are not guaranteed to arrive in the same order in which they are sent. The operations of *dequeue* and *enqueue* of the messages are atomic, so the race condition in the actor model can be avoided. An actor processes messages from its mailbox sequentially. With actors, computation is passive and is performed as a "response" to communication. The replacement behaviour determines the response to the subsequent communication and takes effort for the next message.

In the actor model, because "everything is an actor" and each actor is a completely independent instance, it is straightforward for the programmers to model inherently concurrent systems. The communication in the actor model relies on the asynchronous message passing. An actor can handle several requests simultaneously. Based on the classification provided by Briot et al. (1998), the concurrency level of the actor model can be characterized as *intra-object*.

For asynchronous message passing, three consequences result from the unbounded channels. First, messages have to be buffered, yet space is finite in practice. The program may crash if the message queue overflows. Second, message delivery is not guaranteed if a failure occurs. If process A sends a message to process B and does not get a reply, then process A does not know whether the message was sent, process B crashed while acting on it, or the response could not be delivered. Third, process A can get arbitrarily far ahead of process B.

2.2.2 Synchronous Message Passing

Synchronous message passing avoids these three consequences. With synchronous message passing, both *send* and *receive* are blocking primitives. The sender process delays until the receiver process receives messages from the channel. Thus, sender and receiver synchronize at every communication point. The sender process can proceeds only when the message was indeed delivered. Compared with asynchronous message passing, synchronous message passing does not require dynamic buffer allocation.

The programming notation of synchronous message passing is similar to Communicating Sequential Processes (CSP), and Hoare extended the guarded command idea in CSP (Hoare, 1978) by allowing the guards to become dependent upon the behaviour

of another component of the program. It was first presented as a theoretical approach to the problem of communication between concurrent processes.

In this section, we illustrate synchronous message passing by using CSP programming notation.

In CSP, the notion of a guard and a guarded command are extended to include input and output guards. Dijkstra's guards are simple Boolean expressions and can have two statuses: *true* or *false*. CSP introduces a third status for the guards, called *pending*. In this case, the general form of the CSP guard is either a Boolean expression or an I/O guard of the following two forms:

1. c ? v : describes the operation of accepting a value on channel c, and assigning that value to the variable v.

2. c ! e : describes the operation of evaluating the expression e, the result of which is delivered on the channel c.

In CSP, the fundamental operators are prefixing, recursion, and guarded alternatives. The prefixing operator $\rightarrow$ takes a single event on the left and a process, which engages in events (maybe many or even none events) on the right. Recursion can be used to implement repetition. Guarded alternatives provide nondeterministic choices if the guards in alternatives hold simultaneously.

Listing 2.19: Euclid's GCD Algorithm Implemented in CSP

```
method GCD()
   var x, y: int
   Output ? args(x, y)
   do
      x > y → x := x - y
   []
      x < y → y := y - x
   od
   Output ! result(x)
```

Method *GCD* waits to receive input on its *args* port from a single client process. *GCD* then computes the answer using Euclid's algorithm and sends the answer back to the client's *result* port. The operator $\rightarrow$ used in Listing 2.19 is a guard, not involving communication.

2.2.3 Implementations

The Actor Model: Erlang (Armstrong et al., 1996) is the first inherently concurrent programming language which implements the actor model. The actor model has been applied by more and more programming languages, such as Scala, which provides standard support for the actor model.

The philosophy of the actor model is that "everything is an actor". There is no shared resource in the system. Only immutable data and addresses of actors can be sent through massages. Because the message is asynchronous in the actor model, the messages may not arrive in the same order in which they are sent. However, it is possible for the sender to tag each message with a unique, increment sequence number so that the receiver may rearrange messages into the correct order.

For message handling, Erlang provides both pattern matching and selective receive semantics. For message passing in the actor model, the receiving operation is a blocking operation while the sending operation is always non-blocking.

However, there are several limitations of the actor model. First, asynchronous buffered communication is useful in many cases, but the uncertain order of message arrival might be inconvenient in some programs. Second, objects in the actor model can dynamically change their interfaces (behaviour) by receiving messages, which means the class notion is fragile. This makes static analysis and optimization of the actor's mail system difficult. Third, there is no inheritance and other object-oriented features in the actor model.

CSP: CSP influences the designs of Occam (INMOS Limited, 1984) and Go (Google, 2009). Occam was developed at first for a specific device, known as the *transputer*. Unlike Occam, Go is a general-purpose concurrent programming language that supports a kind of coroutines and synchronous channels.

The concurrency model in Go is based on two fundamental elements: goroutines and channels. Go supplies lightweight threads, called goroutines. The runtime system distributes goroutines into a fixed number of underlying worker threads. The internal scheduler organizes goroutines in a pool and works cooperatively. Channels are implemented as the first-level language's primitives. Go has rich support for message passing, including both synchronous and asynchronous. Besides channels, Go also allows shared variables and implements other synchronization constructs such as semaphores and mutexes.

Chapter 3

Lime

In this chapter, we first discuss why we propose Lime and then introduce Lime by giving its syntax and several sample programs written in it.

3.1 What is Lime

Lime[1] is an action-based object-oriented concurrent programming language, which was developed by the research group of Dr. Emil Sekerinski at McMaster University. The most significant difference between Lime and the traditional OOPLs is that actions and guarded methods are added to implement action systems. In Lime, concurrency is expressed by extending classes with actions and guarded methods. Guarded methods can be suspended when the guard does not hold. Unlike methods, actions in Lime can only be "called" by the scheduler and are executed concurrently in the background. Therefore, each object in Lime has a private lock at its creation. We add the *when* keyword to allow a method call in Lime to be suspended when the guard fails to hold, which means Lime allows guarded methods and actions to "get stuck" at the method calls.

3.2 Why Lime

Action-based object-oriented programming languages simplify both the specification and design of concurrent objected-oriented applications. Following the theory of action systems, if there are multiple method calls in an action, either some restrictions or roll back mechanisms have to be added. We note that for verification (proving the implementation correct) and refinement (transformation from a specification to an implementation) programs have to be translated such that "method calls appear only as the first statement in methods and actions" (Sekerinski, 2002; Back et al., 1997).

[1]The Lime discussed in this thesis is unrelated to the IBM Lime Language.

The underlying refinement theory (Büchi and Sekerinski, 2000) assumes this form of program.

The OO-action system provides a rollback mechanism to recovery when not all the methods calls are enabled in a method or an action. In Seuss, this is solved by allowing a call to a guarded method to be only the first statement in an action or a method.

Unlike Seuss, Lime does not have the restriction which is used to avoid rollback but allows an action or method to get stuck at the point where a method is called. That is, "actions and methods are atomic only up to method calls". This atomic rule provides programmers with a more flexible and convenient design view. Lime compiler accepts programs which contain multiple method calls in a method or an action.

Lime was invented as an inherently concurrent object-oriented programming language, and concurrency is one of the essential features of Lime. The purpose of this work is to increase the efficiency of Lime by improving the implementation.

3.2.1 Actions

Lime's class system is similar to Java. Actions in Lime are declared inside classes and executed automatically. Each object has a copy of the actions of the class. "They are not referenced within the program but are invoked by the scheduler" (Lou, 2004). Lime starts with a single instance, similar to the "main" function in C, which creates N (usually equals to the number of cores) worker threads. Each worker thread is an instance of the scheduler. Each worker thread has a local object queue. The worker thread adds new objects to its local object queue and moves half of the objects to the global queue when its local queue is full. To improve the performance, there is a lock-free local queue implementation of Lime, discussed in Section 6.3.3, the worker thread could steal objects from other threads. The worker thread selects one object from its queue and then evaluates the guard. The worker thread executes the action when the guard is evaluated to *true*. Otherwise, the worker thread goes to the next object. In this case, Lime can support at most N actions to execute in parallel. Actions have no arguments or return values. There are two kinds of actions in Lime, *guarded* actions and *unguarded* actions.

```
action A
   when b do
      S
```

A is the name of the action, b is a Boolean expression and S is the body of the action. This action is enabled if b holds, otherwise it is disabled. The action is executed when the guard is evaluated to true. In theory, the scheduler re-evaluates the guard after the action is executed. That is, the action should be executed forever if the guard is always true.

```
action A
   S
```

An unguarded action means that the condition to execute this action is always true. In other words, action A is always enabled. The worker thread maintains a first-in-first-out local object queue. The worker thread first executes all the actions of the object, and then adds the object to the tail of the local queue if there are enabled actions.

3.2.2 Methods

Unlike actions, methods in Lime may have arguments and return values. There are two types of methods in Lime, *guarded* methods and *unguarded* methods. Guarded methods may accept or suspend the call:

```
method M(Arg1, ..., Argn)
   when b do
      S
```

M is the name of the guarded method, b is a Boolean expression and S is the body of the method. The guarded method is enabled if b holds, otherwise it is disabled.

An unguarded method is the same as a method in other class-oriented programming languages, such as in Java:

```
method M(Arg1, ..., Argn)
   S
```

For the ease of discussion, the following terminology is defined to distinguish objects in Lime:

- Guarded Object: An object that has either at least one guarded method or at least one action, or has both guarded methods and actions.
- Unguarded Object: An object that has neither guarded methods nor actions.
- Active Object: An object that has at least one action.
- Passive Object: An object that does not have any actions.
- Enabled Object: An active object that has at least one enabled action.
- Disabled Object: An active object that has no enabled actions.

3.2.3 Method Calls

Each object in Lime has a private lock. The atomicity policy is that "all methods and actions are atomic up to method calls." That is, this lock should be released when entering the method call while this lock should be acquired when returning from the method call. The method call translation is described in Listing 6.15.

3.3 Lime Syntax

The current Lime syntax is based on the previous Lime syntax as presented by Lou (2004) using indentation rather than the *begin-end* structure to construct a code block. Compared with the *begin-end* structure, indentation takes fewer lines of code, which is helpful for conveying the structure of a large program to the readers as well as for publishing programs. In this section, we introduce the new concrete syntax of Lime by giving a *context-free grammar* (*grammar* for short) for this language, described in EBNF (Wirth, 1977). The abstract syntax of Lime is unchanged. Here, INDENT stands for having more spaces then the previous line. This sets a new level of indentation. NL stands for a new line and retaining the level of indentation as the previous line. DEDENT stands for setting the level of indentation to the previous level (Van Rossum and Drake, 2011). The construct $(a \mid b)$ stands for either a or b, $[a]$ means that a is optional, and $\{a\}$ means that a can be repeated zero or more times.

⟨*compilation_unit*⟩ ::= {⟨*import_stmt*⟩} {⟨*const_decl*⟩} ⟨*class_decl*⟩ {⟨*class_decl*⟩} ⟨*EOF*⟩;

⟨*import_stmt*⟩ ::= **import** ⟨*ID*⟩ '(' ⟨*type_list*⟩ ')' [':' ⟨*type*⟩] NL;

⟨*const_decl*⟩ ::= **const** ⟨*ID*⟩ '=' ⟨*INTEGER*⟩ NL;

⟨*class_decl*⟩ ::= **class** ⟨*ID*⟩ NL INDENT {⟨*class_member*⟩} DEDENT;

⟨*class_member*⟩ ::= ⟨*field_decl*⟩ | ⟨*init_decl*⟩ | ⟨*method_decl*⟩ | ⟨*action_decl*⟩;

⟨*field_decl*⟩ ::= **var** ⟨*id_list*⟩ ':' ⟨*type*⟩ NL;

⟨*init_decl*⟩ ::= **init** ⟨*parameters*⟩ ⟨*block*⟩;

⟨*method_decl*⟩ ::= **method** ⟨*ID*⟩ ⟨*parameters*⟩ [':' ⟨*type*⟩]
[NL INDENT **when** ⟨*guard*⟩ **do**] ⟨*block*⟩ [DEDENT];

⟨*action_decl*⟩ ::= **action** ⟨*ID*⟩
[NL INDENT **when** ⟨*guard*⟩ **do**] ⟨*block*⟩ [DEDENT];

⟨*parameters*⟩ ::= '(' [⟨*type_pars_list*⟩] ')';

⟨*type_pars_list*⟩ ::= ⟨*pars_def*⟩ {',' ⟨*pars_def*⟩};

⟨*pars_def*⟩ ::= ⟨*ID*⟩ ':' ⟨*type*⟩;

⟨*type*⟩ ::= **int** | **bool** | **void** | ⟨*ID*⟩ | ⟨*array_decl*⟩;

⟨*array_decl*⟩ ::= **array of** (**int** | **bool** | ⟨*ID*⟩);

⟨*type_list*⟩ ::= ⟨*type*⟩ {',' ⟨*type*⟩};

⟨*stmt*⟩ ::= ⟨*simple_stmt*⟩ | ⟨*compound_stmt*⟩;

⟨*simple_stmt*⟩ ::= ⟨*small_stmt*⟩ {';' ⟨*small_stmt*⟩} [';'] NL;

⟨*small_stmt*⟩ ::= ⟨*multi_assign*⟩ | ⟨*expr_stmt*⟩ | ⟨*local_decl*⟩
| ⟨*return_stmt*⟩ | ⟨*method_call*⟩;

⟨*multi_assign*⟩ ::= ⟨*id_list*⟩ ':=' ⟨*expr_list*⟩;

⟨*single_assign*⟩ ::= ⟨*id_ele*⟩ ':=' ⟨*expr*⟩;

⟨*compound_stmt*⟩ ::= ⟨*if_stmt*⟩ | ⟨*while_stmt*⟩ | ⟨*for_stmt*⟩;

⟨*local_decl*⟩ ::= **var** ⟨*id_list*⟩ ':' ⟨*type*⟩;

⟨*expr_stmt*⟩ ::= ⟨*expr_list*⟩;

⟨*if_stmt*⟩ ::= ⟨*if_stat*⟩ {⟨*elif_stat*⟩} [⟨*else_stat*⟩];

⟨*if_stat*⟩ ::= **if** ⟨*expr*⟩ **then** ⟨*block*⟩;

⟨*elif_stat*⟩ ::= **elif** ⟨*expr*⟩ **then** ⟨*block*⟩;

⟨*else_stat*⟩ ::= **else** ⟨*block*⟩;

⟨*while_stmt*⟩ ::= **while** ⟨*expr*⟩ **do** ⟨*block*⟩;

⟨*for_stmt*⟩ ::= **for** ⟨*single_assign*⟩ **to** ⟨*expr*⟩ **do** ⟨*block*⟩;

⟨*return_stmt*⟩ ::= **return** [⟨*expr*⟩];

⟨*block*⟩ ::= ⟨*simple_stmt*⟩ | NL INDENT ⟨*stmt*⟩ {⟨*stmt*⟩} DEDENT;

⟨*guard*⟩ ::= ⟨*guard_atom*⟩ ('<=' | '>='|'<'|'>') ⟨*guard_atom*⟩
| ⟨*guard_atom*⟩ ('=' | '!=') ⟨*guard_atom*⟩
| ⟨*guard_atom*⟩ **and** ⟨*guard_atom*⟩
| ⟨*guard_atom*⟩ **or** ⟨*guard_atom*⟩
| ⟨*guard_atom*⟩;

⟨*guard_atom*⟩ ::= ⟨*ID*⟩ | ⟨*INTEGER*⟩ | **not** ⟨*ID*⟩;

⟨*id_list*⟩ ::= ⟨*id_ele*⟩ {',' ⟨*id_ele*⟩};

⟨*id_ele*⟩ ::= ⟨*ID*⟩ [⟨*selector*⟩];

⟨*expr_list*⟩ ::= ⟨*expr*⟩ {',' ⟨*expr*⟩};

⟨*expr*⟩ ::= '-' ⟨*expr*⟩
| **not** ⟨*expr*⟩
| ⟨*expr*⟩ ('*' | '/' | '%') ⟨*expr*⟩
| ⟨*expr*⟩ ('+' | '-') ⟨*expr*⟩
| ⟨*expr*⟩ ('<=' | '>=' | '<' | '>') ⟨*expr*⟩
| ⟨*expr*⟩ ('=' | '!=') ⟨*expr*⟩
| ⟨*expr*⟩ **and** ⟨*expr*⟩
| ⟨*expr*⟩ **or** ⟨*expr*⟩
| ⟨*atom*⟩;

⟨*atom*⟩ ::= ⟨*INTEGER*⟩
| **true**
| **false**
| **nil**
| ⟨*ID*⟩
| ⟨*method_call*⟩
| ⟨*array_decl*⟩
| ⟨*array_ele*⟩;

⟨*method_call*⟩ ::= **new** ⟨*ID*⟩ ⟨*args*⟩ | ⟨*ID*⟩ '.' ⟨*ID*⟩ ⟨*args*⟩ | ⟨*ID*⟩ ⟨*args*⟩;

⟨*array_decl*⟩ ::= **new** (**int** | **bool** | ⟨*ID*⟩) ⟨*selector*⟩;

⟨*array_ele*⟩ ::= ⟨*ID*⟩ ⟨*selector*⟩;

⟨*selector*⟩ ::= '[' ⟨*expr*⟩ ']';

⟨*args*⟩ ::= '(' [⟨*expr_list*⟩] ')';

A field in Lime is a local variable defined in a class. Fields can only be accessed by the methods and actions of the same object. Local variables defined in the methods

or actions can only be accessed inside of the methods or actions. There are no global variables in Lime. For the guards, only the basic logic and comparison operations are allowed. Method calls are not allowed in the guards. The current implementation of Lime only supports static *array*. That is, the size of the array should be known during the compile time. The *init*() is executed once when an object is created. By default, all the fields are set to 0.

3.4 Lime Semantics

In this section, the axiomatic semantics of Lime is defined in two steps. First, a minimal core language with verification rules is introduced. Then, the elements of Lime are defined in terms of the core language and the verification rules of Lime are derived. The core language is the standard guarded command language with atomic statements, parallel composition, and recursive procedures with verification rules based on Owicki and Gries (1976). The definition of Lime in terms of the core language is novel in the way how each object is defined as one process and how methods are defined to be atomic up to method calls. The definition allows the verification rules of Lime to be easily derived from those of Owicki and Gries. The formal definition serves as the reference of the implementation and provides a correctness theory for Lime programs.

3.4.1 Core Language

The formalization builds on a typed, higher-order logic. Guards are Boolean expressions. For the purpose of defining the core language, the exact syntax of expressions does not need to be specified. In this section, we follow a standard exposition of verification rules (Apt and Olderog, 2019; Apt et al., 2010; Andrews, 2000) without making any significant changes of our own. Statements of the core language are inductively defined as consisting of:

- **skip**, the *empty statement*;
- $\overline{x} := \overline{e}$, the *multiple assignment*, for list $\overline{x}$ of variables and list $\overline{e}$ of expressions;
- $x :\in e$, the *nondeterministic assignment*, which assigns any value of set expression e to variable x or blocks if e is empty;
- $S_0; S_1$, the *sequential composition*, for statements S_0 and S_1;
- **if** $b_0 \to S_0 \,[]\, b_1 \to S_1$, the *conditional statement*, which executes statement S_i if guard b_i is true, choosing arbitrarily if both b_0 and b_1 are true and blocking if neither is true;

- **do** $b_0 \to S_0 \,[]\, b_1 \to S_1$, the *repetitive statement*, which executes statement S_i if guard b_i is true and starts over again, choosing arbitrarily if both b_0 and b_1 are true and terminating if neither is true;

- **var** $\overline{x} : X; S$, the *local variable declaration* of a list $\overline{x}$ of variables of type X with statement S as scope;

- $\overline{x} := m(\overline{e})$, the *procedure call* of procedure m with actual value parameters $\overline{e}$ and actual result parameters $\overline{x}$;

- $\langle b \to S \rangle$, the *await statement* with guard b and statement S;

- $\| \; i \in I \cdot S_i$, the *parallel composition* of statements S_i, called processes.

The conditional and repetitive statements generalize to more than two alternatives and specialize to one alternative. In the await statement, the body does not contain repetitions, procedure calls, await statements, or parallel compositions. Thus, await statements cannot be nested. The parallel composition specializes to a finite number or processes, written $S_0 \parallel \cdots \parallel S_n$. When writing statements, sequential composition binds tighter than other constructs, e.g. **if** $b \to S \,[]\, c \to T_0; T_1$ is understood as **if** $b \to S \,[]\, c \to (T_0; T_1)$. Indentation is used instead of explicit parenthesis and ; is left out at the end of a line.

A *concurrent program* consists of a set of declarations of *global variable declarations*, which can be uninitialized, **var** $\overline{x} : X$, or initialized, **var** $\overline{x} : X := \overline{x_0}$; a set of (global) *procedure declarations* of the form **procedure** $m(\overline{v} : V) \to (\overline{w} : W)$ S where $\overline{v}$ is a list of formal value parameters, $\overline{w}$ is a list of formal result parameters, statement S is the body; and the main program itself, a statement. The main program is a potentially nested parallel composition; parallel composition is only considered on the outer levels, not inside any of the other constructs. A program is executed by first initializing the global variables and then executing all processes in parallel such that on termination the final state can be observed.

Statements $\overline{x} := \overline{e}$, $x :\in e$, $\langle b \to S \rangle$ are *atomic*: they are assumed to be executed without interference by other processes. The *correctness assertion* $\{p\}S\{q\}$ states that if statement S is started in a state satisfying predicate p, the state upon termination will satisfy q. A *predicate* is a Boolean expression that is used in an assertion; it does not need to be executable, unlike a Boolean expression in a program. In an *annotated statement* assertions are interspersed before and after statements; these are used to establish the correctness of statements, for which we employ their *axiomatic definition*. The left side of the following definitions is a Hoare triple, whose meaning is given on the right side.

Definition (Statement Correctness)**.** Let p, p_i, q, q_i, r be predicates.

$\{p\}$ **skip** $\{q\}$	if	$p \Rightarrow q$
$\{p\}\ \overline{x} := \overline{e}\ \{q\}$	if	$p \Rightarrow q[\overline{x}\backslash\overline{e}]$
$\{p\}\ x :\in e\ \{q\}$	if	$p \Rightarrow (\forall x \in e \cdot q)$
$\{p\}\ S_0; S_1\ \{q\}$	if	$\{p\}\ S_0\ \{r\}$
		$\{r\}\ S_1\ \{q\}$
$\{p\}$ **if** $b_0 \to S_0 \,[]\, b_1 \to S_1\ \{q\}$	if	$\{p \wedge b_0\}\ S_0\ \{q\}$
		$\{p \wedge b_1\}\ S_1\ \{q\}$
$\{p\}$ **do** $b_0 \to S_0 \,[]\, b_1 \to S_1\ \{q\}$	if	$\{p \wedge b_0\}\ S_0\ \{p\}$
		$\{p \wedge b_1\}\ S_1\ \{p\}$
		$p \wedge \neg b_0 \wedge \neg b_1 \Rightarrow q$
$\{p\}$ **var** $\overline{x} : X; S\ \{q\}$	if	$\{p\}\ S\ \{q\}$ $\overline{x}$ not in p, q
$\{p\}\ \overline{x} := m(\overline{e})\ \{q\}$	if	**procedure** $m(\overline{v} : V) \to (\overline{w} : W)\ S$
		$\{p'\}\ S\{q'\}$ $\overline{w}$ not in p', $\overline{v}$ not in q'
		$p \Rightarrow p'[\overline{v}\backslash\overline{e}]$
		$q'[\overline{w}\backslash\overline{e}] \Rightarrow q$
$\{p\}\ \langle b \to S\rangle\ \{q\}$	if	$\{p \wedge b\}\ S\ \{q\}$
$\{p\}\ \ \| \ i \in I \cdot S_i\ \{q\}$	if	$(\forall i \in I \cdot \{p_i\}\ S_i\ \{q_i\})$
		$p \Rightarrow (\forall i \in I \cdot p_i)$
		$(\forall i \in I \cdot q_i) \Rightarrow q$
		S_i do not interfere with S_j for $i \neq j$

The notation $f[\overline{x}\backslash\overline{e}]$ stands for substituting all free occurrences of variables $\overline{x}$ in expression f with expressions $\overline{e}$. An atomic statement S that starts in p *does not interfere* with q if $\{p \wedge q\}\ S\ \{q\}$. Suppose (composed) statements S, T have an annotation before each atomic statement. Then S *does not interfere* with T if all atomic statements of S do not interfere with any annotation of T.

For example, consider the annotated program:

$$
\begin{array}{l}
\{true\}\ y := x\ \{x = y\} \\
(\ \{x = y \vee x = y + 2\}\ x := x + 1\ \{x = y + 1 \vee x = y + 3\}\ \| \\
\ \ \{x = y \vee x = y + 1\}\ x := x + 2\ \{x = y + 2 \vee x = y + 3\}\ ) \\
\{x = y + 3\}
\end{array}
$$

The assignment $y := x$ is correct as $true \Rightarrow (x = y)[y\backslash x]$ according to the rule for assignments. For the parallel composition, we have

- $\{x = y \vee x = y + 2\}\ x := x + 1\ \{x = y + 1 \vee x = y + 3\}$;
- $\{x = y \vee x = y + 1\}\ x := x + 2\ \{x = y + 2 \vee x = y + 3\}$;
- $x = y \Rightarrow (x = y \vee x = y + 2) \wedge (x = y \vee x = y + 1)$;
- $(x = y + 1 \vee x = y + 3) \wedge (x = y + 2 \vee x = y + 3) \Rightarrow x = y + 3$;
- $x := x+1$ when started in $x = y \vee x = y+2$ does not interfere with $x = y \vee x = y + 1$ and with $x = y + 2 \vee x = y + 3$;
- $x := x+2$ when started in $x = y \vee x = y+1$ does not interfere with $x = y \vee x = y + 2$ and with $x = y + 1 \vee x = y + 3$.

According to the rule for parallel composition, it allows one to conclude:

$$\{x = y\}\ x := x + 1 \parallel x := x + 2\ \{x = y + 3\}$$

Finally, the rule for sequential composition with $x = y$ as the intermediate assertion allows to conclude:

$$\{true\}\ y := x\ ;(x := x + 1 \parallel x := x + 2)\ \{x = y + 3\}$$

Two generalized control structures are added to the core language. The statement **if** $\langle b_0 \to S_0\rangle; T_0 [] \langle b_1 \to S_1\rangle; T_1$ executes $S_0; T_0$ if b_0 holds and executes $S_1; T_1$ if b_1 holds, but the evaluation of b_i and subsequent execution of S_i are atomic. The statement **do** $\langle b_0 \to S_0\rangle; T_0 [] \langle b_1 \to S_1\rangle; T_1$ executes $S_0; T_0$ and starts over again if b_0 holds, executes $S_1; T_1$ and starts over again if b_1 holds, and blocks if neither b_0 nor b_1 holds, but the evaluation of b_i and subsequent execution of S_i are atomic.

Definition (Atomic Conditional and Repetitive Statements)**.** Let S_i be statements that do not contain repetitions, procedure calls, await statements, or parallel compositions and let T_i be arbitrary statements. Let C be an atomic statement that evaluates b_i, executes S_i accordingly, and sets integer c to the alternative executed, or blocks if all b_i are false:

$$C \quad = \quad \langle b_0 \vee b_1 \to \textbf{if}\ b_0 \to S_0\ ;\ c := 0\ []\ b_1 \to S_1\ ;\ c := 1\rangle$$

Then:

$$\begin{array}{lcl} \textbf{if}\ \langle b_0 \to S_0\rangle\ ;T_0 & = & \textbf{var}\ c : integer \\ \ [] \langle b_1 \to S_1\rangle\ ;T_1 & & \quad C\ ;\ \textbf{if}\ c = 0 \to T_0\ []\ c = 1 \to T_1 \end{array}$$

$$\begin{array}{lcl} \textbf{do}\ \langle b_0 \to S_0\rangle\ ;T_0 & = & \textbf{var}\ c : integer \\ \ [] \langle b_1 \to S_1\rangle\ ;T_1 & & \quad C\ ;\ \textbf{do}\ c = 0 \to T_0\ ;C\ []\ c = 1 \to T_1\ ;C \end{array}$$

The atomic repetitive statement blocks when all guards are false and starts over again when one guard holds and the corresponding statement is executed, meaning that it never terminates; the plain repetitive statement terminates when all guards are false.

Functions are used to model both arrays and fields of objects. If $g : X \to Y$ is a function and $f : X$ a value of the domain, the function application is written as $g(f)$ (when g is an array) or as $f.g$ (when g is a field and f is an object). If $e : Y$ is a value of the range, $g[f \leftarrow e]$ denotes the modification of g to be e at f and unchanged otherwise. Assignments to function elements and procedure calls that update function elements are formally defined by replacing the function with a modified one:

Definition (Function updates)**.** Let g be a variable of function type:

$$\begin{array}{lcl} g(f) := e & = & g := g[f \leftarrow e] \\ g(f) := m(e) & = & \mathbf{var}\, h \,;\, h := m(e) \,;\, g := g[f \leftarrow h] \end{array}$$

Using the *dot notation*, $g(f) := e$ is also written as $f.g := e$. Updates can be "nested" in the sense that $g(f)(h) := e$ and means $g := g[g(f) \leftarrow (g(f)[h \leftarrow e])]$ and, using the dot notation, can be written as $f.g(h) := e$.

The common *if-statement* and *while-statement* are defined in terms of the conditional and repetitive statements:

Definition (if-statement, while-statement)**.** Let b, b_i be Boolean expressions and S, S_i, T be statements:

$$\begin{array}{lcl} \mathbf{if}\, b\, \mathbf{then}\, S & = & \mathbf{if}\, b \to S \,[]\, \neg b \to \mathbf{skip} \\ \mathbf{if}\, b\, \mathbf{then}\, S\, \mathbf{else}\, T & = & \mathbf{if}\, b \to S \,[]\, \neg b \to T \\ \mathbf{if}\, b_0\, \mathbf{then}\, S_0\, \mathbf{elif}\, b_1\, \mathbf{then}\, S_1\, \mathbf{else}\, T & = & \\ \quad \mathbf{if}\, b_0 \to S_0 \,[]\, \neg b_0 \wedge b_1 \to S_1 \,[]\, \neg b_0 \wedge \neg b_1 \to T & & \\ \mathbf{while}\, b\, \mathbf{do}\, S & = & \mathbf{do}\, b \to S \end{array}$$

Finally, the atomic statement $\langle S \rangle$ abbreviates $\langle \mathit{true} \to S \rangle$.

3.4.2 Concurrent Objects

A class is defined by mapping each field to a global variable, each method to a procedure with an additional *this* parameter, and all actions together to one procedure that repeatedly executes one of the enabled actions. A program consists of the parallel composition of the actions of all allocated objects together with the main program. Following translation scheme makes method and action bodies atomic up to method calls. For this, an additional field, *lock*, is defined for each object: when a method is entered, an action is started, or a suspended method or action is restarted, *lock* is set to *true*. When a method or action terminates or execution leaves an object by calling

another object, *lock* is set to *false*. Let *Ref* be a set of potential object references. Abstractly, a Lime program consists of a number of class declarations of the form

Listing 3.1: Lime Class

```
class C
  var v: V
  init ()
    I
  method m0(u0: U0) → (w0: W0)
    when g0 do M0
  method m1(u1: U1) → (w1: W1)
    when g1 do M1
  action a0
    when h0 do A0
  action a1
    when h1 do A1
```

where g_i, h_j are Lime guards M_i, A_j are Lime statements. Lime guards are executable Boolean expressions that must be only over fields v; in methods, guards g_i can also be over value parameters u_i. Expressions in method and actions bodies are over fields v as well as global variables (that are used for I/O); expressions in method bodies M_i can additionally be over value parameters u_i and result parameters w_i. Lime statements are inductively defines as consisting of:

- $\overline{x} := \overline{e}$, the *multiple assignment*, where $\overline{x}$ may only contain fields v, global variables (that are used for I/O), and in methods bodies M_i also u_i, w_i;
- $S_0; S_1$, the *sequential composition*;
- **if** b **then** S, **if** b **then** S **else** T, **if** b_0 **then** S_0 **elif** b_1 **then** S_1 **else** T, the *if-statement*;
- **while** b **do** S, the *while-statement*;
- **var** $\overline{x} : X; S$, the *local variable declaration*;
- $\overline{x} := o.m(\overline{e})$, the *method call* of method m of object o with actual value parameters $\overline{e}$ and actual result parameters $\overline{x}$;
- $o :=$ **new** $C()$, the *object creation*.

The multiple assignment statement, sequential composition, if-statement, while-statement, and local variable declaration retain their meaning. If o is an object of class C, a call to method m is defined as calling procedure $C.m$ and the object creation is defined as calling procedure $C.init$, defined below. For every class C, a variable C is the set of object created from the class, as defined below. Classes are used as types, meaning that **var** $\overline{c} : C$ stands for **var** $\overline{c} : Ref$ and additionally implies that

$c \in C \cup \{nil\}$ is (implicitly) part of any annotation in the scope of C (which is ensured by the compiler).

Definition (Object creation, method call)**.** Suppose o is of class C:

$$\begin{array}{lcl} \overline{x} := o.m(\overline{e}) & = & this.lock := false\ ;\ \overline{x} := C.m(o, \overline{e})\ ; \\ & & \langle \neg this.lock \rightarrow this.lock := true \rangle \\ \overline{x} := \textbf{new}\ C() & = & \overline{x} := C.init() \end{array}$$

Definition (Classes)**.** Take the class C defined in Listing 3.1 as an example: Writing I', g_i', M_i', h_j', A_j' as a shorthand for the same with all occurrences of v replaced with $this.v$, class C is defined as:

$$\begin{array}{l} \textbf{var}\ \ C : set(Ref) := \{\} \\ \textbf{var}\ \ C.lock : Ref \rightarrow \textbf{bool} \\ \textbf{var}\ \ C.v : Ref \rightarrow V \\ \textbf{procedure}\ \ C.new() \rightarrow (this : Ref) \\ \quad \langle this :\notin C \cup \{nil\}\ ;\ C := C \cup \{this\}; this.lock := true \rangle;\ \ I';\ \ this.lock := false \\ \textbf{procedure}\ \ C.m_0(this : Ref,\ \ u_0 : U_0) \rightarrow (w_0 : W_0) \\ \quad \langle g_0 \wedge \neg this.lock \rightarrow this.lock := true \rangle;\ \ M_0';\ \ this.lock := false \\ \textbf{procedure}\ \ C.m_1(this : Ref,\ \ u_1 : U_1) \rightarrow (w_1 : W_1) \\ \quad \langle g_1 \wedge \neg this.lock \rightarrow this.lock := true \rangle;\ \ M_1';\ \ this.lock := false \\ \textbf{procedure}\ \ C.action(this : Ref) \\ \quad \textbf{do}\ \ \langle h_0 \wedge \neg this.lock \rightarrow this.lock := true \rangle;\ \ A_0';\ \ this.lock := false \\ \quad [\!]\ \ \langle h_1 \wedge \neg this.lock \rightarrow this.lock := true \rangle;\ \ A_1';\ \ this.lock := false \end{array}$$

A Lime program consists of a set of class declarations and a main program, which is abstractly a statement and which is specified as the initialization of a class with the name *Start*.

Definition (Concurrent program)**.** A behaviour of program with classes $C_0, C_1, Start$ is defined as the statement:

$$\begin{array}{l} (\|\ x : Ref \cdot x \in C_0 \rightarrow C_0.action(x)) \ \| \\ (\|\ x : Ref \cdot x \in C_1 \rightarrow C_1.action(x)) \ \| \\ (\textbf{var}\ \ s : Start; s := \textbf{new}\ \ Start()) \end{array}$$

A program starts to execute the main program, which can create objects with actions that continue to execute. The whole execution may carry on indefinitely or until the main program terminates and all actions are disabled, i.e. all procedures $C_i.action$ block. A program may operate on global variables through which its execution is observed.

3.5 Lime Examples

With these examples, we show how it is easier to develop concurrent programs in Lime compared to the other synchronization constructs, such as semaphores and channels. The programmer only needs to create the actions the objects have and does not need to worry about how those actions are scheduled in the background. All that scheduling work is left to the Lime runtime system. We illustrate the primary constructs of Lime by using various classic concurrent examples.

In this section, we first show the *semaphores* construct can be easily implemented in Lime and then present the solutions of *dining philosophers* problem by using Lime, *semaphores* and *channels*. Third, we discuss the solutions of *Reader-Writer* problem by using *semaphores* and Lime. In addition, we introduce three examples in Lime: *Delayed Doubler*, *priority Queue* and *Leaf-oriented Trees*.

3.5.1 Semaphores

Semaphores are a fundamental construct of a concurrent program. This example — Listing 3.2 — shows how a general semaphore can be easily implemented in Lime.

Listing 3.2: Semaphore Using Lime

```
class Semaphore
   var value: int
   init()
      value := 0
   method P()
      when value > 0 do
         value := value - 1
   method V()
      value := value + 1
```

The *Semaphore* class maintains the invariant $value \geq 0$. Method P is enabled if $value > 0$. Method P is disabled if $value \leq 0$. The atomicity policy is that "all methods and actions are atomic up to method calls". Hence all methods and the initialization of the class *Semaphore* are executed atomically. That is, P is suspended when *value* is 0.

Compared with the definition of semaphore we discussed in Section 2.1.2, the implementation of semaphore in Lime has been reduced to its essential elements. More importantly, Lime allows programmers to handle the concurrency as an implementation issue in the same way as the choice of an algorithm.

3.5.2 Dining Philosophers

The *dining philosophers* problem is a traditional problem in the history of concurrency created by Dijkstra (1987) as an exam question. This example — Listing 3.5 — shows

how to implement the dining philosophers problem in Lime. The eating process of each philosopher is implemented as an action, and one philosopher can eat only when the forks on both sides are picked up.

To compare with Lime, we also provide two implementations by using semaphores and channels, respectively, for the *dining philosophers* problem.

> **Dining Philosophers Problem.** Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a large bowl of spaghetti, and the table is laid with five forks. On feeling hungry, a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he has finished, he puts down both forks, and leaves the room. (Hoare, 1978)

Semaphores can be used to solve this dining philosopher problem (Andrews, 1991). There is one semaphore for each fork. First, a local two-phase prioritization scheme is used to ensure that neighbouring philosophers do not eat at the same time. However, it can result in deadlock because there is a circular waiting. To avoid deadlock, all the Philosophers in Listing 3.3, 3.4, 3.5 pick up first the lower-number fork and then the higher-number fork, rather than their left fork and then their right fork.

Listing 3.3: Dining Philosopher Solution Using Semaphores

```
var forks[1..5]: sem
process Philosopher[i: 1..4]
   while true do
      P(fork[i]); P(fork[i + 1])
      eating
      V(fork[i]); V(fork[i + 1])
      thinking
process Philosopher[5]
   while true do
      P(fork[1]); P(fork[5])
      eating
      V(fork[1]); V(fork[5])
      thinking
```

Channels can also be used to solve this dining philosophers problem. Five philosophers and five forks can be represented by processes. Each fork process listens to *pickup* and *putdown* channels. "<- **chan**" represents receiving a value from the channel while "**chan** <-" represents sending a value to the channel. In this case — Listing 3.4 — a philosopher would need to send messages to the left fork and right fork, respectively.

Listing 3.4: Dining Philosopher Solution Using Channels

```
var pickup[1..5]: chan
var putdown[1..5]: chan
process fork[i: 1..5]
   while true do
      select
         case <- pickup[i]:
            <- putdown[i]
process philosopher[i: 1..4]
   while true do
      pickup[i] <- i
      pickup[i + 1] <- i
      eating()
      putdown[i] <- i
      putdown[i + 1] <- i
      thinking()
process philosopher[5]
   while true do
      pickup[1] <- 5
      pickup[5] <- 5
      eating()
      putdown[1] <- 5
      putdown[5] <- 5
      thinking()
```

Here the monitor mechanism is used in Lime (Listing 3.5). Compared with the solutions by using semaphores and channels, although Lime uses the same strategy to avoid deadlock, Lime provides an unified design view for the programmers to solve the problem by introducing a guard *avail* to the *Fork* object. We add two guarded methods: *pickup* and *putdown* to the *Fork* object and two unguarded actions to the *Philosopher* object.

Listing 3.5: Dining Philosopher Solution Using Lime

```
class Fork
   var avail: bool
   init()
      avail := true
   method pickup()
      when avail do
         avail := false
   method putdown()
      when not avail do
         avail := true
class Philosopher
   var leftFork, rightFork: Fork
```

```
   var state: {thinking, eating}
   init(left, right: Fork)
      leftFork, rightFork, state := left, right, thinking
   action eat
      when state = thinking do
         leftFork.pickup()
         rightFork.pickup()
         state := eating
   action think
      when state = eating do
         state := thinking
         leftFork.putdown()
         rightFork.putdown()
class Start
   var n, i: int
   var philosophers: array of Philosopher
   var forks: array of Fork
   init()
      n := getArg(1)
      forks, philosophers := new Fork[n], new Philosopher[n]
      for i := 0 to n - 1 do
         forks[i] := new Fork()
      for i := 0 to n - 2 do
         philosophers[i] := new Philosopher(forks[i], forks[i+1])
      philosophers[n-1] := new Philosopher(forks[0], forks[n-1])
```

3.5.3 Reader-Writer

The following example shows how to use a monitors mechanism to implement the reader-writer problem (Courtois et al., 1971) in Lime. We also provide another pseudo-code implementation in Listing 3.6 by using semaphores.

Listing 3.6: Reader Writer Solution Using Semaphores

```
var n := 0: int
var r := R: sem
var mutexR := 1: sem
var rw := 1: sem
process Reader[i: 1..N]
   while true do
      P(mutexR)
         P(r)
         n := n + 1
         if n = 1 then
            P(rw)
```

```
        V(mutexR)
        read the data
        P(mutexR)
            n := n - 1
            if n = 0 then
                V(rw)
            V(r)
        V(mutexR)
process Writer[j: 1..N]
    while true do
        P(rw)
        write the data
        V(rw)
```

This example (Listing 3.7) needs to ensure that a resource is either accessed by up to R readers or a single writer. Field n is the number of readers who are currently reading the resource, R is the maximum number of readers who can read the resource simultaneously.

Compared with semaphores (Listing 3.6), Lime and monitors provide a simpler design view, but more importantly, Lime makes concurrency transparent to the programmers by using only one guard n.

Listing 3.7: Reader Writer Solution Using Lime

```
class ReaderWriter
    var n, N: int
    init(R: int)
        n, N := R, R
    method startRead()
        when n > 0 do
            n := n - 1
    method startWrite()
        when n = N do
            n := 0
    method endRead()
        n := n + 1
    method endWrite()
        n := N
```

The class initialization sets fields N and n to the maximum number of readers. The class maintains the invariant $0 \leq n \leq N$. Methods *startRead* and *startWrite* are enabled if their respective guard is true, otherwise disabled. If *rw* is an object created by *ReaderWriter*(N), then a typical access of the resource would be *rw.startRead; ...; rw.endRead* or *rw.startWrite; ...; rw.endWrite*. The atomicity policy is that "all methods and actions are atomic up to method calls". Hence all methods and the initialization of the class *ReaderWriter* are executed atomically, but the calls

rw.startRead and *rw.startWrite* may be suspended. We assume here that all fields are private to an object and all methods are public in Lime.

3.5.4 Delayed Doubler

The following example is the *Delayed Doubler* created by Sekerinski (2002):

> "Both the *Doubler* and *DelayedDoubler* classes allow an integer to be stored and its double to be retrieved. In *Doubler* the operation of doubling is performed when the number is stored. In *DelayedDoubler* instead a "background" action is enabled that perform[s] the doubling, allow[ing] a call to *store* to return quicker. The *retrieve* method needs to be suspended until the doubling occurs, which is controlled by the additional variable *d*" (Cui, 2009).

This example — Listing 3.8 — represents a universal pattern. For example, programmers have the illusion that writing data to a file happens instantly, but the data is cached in a buffer, and a background program starts to write the data to the hard disk. The same thing happens when programmers add a record to the database. In each case, this pattern aims to increase the responsiveness of the concurrent system which we discussed in the Section 1.1.1. This is the first example which contains a guarded action. This *DelayedDoubler* class maintains the invariant $(d = true) \Longrightarrow (y = 2 * u)$.

It is straightforward to implement this example by using asynchronous channels or futures. In principle, synchronous channels are capable of simulating the behaviour of asynchronous channels, however, introducing complexity. It is difficult to implement this example by using semaphores or monitors.

future *double*(*x*)

Listing 3.8: Delayed Doubler Example Using Lime (Cui, 2009)

```
class Doubler
   var x: int
   method store(u: int)
      x := 2 * u
   method retrieve(): int
      return x
class DelayedDoubler
   var y: int
   var d: bool
   init()
      y, d := 0, true
   method store(u: int)
```

```
      y, d := u, false
   method retrieve() : int
      when d do
         return y
   action double
      when not d do
         y, d := 2 * y, true
```

3.5.5 Priority Queue

The following example — Listing 3.9 — shows how to use Lime to implement a concurrent priority queue as described by Sekerinski (2003):

> A priority queue offers a method $add(e)$ for storing an positive integer e, a method *remove* for removing the least integer stored so far, and a method *empty* for testing whether the priority queue is empty. Elements are stored in field m in ascending order (duplicates are allowed). The priority queue starts with a sentinel node ($m = 0$). Field l points to the next node or is *nil* (last element of the queue). An element is added to the priority queue by either storing it in the current node if it is the last one (and creating a new last node), or by depositing it in field p of the current node and enabling an action (*doAdd*) that will move either the new element or the element of the current node one position down. The minimal element is removed by returning the element of the current node immediately and enabling an action (*doRemove*) that will move the element of the next node one position up, or set the l point to *nil* if the node becomes the last one. Field a represents the *doAdd* action while field r represents the *doRemove* action.

Like the delayed doubler example, this priority queue example also implements the *early return* mechanism. It is complicated to implement early return by using semaphores and monitors. The early return can be implemented in Lime by actions. For example, as shown in Figure 3.1, the second node is *enabled*, which needs to send 6 to the next node. At the same time, the *head* node is available to receive a new element. There is also a sentinel node at the end of the queue. As defined above, the return value of $q.remove()$ should be either a positive integer or 0, which means the priority queue is empty. The method call $l.add()$ in the action of *doAdd* means that the current node makes a call to the next object l. The lock of the current node is released before that call and regained on return.

Listing 3.9: Priority Queue Example Using Lime (Sekerinski, 2003)

```
class PriorityQueue
   var m,p: int
```

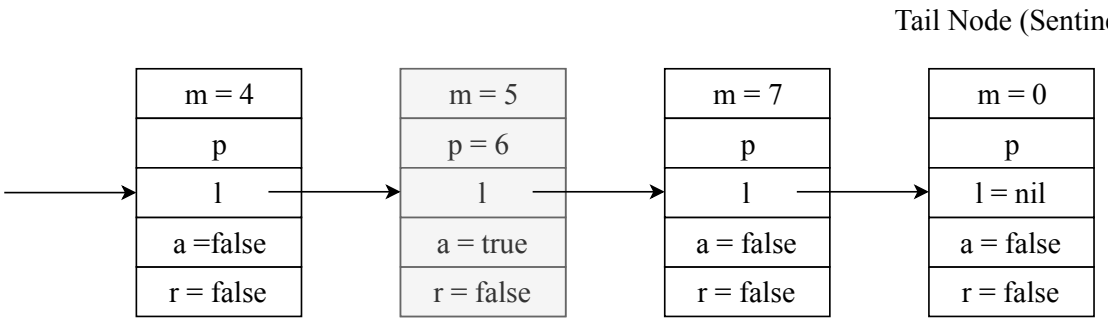

Figure 3.1: Priority Queue with Input of "4, 5, 7, 6"

```
var l: PriorityQueue
var a,r: bool
init()
   l,a,r,m := nil,false,false,0
method empty() : bool
   when not r do
      return l = nil
method add(int e)
   when not a and not r do
      if l = nil then
         m, l := e, new PriorityQueue()
      else
         p, a := e, true
method remove() : int
   when not a and not r do
      r := true
      return m
action doAdd
   when a do
      if m < p then
         l.add(p)
      else
         l.add(m)
         m := p
      a := false
action doRemove
   when r do
      if l = nil then
         r := false
         return
      elif l.empty() then
         l := nil
```

```
        else
            m := l.remove()
        r := false
```

3.5.6 Leaf-oriented Trees

The next example — Listing 3.10 — is about parallelizing operations on sets, which is implemented by using leaf-oriented trees (Sekerinski, 2003).

> In a leaf-oriented tree, the internal nodes contain only guides and the elements are stored in the leaves. Insertion either creates two new leaves [one is the original element and the other is the element to be stored] or only deposits an element in an internal node. Each node has an action that would eventually move the deposited element one level closer to its final position. This action needs to hold a lock only on the current node and one of its children. Thus insertions can proceed in parallel in different parts of the tree. The methods *add* and *has* are guarded in order to prevent possible overtaking.

For example, as shown in Figure 3.2, the nodes which contain $key = 3$ and $key = 5$ are enabled and will send the elements, $p = 2$ and $p = 8$, to their child nodes, respectively. At the same time, the root node is available to receive a new element. The final structure of this leaf-oriented tree is shown in Figure 3.3. For the method calls, for example $right.add(p)$, it means that the current parent node makes a method call to its *right* child node. The lock of the current node is released before that call and regained on return. The class *Node* maintains the local invariant `(left = nil )= (right = nil)` and the global invariant `(left ≠ nil) ⟹ (left.key ≤ key) ∧ (right.key > key)`.

Listing 3.10: Leaf-oriented Trees Example Using Lime

```
class Node
   var key, p: int
   var left, right: Node
   var a: bool
   init(x: int)
      key, left, right, a := x, nil, nil, false
   method add(x: int)
      when not a do
         if left != nil then a, p := true, x
         elif x < key then
            left,right, key := new Node(x), new Node(key), x
         elif x > key then
            left,right := new Node(key), new Node(x)
   method has(x: int): bool
```

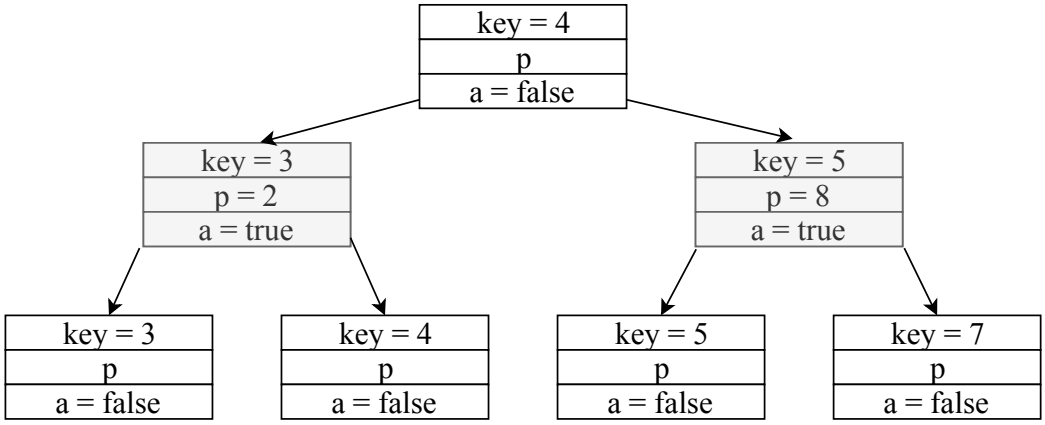

Figure 3.2: Leaf-oriented Tree Example With Input: "5,4,3,7,2,8", Intermediate State

```
      when not a do
         if left = nil then return x = key
         elif x <= key then return left.has(x)
         else return right.has(x)
   action addToChild
      when a do
         if p <= key then left.add(p)
         else right.add(p)
         a := false
```

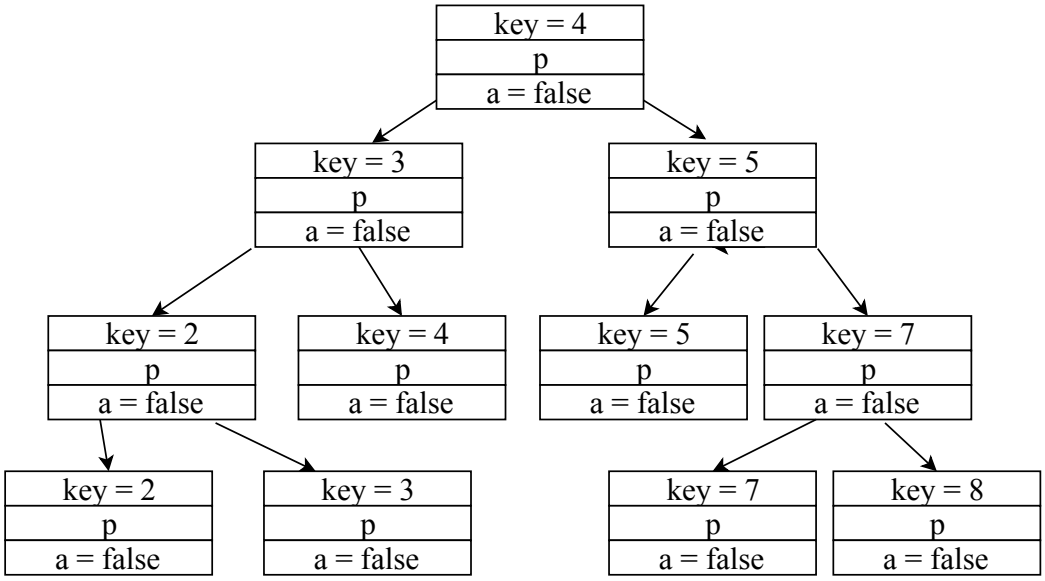

Figure 3.3: Leaf-oriented Tree Example With Input: "5,4,3,7,2,8", Final State

Chapter 4

Contributions to Lime

In this chapter, we first present the previous implementations of Lime and then discuss the problems addressed during the earlier implementations. Last, we summarize the contributions of this work.

4.1 Previous Implementations of Lime

In this section, we discuss the previous two implementations of Lime.

4.1.1 Implementation of Lime Using Monitors

Lou (2004) presents a design and implementation of a Lime compiler written in Java. The target language of this Lime compiler is Jasmin, a Java assembly language. The compilation is separated into two steps. First, the program is translated into an intermediate Lime program without actions and guarded methods. Actions and guarded methods are translated into unguarded methods by using monitors at the second step.

During program execution, there exist several worker threads that execute the methods and actions of the objects. There is a constraint that one thread can lock at most one object at a time. Each worker thread locks an object when working on it. If an object makes a call to another object, the lock is released before that call and regained on return, as each worker thread can lock at most one object at a time

When a worker thread executes a guarded method of an object, the thread first evaluates the guard of the method. The thread executes the method if the guard holds; otherwise the thread waits. The suspended threads should wake up to re-evaluate the guards when a thread exits from the same object. This is because the execution of the method or action of the object may affect the guards of other guarded methods or actions. The overhead of context switch for the Java threads becomes significant when the number of the objects exceeds the number of the available cores.

4.1.2 Implementation of Lime Using Condition Variables

Cui (2009) implements a Lime compiler written in Pascal. The target language of this Lime compiler is assembly code. The generated code makes calls to the Pthread library. The guards of the object are translated into Pthread's condition variables. Each active object is associated with an object thread at runtime. This object thread continuously selects enabled actions to run, and therefore never terminates. The entire program terminates only if all object threads are blocked on a condition such that it can no longer find an enabled action to run.

When a thread tries to invoke a guarded method, it must acquire the object's lock and then wait on the guard if the guard does not hold. After the execution of the method, the other suspended threads should wake up to re-evaluate the guard.

The main thread examines the termination condition by repetitively checking a global counter shared among the threads. This counter is initialized to the number of the threads and is decreased by one if one thread cannot find any enabled actions to execute. The main thread checks this counter by busy waiting and terminates when this counter reaches zero.

Same as Lou's implementation, Cui's implementation also suffers from the significant overhead of the context switches. Besides, their implementations may run out of stack space if there are too many threads created. Therefore, the stack mechanism needs to be improved for Lime.

4.2 The Addressed Problems

4.2.1 Threads

For the previous Lime implementations, mapping active objects to threads, either Java threads or Pthreads, can easily lead to programs with thousands of threads. The threads getting stuck during the method calls can lead to a tremendous number of context switches. Context switches introduce a significant amount of overhead to the overall program execution time when the number of the active objects exceeds the number of cores. Both Java threads and Pthreads are mapped to OS threads. During the context switch, the scheduler needs to save and restore all registers belonging to the thread. For instance, there are 16 registers on the x86 architecture and 32 registers on the x86-64 architecture. The context switch overhead becomes a performance concern.

4.2.2 Stack Mechanisms

The previous implementation work on Lime showed that an improved call stack mechanism is needed for Lime. Every active object being concurrent can easily lead to programs with thousands of threads. On the one hand, if stack sizes are set large enough, virtual address space becomes quickly exhausted as the number of the threads

increases. On the other hand, if stack sizes are "intentionally set low, stack-gobbling features, most notably recursion, are disabled as a workaround" (Moore-Oliva et al., 2014). In a word, the requirements for the new stack mechanism are (1) the default stack size should be small and (2) the stack can grow and shrink on demand.

"To discover or identify an efficient multi-threaded call stack mechanism that works as well and as transparently as the call stack mechanism for single-threaded processes", Moore-Oliva (2010) implements a C-like compiler in Java. This compiler directly generates assembly code for different stack mechanisms. My work in this part is related to improve the performance of Guard-Page stack mechanism. Segmented stacks mechanisms are examined based on this C-like compiler, and the comparison results for stack mechanisms are general and independent of Lime.

4.2.3 Guard Implementations

Lime can be implemented at different levels. At one extreme, a straightforward implementation approach is to achieve all of the concepts on top of the existing object-oriented programming languages, like Java. The synchronization mechanism, such as monitors or semaphores, can be used to implement the guards. Actions are implemented as daemon threads. In this case, the efficiency of the model highly depends on the underlying programming languages. At the other extreme, these models can be directly implemented on top of the operating systems. Instead of using synchronization mechanisms provided by the library, hardware instructions can be used to implement the locks for each object and guards for both methods and actions. The developers have to build the runtime system for the models. That is, it is possible to improve the efficiency of these models. Lime can be implemented by using any approach lying at or anywhere between these extremes.

4.3 Contributions of This Work

The most significant difference between Lime and the traditional object-oriented languages is that actions and guarded methods are added to the system. According to the theory of action systems, active objects can be executed concurrently. Therefore, each object has a private lock.

4.3.1 Lime Compiler

Our goal is to develop an efficient implementation of Lime. The compiler should focus on the key features, such as the guards implementation. The current implementation of Lime compiler leaves other features for future work. Therefore, the Lime compiler should be simple, easy to understand and maintain. Antlr4 provides not only a suite of tools to make writing grammars easy but also proper error messages to make debugging grammars easy (Parr, 2004). We use Antlr4 to generate the scanner and

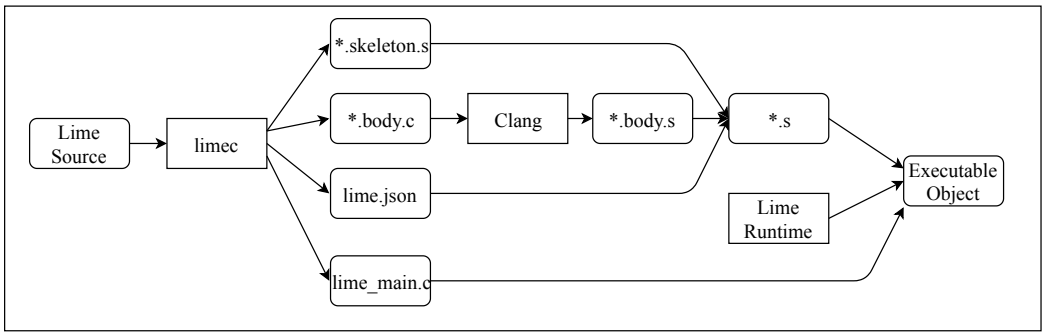

Figure 4.1: Lime Compiler

parser for Lime source code. Then, the compiler uses StringTemplate (Parr, 2012) and LLVM (a collection of modular and reusable compiler and toolchain technologies) to generate 32-bit NASM Assembly. The source code of the Lime compiler can be found in the folder: `./Limec/Compiler/Lime4` of the gitlab project: (https://gitlab.cas.mcmaster.ca/yaos4/thesis_code.git).

The Lime compiler accepts one single file as input which could contain one or more classes. The *Start* class in Lime is similar to the main function in C. As shown in Figure 4.1, each class is translated into two files: *classname.skeleton.s* and *classname.body.c*. *Classname.skeleton.s* contains the skeleton assembly code for the class, the guard evaluation code for methods and actions and the initialization code for the object. *Classname.body.c* contains the methods and actions body (without guards) of the class. The translation scheme is implemented in the skeleton assembly code. The Lime compiler takes advantage of LLVM's back-end such as optimizations.

4.3.2 Lime Runtime System

As we have discussed before, because the efficiency of Lime model is our priority goal, the approach we choose to implement Lime is closer to the operating system level, rather than the programming language level. In Lime, an efficient implementation of private locks and guards is needed. Most programming languages do provide primitives for conditional synchronization, such as semaphore and monitors. It is inefficient to use synchronization primitives, such as *wait* and *signal*, to implement the guards for each object. The results from the previous research show the performance decreases dramatically when the number of objects exceeds a threshold.

The private lock for each object in Lime should be implemented by hardware instructions, such as *compare-and-swap* instructions. The usage of the lock function provided by underlying programming languages also yields an additional overhead. The scheduler in Lime continues to search for the next available object if the current object is locked by the other worker thread, rather than spinning.

Active objects in Lime are mapped to coroutines, rather than threads. A coroutine

can suspend when it reaches the suspend point and return to the caller. A suspended coroutine can resume the execution from the suspend point. Thus, coroutines are the right primitives for active objects, as coroutines can yield execution when the guards are disabled and resume execution when the guards become true. More importantly, coroutines are lightweight compared to threads and processes in operating systems. Coroutines can use a dynamically allocated segmented stack mechanism, which starts with a small size (4KB) and grows on demand. This allows the system to create thousands of coroutines without consuming all available RAM.

The assembly language used in this work is 32-bit x86 instructions compatible with the open source assembler NASM. The architecture consists of eight 32-bit general registers. The EBP register is reserved by Lime runtime system for the pointer which points to the currently executing active object. Method arguments in Lime are passed on the stack, and the return value is kept in the EAX register. For the other calling conventions which pass arguments in registers, such as ARM (A32) and x86-64, extra space is needed to allocate for register-based arguments on the stack during the context switches.

In Lime, coroutines are scheduled by the Lime runtime system using an efficient cooperative scheduler. The context switch of coroutines is efficient because coroutines only switch at well-defined points and only three registers (EBP, ESP, EIP) are reserved on the stack. Lime coroutines use segmented stacks, which start with a small size and can grow and shrink on demand. The segmented stack mechanism needs to insert checkpoints to determine whether a stack overflow is about to happen or not. The overhead of these checkpoints can be accumulated, especially in loops and deep recursive calls. In Lime, we can avoid this overhead by modifying the calling convention.

The construction of Lime could be a significant contribution to the area of concurrent object-oriented programming. The development of Lime greatly enhances our understanding that an object is a natural "unit" of concurrency. The contribution is split into the following parts:

- The comparison of stack mechanisms: Compared with the traditional stack mechanisms, a dynamic stack mechanism can eliminate the overhead for stack checks during the runtime by modifying the calling convention. The source code and tested benchmarks can be obtained from `./Limec/LimeCC`. The comparison of stack mechanisms is discussed in Chapter 5.

- The guards implementation: Instead of using existing synchronization primitives to implement guards, we introduce a simple but efficient approach for the guard's implementation. In Lime, the method call could "get stuck" when the guard does not hold. Thus, the coroutine can yield the execution. The scheduler will choose the next coroutine to execute. The implementation of guards is discussed in Chapter 6.

- Lime runtime system: At runtime, a global active object queue is maintained. Each worker thread has a local active object pool which contains the active objects which belong to the worker thread. We also introduce a worker thread pool which can dynamically create and destroy worker threads according to the working load. The source code and tested benchmarks can be obtained from `./Limec/Runtime`. The implementation of the Lime runtime system is discussed in Section 6.3.3.

Chapter 5

Stack Mechanisms

In this chapter, we first introduce the background of current stack mechanisms. Then, we categorize and discuss existing and proposed multi-threaded stack mechanisms. Third, we elaborate on the implementation of Lime's stack mechanism, which we call the Guard-Page stack mechanism. Lastly, we compare the performance with the other stack mechanisms. All the work discussed in this chapter is based on the author's paper (Moore-Oliva et al., 2014), which stems from Moore-Oliva (2010). Some of the text in this Chapter is taken from (Moore-Oliva et al., 2014). My contributions to the joint paper are as follows:

- Found the reason why the previous work did not get positive results: it compared the performance mainly between *malloc* and *mprotect*. The stack extension runtime difference is minified. The Guarded-Page stacks should be pre-allocated and reused.
- Improved the performance of the Guard-Page mechanism;
- Extended the benchmarks for different stack usages.

5.1 Introduction

The traditional stack mechanism — where the stack and heap grow from opposite sides — is often taken for granted because it provides a simple but efficient way to keep track of variables and control flow. For a single-threaded process, there is no reason to use anything but the traditional call stack mechanism. In fact, with virtual address space being so abundant, operating systems take the strategy of allocating a "large enough" stack and do not extend it on stack overflow. For example, Turing Plus (Holt and Cordy, 1985) uses a fixed size stack and a controlled abort (with a system exception number) occurs when the stack overflows.

Since 2005, the trend for software development has been towards "more concurrency". The fixed size "large enough" stack mechanism would exhaust the virtual address space when the number of the threads increases.

For a 64-bit system, the virtual address space is 16 exabytes (EB). However, it does not allow the whole virtual address space to be used because of the overhead from the address translation. So Windows for x64 uses just 8 TB of user space (Microsoft, 2017). That is, we can only create at most 8192 threads with a "large enough" stack (1 GB) on Windows for x64 (we do not consider that the page tables need to take up RAM as well). This work advocates a programming style in which programmers do not need to be concerned about thread creation: rather every object is conceptually concurrent. This allows an implementation to exploit all available cores and thus scale the performance of programs with the number of processor cores. For example, in a molecular simulation program, every molecule can be an object, leading to millions of threads (Weber et al., 2014). Programmers should not need to map molecules to a smaller number of worker threads (Lea, 2000), which can run thousands of coroutines. In addition, 32-bit processors still play a dominant role in embedded systems, according to a study of embedded markets presented by EETimes and Embedded (2019).

In Lime, every object being concurrent in principle can quickly lead to programs with thousands of coroutines. Each coroutine requires a stack. That is, Lime needs to support thousands of stacks. On the one hand, if stack sizes are set large enough, virtual address space becomes quickly exhausted as the number of the coroutines increases. On the other hand, if "stack sizes were intentionally set low, stack-gobbling features, most notably recursion, were disabled as a workaround" to avoid the stack overflow (Moore-Oliva et al., 2014). Take the priority queue in Section 6.4.1 as an example, and one program can contain as many as ten thousand coroutines. Lime needs a stack mechanism which can run thousands of coroutines with a small stack growing and shrinking on demand.

5.2 Related Work

This section categorizes and discussed existing and proposed stack mechanisms. First, we briefly discuss the existing single-threaded stack mechanism. Second, we compare the shared stack mechanisms which allow the threads to share the stack space. Third, we discuss cactus stack mechanisms that use the cactus stack data structure to link multiple stack chunks into a single stack.

5.2.1 Single-Threaded Call Stack

A single-threaded program has one stack and one heap memory region. The stack grows from high to low while the heap grows from low to high. This memory layout allows the stack and the heap to share the available memory (heap fragmentation issue aside). There is a potential that the heap and the stack can use up all the available memory. Due to this, operating systems such as Solaris, Linux, Windows and Unix (including iOS and Android) set stack space to a fixed "large enough" size.

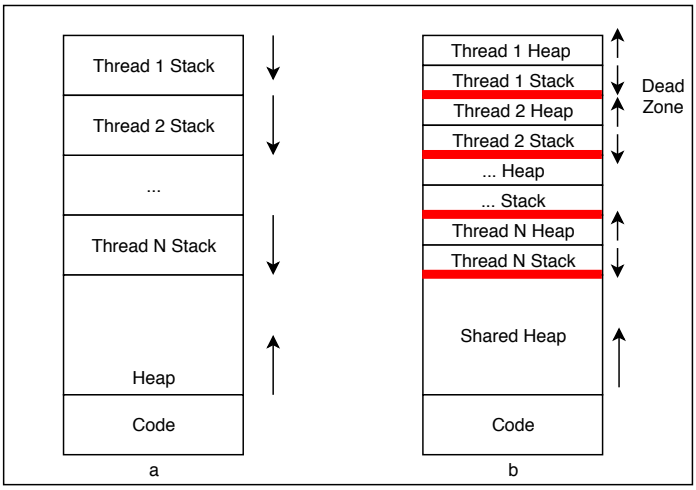

Figure 5.1: Single-threaded Memory Layout Extended to Multiple Threads: (a) With Single Shared Heap and (b) With Multiple Heaps to Reduce Heap Contention.

In this case, a segment fault should be generated when the stack "collides" with the heap (Sun Microsystems, 2008; Microsoft, 2013; Linux, 2012).

In operating systems, every process has two stacks:

- *User-mode stack*: is used to keep temporary data (local variables of the methods) in the user level.
- *Kernel-mode stack*: is used to store temporary data for the kernel. Every process has a kernel stack that can only be accessed in the kernel level.

For the user-mode stack, the multi-threaded stack mechanism used by *Solaris* (Sun Microsystems, 2008) has become the standard stack mechanism for modern operating systems. Each thread reserves a fixed size stack of the virtual address space. By default, a "large-enough" stack (typically 2 MB) is allocated for each thread during thread creation. A *red-zone* is used to detect stack overflows. This *red-zone* is a page of memory without read and write permissions and appended to a thread's stack. Any access to the *red-zone* will cause a memory fault. It could be identified at the compile time if the stack frame for a function is larger than the *red-zone*. *Windows* allocates by default 1 MB to each thread. *MacOS* allocates by default 8 MB to the *main thread* and 512 KB to *secondary threads*; *iOS* allocates by default 1 MB to the main thread and also 512 KB to the secondary thready. *Android* has two separate stacks: one for native code and one for Java code. The default Java stack size is 32 KB, while the native stack size is 1 MB.

Concurrent Oberon (Lalis and Sanders, 1994) uses a fixed size stack mechanism on the heap. A checkpoint is inserted in the prologue of every function and stack overflow can be detected before it occurs. Although this check increases runtime overhead, it

does provide a solution for the system which does not have an MMU. The stack can be recycled once the thread terminates.

US patent 7,477,829 (Wilding and Wood, 2008), depicted in Figure 5.1 (b), attempts to reduce heap contention. Unlike the traditional stack/heap memory layout for the single-threaded program, the stack and the heap start from the same base address and grow in opposite directions. Overflow is detected via the use of *dead-zone*, same as *red-zone* discussed previously. However, the patent does not specify how to calculate the initial base address of the stack and the heap. In theory, this patent has the ability to custom the stack and heap size for each thread at the thread creation.

All of the above stack mechanisms have the limitation that stack space cannot be shared among threads, and each thread needs to have a "large enough" stack to handle the worst-case usage. However, this can lead to the false "run out of space" errors even if there is plenty of unused space available in the other threads' stacks.

5.2.2 Shared Call Stacks

Hybrid stack sharing (Wong and Dagevill, 1994) attempts to share a fixed number m of stacks evenly among n threads, where $n \leq m$. On a context switch, if all stacks are used, the used portion of the current thread's stack is copied to the heap memory. The scheduled thread's stack is copied in. Hybrid stack sharing assumes the stack is large enough, and stack overflow would never occur. This stack mechanism improves memory usage by allowing multiple threads to share one stack. However, it introduces runtime overhead during a context switch when all the stacks are used.

Multitask stack sharing (Middha et al., 2008) provides a multi-threaded stack mechanism for embedded systems. Because the address space is limited, each thread begins with a smaller stack size (compared with Solaris' stack size). Stack overflow is detected at runtime in the prologue of each function. Each thread reserves one page at the end of the stack for the overflowing thread's stack. In this case, part of the stack can be shared among all threads, which improve the stack memory sharing. However, the program could "run out" of the stack memory when there is plenty of unused memory available.

5.2.3 Cactus Stacks

Stackless Python (Tismer, 2000) moves stack data into the interpreter frames, which also contain code. In this case, the stack is allocated on the heap, and there is no limit on stack size. However, it introduces runtime overhead because every function call requires a heap allocation.

Thread segment stacks (Pizka, 1999) is a multi-threaded stack implementation for GCC. Each thread has a contiguous stack space, and stack overflow is detected at runtime in the prologue and epilogue of every function. When stack overflow occurs, a *linear extension* is performed first. The *linear extension* attempts to append a new

page contiguously to the current virtual address. If the following virtual address is unavailable, a new stack segment is allocated and linked. This stack mechanism does not have the false "out of stack space" errors and allows the thread to begin with a smaller stack size. However, it introduces the runtime overhead for every function call (there are eight instructions inserted to check the stack overflow for every function call).

Google's Go implements a segment stack mechanism by initially allocating 4 KB for each thread (goroutine) and having the linker inserts a preamble at each procedure (function) call. When overflow is detected, a new stack page is allocated and linked to the previous stack page. Because of the "hot split" problem (Cheney, 2014), Go programming language replaces this segment stack mechanism by a stack copying mechanism, which creates the new segment stack by doubling the size and copies the old segment stack into it. To support the stack copying, Go implements the escape analysis by tracing the flow of the input and output values from the call graph (Google, 2018a).

IBM's z/OS implements a segmented stack mechanism (IBM, 2010) for passing arguments to external routines, which replaces the explicit inline check for overflow with storage protect mechanism that detects stores past the end of the stack segment. A guard page is appended to each stack segment. The guard page triggers an exception, which causes a new stack segment allocation. To make the stack appear contiguous to the application, all fields on the stack for the caller (including the arguments) should be copied into the new stack segment.

Capriccio (von Behren et al., 2003) improves the runtime overhead of Thread Segment Stacks by analyzing the call graph of a program. Based on the call graph, Capriccio calculates stack usage and inserts the stack checkpoint at compile-time, rather than inserting checkpoints to every function. For example, there are two consecutive function calls, *foo16* and *foo32*, requiring 16 bytes and 32 bytes of stack space respectively. There is only one checkpoint for 48 bytes inserted in the prologue of the function *foo16*. For the external library function calls, if the maximum stack usage is unknown, a default "large enough" stack chunk is allocated. A checkpoint is inserted in the prologue of every recursive function. Capriccio removes the false "out of stack" errors and minimizes the overhead of inline stack checkpoints by analyzing the call graph.

5.2.4 Stack Mechanisms Summary

Table 5.2.4 contains a summary of various programming languages' stack mechanisms. The meaning of the column is:

- Stack Type — This refers to whether the stack can grow or not.
  Dynamic. The stack grows automatically as needed during the runtime.
  Fixed Size. The stack has a fixed size and can not grow.

- Thread Type - As discussed in Section 1.1.1, the concept of a thread in programming languages is mapped to either the operating system thread or the thread library.
  Native Threads. Native threads mean that each thread (or coroutine) is directly mapped to a thread generated at the operating system level.
  User-Space. User-Space means that each coroutine created by a programming language is manipulated in the user space.
  Java Threads. JVM decides how to map a Java thread to the underlying operating system.
  Pthreads. Pthreads refers to the Pthread implementation on different operating systems.

Programming Language	Version	Stack Type	Thread Type
Go Goroutines	1.1	Dynamic, Stack Copying	User-Space
Erlang Processes	20.2	Dynamic, Garbage Collector	User-Space
Scala Coroutines	2.12.4	Fixed Size	User-Space
Lua Corotines	5.3.4	Fixed Size	User-Space
Kotlin Coroutines	1.2	Fixed Size	User-Space
Rust Threads	1.24.0	Fixed Size	Native Threads
Haskell Threads	2010	Fixed Size	Native Threads
Kotlin Threads	1.2	Fixed Size	Java Threads
Scala Threads	2.12.4	Fixed Size	Java Threads
Scheme Threads	4	Fixed Size	Pthreads
Lua Threads	5.3.4	Fixed Size	Native Threads
JavaScript	V8	Fixed Size	Single Threads

Table 5.1: Stack Mechanisms Summary for Programming Languages

5.3 Experimental Setup

Our goal is to discover an efficient stack mechanism that works for both multi-threaded and single-threaded processes. Therefore, scalability is a requirement that must be met. In addition, there are some other requirements:

- The stack mechanism should support concurrent multithreading;
- The stack memory could be shared among threads;
- Stack data should be referenceable. That is, the stack data can not be moved around. Because the current implementation of Lime uses pointers, moving stack data would introduce significant complexity to the runtime system.

The following stack mechanisms do not meet the above requirements:

- All stack mechanisms from Section 5.2.1 that lack dynamic sharing of stack memory among threads. With these mechanisms, each thread is assigned with an exclusive, fixed-size stack space.
- Hybrid stack sharing uses a fixed number of fixed-size stacks for all the running threads. However, the overhead of context switching would be significant for a large number of threads.
- Multitask stack sharing was designed for embedded systems and did not support concurrent multi-threading. Extending this stack mechanism to support concurrent multi-threading would require synchronization to avoid race conditions between threads.
- Go's stack copying mechanism relies on the escape analysis, provided by Go's runtime system that moves stack data around.

Capriccio is the only stack mechanism that meets the criteria. Capriccio, which we call the Look-Ahead stack mechanism, can predict stack overflow at compile time.

5.4 Moore-Oliva's Lime Calling Convention

Instead of modifying the open-source compiler frameworks, such as GCC and LLVM, Moore-Oliva (2010) built a C-like Lime compiler and runtime system from scratch for the following two reasons: "First, it saves time and can avoid unforeseen complications resulting from modifying an existing complicated code-base. Second, it was unknown if some optimizations relied on a contiguous stack frame".

However, how to implement the Lime compiler in LLVM has always been one of our goals. LLVM has features that are very attractive for developing a Lime compiler: such as portable code generation, reusable standard optimizations, and multiple backends, including ARM, x86 and PowerPC processors.

Inspired by the XRay project (Berris et al., 2016), we found that the Lime calling convention can be implemented by modifying LLVM's *cdecl* calling convention. First, pseudo instructions are inserted during the LLVM's *MachineFunction* pass. Second, the pseudo instructions are replaced with actual instructions when LLVM emits the instructions. Because Lime's calling convention changes the layout of the stack frame, we have to modify the LLVM source code rather than utilize the public interfaces to implement the Lime calling convention.

We first give a brief overview of the Lime calling convention, the structure in detail will be discussed in the following section:

- The callee cleans up the arguments from the stack.

- Any procedure call can trash any register (except EBP, ESP and EIP registers).
- The first instruction for a procedure call is to store the return address on the stack.
- The return address is stored on the bottom of stack frame.

5.5 Implemented Stack Mechanisms

In this section, we first introduce two traditional fixed-size stack mechanisms. Then, we discuss a straightforward segmented stack mechanism: each stack frame is allocated on the heap. Lastly, we present the LookAhead stack mechanism and Guard-Page stack mechanism. All these stack mechanisms are implemented in a C-like Lime compiler to compare performance.

5.5.1 Traditional Fixed-Size Stack with "Caller-cleanup"

This call stack mechanism does not meet the criteria outlined in Section 5.3, but it is closer to the LookAhead stack mechanism discussed later on and allows for a better comparison of the measurements.

None of the stack mechanisms discussed in this section are restricted to Intel X86. They can be implemented on other architectures, such as ARM. For example, this "caller-cleanup" mechanism can be implemented on ARMv7 as shown in the Listing 5.2 and 5.4 .

Caller Instructions The caller routines for this stack mechanism implement the *cdecl* calling convention. The caller is responsible for pushing arguments to the stack, as well as cleaning the stack on procedure exit by adding *args_size* to the stack pointer. The ESP (or SP on ARM) register points to the top of the stack.

Listing 5.1: "Caller-cleanup" Caller Instructions on X86

```
PUSH arg1
...
PUSH argn
CALL callee_name
ADD ESP, args_size
```

Listing 5.2: "Caller-cleanup" Caller Instructions on ARM

```
PUSH {r0-r3}
BX callee_name
ADD SP, #args_size
```

Callee Instructions The callee ensures that the stack pointer has the same value on return from the function call as it had on entry.

Listing 5.3: "Caller-cleanup" Callee Instructions on X86

```
callee_name:
          SUB ESP, callee_stack_size
          ... #Body of procedure
          ADD ESP, callee_stack_size
          RET
```

Listing 5.4: "Caller-cleanup" Callee Instructions on ARM

```
callee_name:
          PUSH {r4, LR}
          SUB SP, #callee_stack_size
          ...
          ADD SP, #callee_stack_size
          POP {r4, PC}
```

5.5.2 Traditional Fixed-Size Stack with "Callee-cleanup"

This mechanism also does not meet the criteria outlines earlier, but it is closer to the Guard-Page stack mechanism discussed later on and allows for a better comparison of the measurements.

Caller Instructions The caller pushes arguments to the stack, but does not clean the stack on function exit.

Listing 5.5: "Callee-cleanup" Caller Instructions

```
          PUSH return_address
          PUSH arg1
          ...
          PUSH argn
          JMP callee_name
```

Callee Instructions The callee ensures that the stack pointer has the same value on return as before the caller pushed the arguments on the stack.

Listing 5.6: "Callee-cleanup" Callee Instructions

```
callee_name:
           SUB ESP, callee_stack_size
           ... #Body of procedure
           ADD ESP, callee_stack_size + arg_size
           JMP [ESP]
```

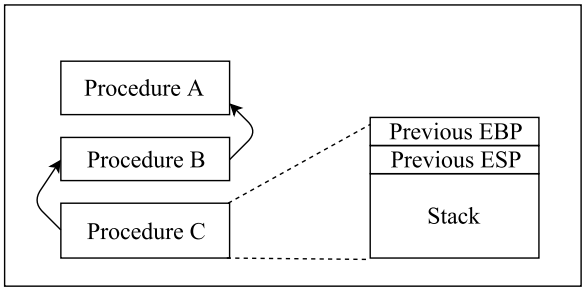

Figure 5.2: Per Procedure Heap Allocation Call Stack

5.5.3 Per Procedure "Heap" Allocation

As shown in Figure 5.2, a straightforward segmented stack mechanism allows each function invocation to have its stack frame, just "large enough" to hold the callee's stack frame as well as a pointer to the caller's stack frame. The stack frames are structured as a linked list allocated on the heap. The caller first allocates a new stack frame, then pushes the arguments on the new stack frame (while referring to its stack frame), calls the callee, and finally deallocates the stack frame. Allocation is done by calling *malloc*, which requires its own stack space. Neither the current nor the new stack frame can be used for that; hence a per-thread stack region is reserved for this purpose.

5.5.4 LookAhead Stack Mechanism

LookAhead stack mechanism is structured as a linked list of *stack chunks*. Unlike the stack mechanism we discussed in Section 5.5.3, where each function has a region of memory dynamically allocated containing just one stack frame, one stack chunk may contain many stack frames, as depicted in Figure 5.3. When a function call would cause an overflow, a new stack chunk is created and linked. The EBP register is reserved for pointing to the current stack chunk. The stack overflow is detected by using Capriccio's (von Behren et al., 2003) mechanism.

Caller Instructions The instructions discussed here are generated when the function call needs a checkpoint. When the function call does not need a checkpoint, the caller instructions are the same as to the code detailed in Section 5.5.1.

Listing 5.7: LookAhead Caller Instructions

```
MOV EAX, ESP
MOV EDX, ESP
ADD EDX, (STACK_CHUNK_SIZE
     -LONGEST_PATH(callee_name)-16)
CMP EDX, EBP
```

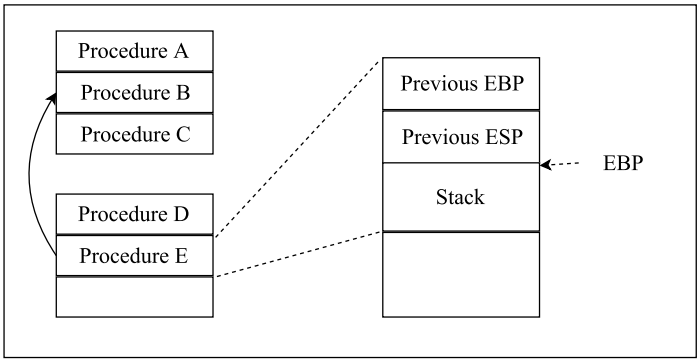

Figure 5.3: Stack Chunk for Look-Ahead Overflow Detection

```
      JGE label_1
      CALL STAMEX_OVERFLOW_HANDLER
label_1:
      PUSH arg_1
      ...
      PUSH arg_n
      CALL callee_name
      ADD ESP, args_size
      CMP EBP, ESP
      JNE label_2
      CALL STAMEX_UNDERFLOW_HANDLER
label_2: ...
```

The overflow handler allocates a new stack chunk by calling *malloc*, saves the previous values of EBP and ESP, and reserves a new thread stack. Since a subsequent call to *malloc* (and *free*) requires its own stack space, the thread stack needs to be reserved for this purpose. The stack pointer ESP is set so that, on return from the handler, the caller can push all the parameters on the stack in the possibly newly allocated chunk. The underflow handler restores the previous stack chunk and frees the current stack chunk.

Callee Instructions The callee ensures that the stack pointer points to the same location as it pointed on entry. The instructions are identical to the instructions detailed in Listing 5.6.

5.5.5 Guard-Page Stack Mechanism

Guard-Page stack mechanism is structured as a linked list of stack chunks, as depicted in Figure 5.3. On overflow, a new stack chunk is created and linked. The caller instructions are modified to ensure that "the deepest region of memory that the

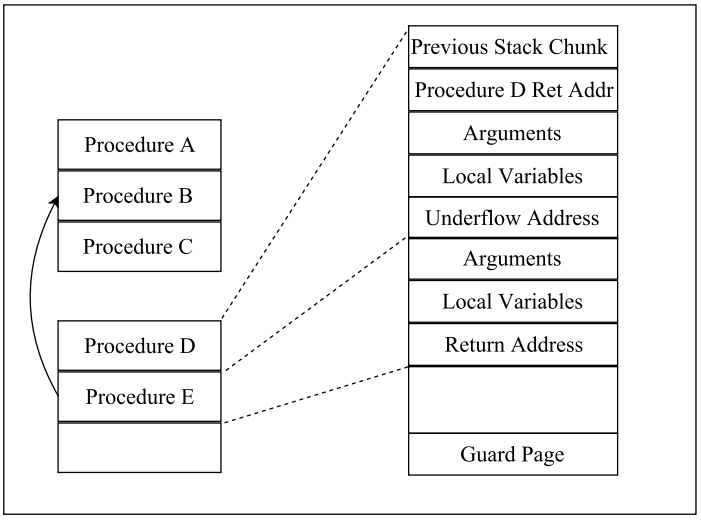

Figure 5.4: Stack Chunk for Guard-Page Overflow Detection

callee will use is accessed first". If the accessed memory is beyond the current stack chunk, the SIGSEGV signal is triggered by the guard page, which is appended to the current stack chunk. Then, the stack extension is performed for the thread. The current Lime implementation assumes that the stack frame for a function is always smaller than the guard page. To support the stack underflow, the first procedure's return address is replaced with the stack underflow procedure's address (Procedure D Ret Addr and Underflow Address in Figure 5.4). The following procedures store their return address to the bottom of the stack frame (Procedure E's Return Address in Figure 5.4). The underflow procedure is responsible for cleaning up the current stack chunk, restoring the previous stack chunk and executing the program.

Caller Instructions Lime stack mechanism modifies the stack frame's layout by moving the return address to the end of the stack frame. The caller instructions discussed in Listing 5.4 is used for invoking Lime procedures. For the external library functions, a trampoline procedure is needed.

To check if there is enough stack space left for the next procedure call, a store instruction which writes the procedure's return address (EDX register) to the stack is performed. If the store instruction fails, a SIGSEGV is generated, and the signal handler is responsible for creating a new stack chunk, loading the underflow procedure's address to the EDX register and executing the store instruction (executed before) again. Also, the underflow procedure's address is replaced with the next procedure's return address.

Listing 5.8: Guard-Page Caller Instructions

```
MOV EAX, ESP
```

```
            MOV EDX, return_label
            MOV [ESP-callee_stack_size], EDX
            PUSH arg1
            ...
            PUSH argn
            JMP callee_name
return_label: ...
```

Callee Instructions The callee ensures that the entire stack frame is cleaned up before returning. Compared to the standard C calling convention, Lime's callee needs to execute the underflow procedure before returning to the previous procedure.

Listing 5.9: Guard-Page Callee Instructions

```
callee_name:
            SUB ESP, callee_stack_size
            ... #Body of procedure
            ADD ESP, callee_stack_size+arg_size
            JMP [ESP-(callee_stack_size+arg_size)]
```

Stack Overflow and Underflow When a SIGSEGV is generated, the signal handler will first allocate a memory aligned chunk by calling *memalign*, and then append a guard page to the stack chunk, ensuring that any accesses will cause a SIGSEGV. As the *cdecl* calling convention differs from the Lime calling convention and as calls to *memalign* and *mprotect* need their own stack space, a separated stack is needed.

5.6 Experiments

In order to isolate the overhead of procedure call mechanisms from other computations, three programs with little computation but extensive calls were selected as usage profiles, each with different characteristics: Summation, Unbalanced Binary Tree, and Quicksort. Some experiments have a single-threaded and multi-threaded version. Each multi-threaded version has two variations: the "cores" variation and the "quantity" variation. The "cores" variation tests from one to eight threads to examine the scalability over four individual cores (or eight hardware threads provided by "Hyper-Threading Technology" (Marr et al., 2002)). The "quantity" variation tests the scalability when the number of threads is greatly larger than the number of available physical cores in the system. In multi-threaded experiments, the stack address space of 1 GB is divided equally among each thread, so to keep the total used memory constant for avoiding impacts of the virtual memory management. The size of stack chunks is eight pages or 32 kilobytes, excluding the guard page.

There are two summation experiments: deep summation and big summation. The deep summation generates the stack overflow by executing a deep recursion while the big summation produces the stack overflow by increasing the size of the stack frame.

Deep Summation. This program sums the numbers from 1 to n recursively. This example is compiled without tail recursion optimization on purpose.

Listing 5.10: Summation

```
int summation(int n) {
   int ret;
   if (n == 0) { return 0; }
   ret = n + summation(n - 1);
   return ret;
}
```

This experiment aims to magnify the procedure invoking overhead by calling a deep recursive procedure with a small stack frame, and that contains a minimum of computation. For the multi-threaded version, each thread sums the numbers from 1 to m (m is divisible by the number of threads). The "cores" variation was run with from 1 to 20 threads, and the "quantity" variation was run with 20, 32, 64, 128, 256, 512 and 1024 threads.

Big Summation. This experiment aims to test the effect the stack frame size has on the various stack implementations. Programs allocate stack frames of various sizes. To understand what the typical distribution of stack frame sizes is, we analyzed Gnuplot 4.6.0 (we analyzed three other well-known open-source programs as well to be sure that Gnuplot is representative). Figure 5.5 shows the relative frequency of declared C functions for stack frame sizes from 4 to 256 bytes and the relative number of calls in a typical run. It turns out that 98% of function calls are to functions with a stack frame of 256 bytes or less, and about 30% are to functions with a stack frame size of 32 bytes. The average is about 50 bytes.

The function summation above has a stack frame of 8 bytes (4 for the return address and 4 for the parameter). We have modified it by allocating local variables to increase the stack frame size to 16, 32, 48, 64, 80, 96, 112 and 128 bytes.

Unbalanced Binary Tree. This experiment implements an ordered binary tree, and this tree is a balanced binary tree of integers that is 20 levels deep. There is an unbalanced branch (1 million integers) added to the tree. The program will spend 70% of the time in searching for the integer contained within the 20 levels deep balanced portion and 30% of the time for the unbalanced branch, triggering a spike in stack usage.

This experiment compares the performance among the different stack mechanisms for a large number of highly variable-sized stacks usage. For the multi-threaded

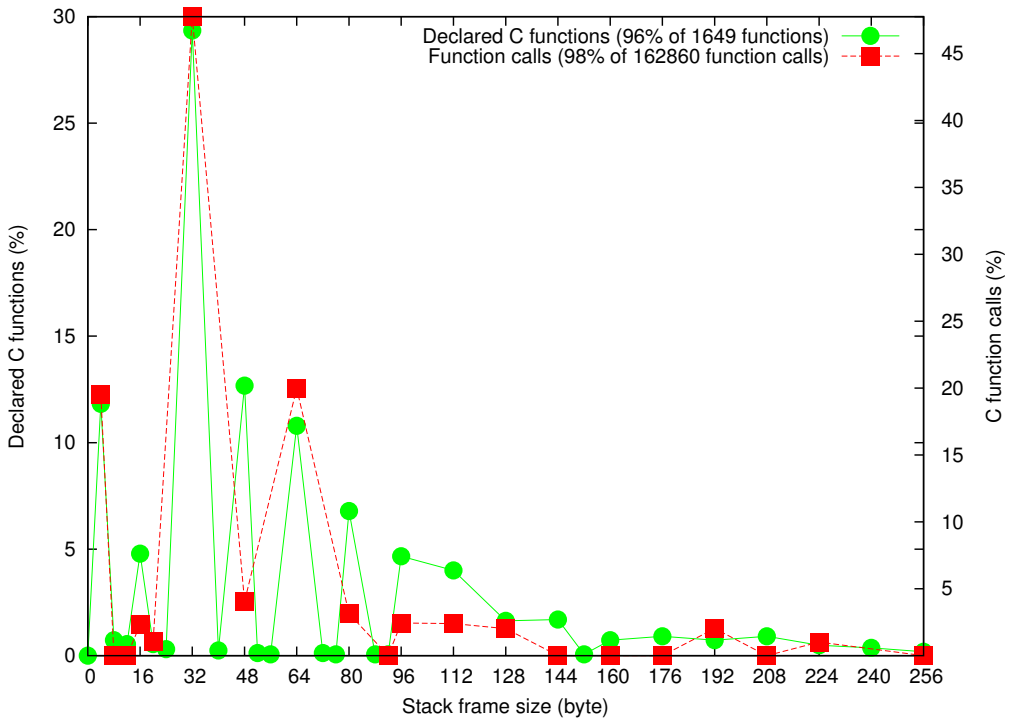

Figure 5.5: Distribution of stack frame sizes of Gnuplot

version, each thread performs 100 searches as the number of threads increase. The "cores" variation was run with from 1 to 20 threads, and the "quantity" variation was run with 20, 32 and 64 threads.

Quicksort The implementation is taken from (Kernighan and Ritchie, 1988). This experiment is meant to be representative of programs that do not have a deep calling structure but instead contain some computation (here the comparisons and swaps). We compare various stack mechanisms for sorting $10^6, 10^7$, and 10^8 random integers with a single-threaded version only. As the calling structure is so shallow that all stack frames are in the first chunk of Guard-Page and Look-Ahead, there would be no contention between threads in a multi-threaded version.

5.6.1 Results

The experiments were run on the following processors:

Pentium 4 released in November 2000, 3.2GHz, containing 42 million transistors, is based on the NetBurst architecture featuring a 20 stage pipeline to achieve a high CPU speed.

Core 2 Duo released in May 2007, 1.8GHz, containing 291 million transistors, has 2 (physical) processor cores and a 14 stage pipeline.

Sandy Bridge I7 released in January 2011, 3.4GHz, containing 1.16 million transistors, has a 14-17 stage pipeline and has 4 physical cores and 8 logical cores through hyper-threading technology.

Haswell I7 released in August 2013, 3.4GHz, containing 1.4 billion transistors, has a 14-19 stage instruction pipeline and has 4 physical cores and 8 logical cores through hyper-threading technology. It improves the back end of the pipeline: the instruction decode queue is not statically partitioned between the two threads that each core can service.

Each experiment was run sixty individual times. The results reported here are the averages with a 95% confidence interval. The difference between the maximum and minimal total execution time, as reported in Table 5.2 for one set of experiments, is small enough, particularly for larger running times. For the other experiments, only the average value is reported.

5.6.2 Impact of Processor Architecture

In the first experiment, we analyze the impact of the processor architecture on the relative efficiency of the procedure calling mechanism. We use single-threaded Deep Summation with different depths of recursion, as Deep Summation makes the most use of the stack; the results are reported in Figure 5.6. As expected, the running time is linear with respect to the sum being calculated. However, while on older processors, Guard-Page, Look-Ahead, Caller-cleanup, and Callee-cleanup perform nearly identical, the newer the processor, the better Guard-Page and Callee-cleanup perform: Look-Ahead and Caller-cleanup perform half the cleanup in the callee and half the cleanup in the caller, resulting in one more instruction. A possible explanation is that the deep pipeline of Pentium 4 can cope with that better than newer processors.

Caller-cleanup and Callee-cleanup use a fixed stack size of 1GB and are not scalable, whereas Guard-Page and Look-Ahead allocate chunks of eight pages (plus one guard page for Guard-Page). Guard-Page and Callee-cleanup have a similar caller and callee sequences. However, the extra overhead for allocating chunks makes Guard-Page slower than Callee-cleanup: for summing up to 100 million, Guard-Page and Look-Ahead allocate approximately 24420 chunks (due to internal fragmentation, LookAhead needs five more chunks than Guard-Page). Surprisingly, Guard-Page is more efficient than Caller-cleanup, the standard GCC convention, which itself is marginally faster than Look-Ahead.

5.6.3 Impact of Usage Profile in Single-Threaded Runs

All the remaining experiments were carried out on Haswell I7.

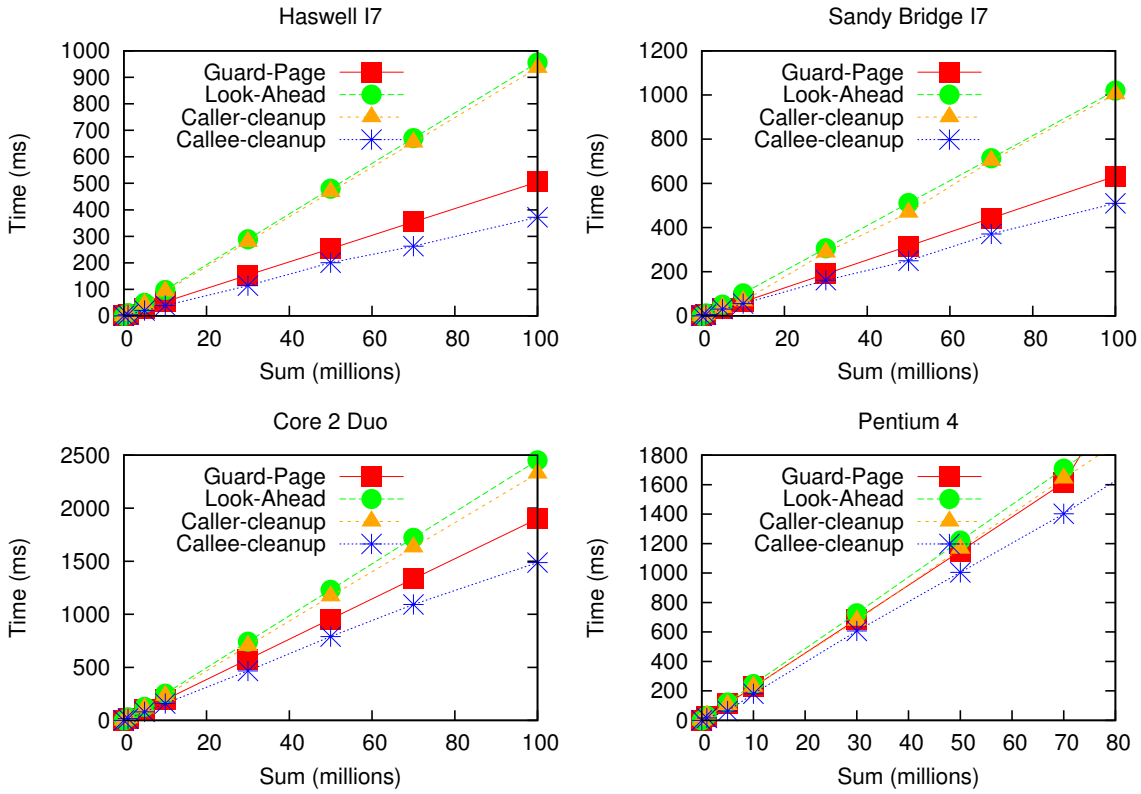

Figure 5.6: Single-Threaded Deep Summation on Haswell i7, Sandy Bridge i7, Core 2 Duo and Pentium 4

Deep Summation. As can be seen in Figure 5.7, the overhead of the dynamic memory allocation for the Heap mechanism is significant. The figure also gives the times of GCC without optimization ("gcc") and GCC with optimization ("gcc -O2"). GCC with optimization has the best performance because it eliminates the recursive calls and there is no stack space used for method calls. Our Lime compiler with Caller-cleanup performs somewhere in between. The figure also magnifies the observations from the first experiment.

Unbalanced Binary Tree. As can be seen in Figure 5.8, the overhead of the dynamic memory allocation for the Heap mechanism continues to be very significant. The trends observed in Summation continue to hold; the only difference is that the Caller-cleanup mechanism runs faster than the Guard-Page mechanism: the overhead from dynamic memory allocation for every stack chunk is more significant than the overhead of Caller-cleanup compared to Callee-cleanup.

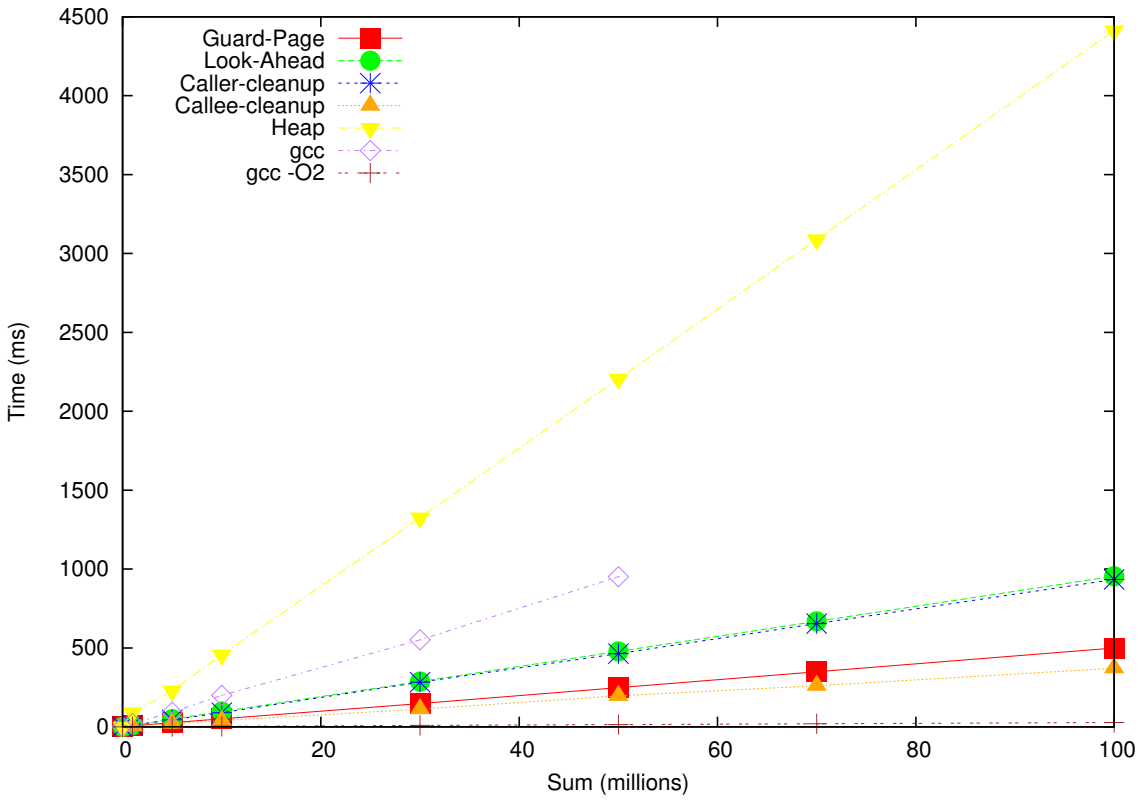

Figure 5.7: Deep Summation Single Threaded

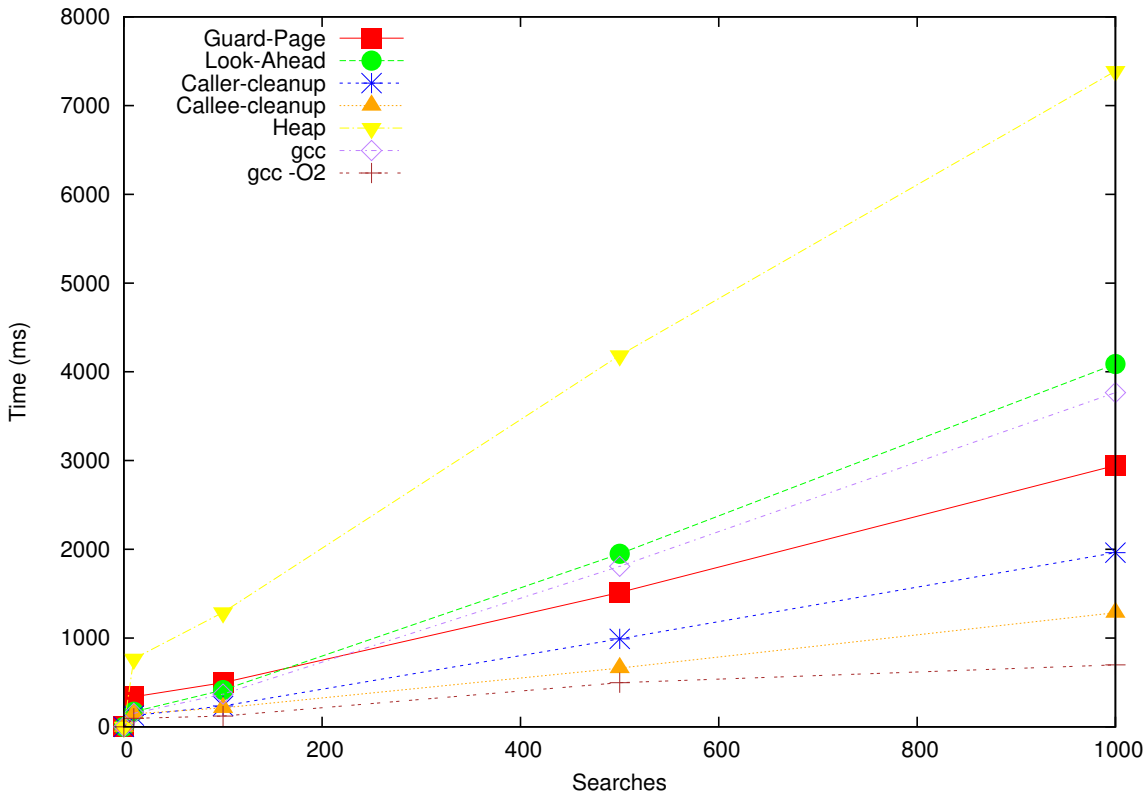

Figure 5.8: Unbalanced Binary Tree Single Threaded

Threads	1	2	3	4	5	6	7	8
Maximum	451	551	630	720	824	927	1060	1193
Average	430	529	615	701	806	911	1042	1156
Minimum	417	502	586	659	778	898	1014	1125

Table 5.2: The Original Times in ms of the Unbalanced Binary Tree Multi-threaded Look-Ahead Experiment Over 60 Runs

Elements	10^6	10^7	10^8
Callee-cleanup	152 ms	1,686 ms	19,086 ms
Guard-Page	156 ms	1,737 ms	19,618 ms
Caller-cleanup	157 ms	1,755 ms	20,007 ms
Look-Ahead	159 ms	1,775 ms	20,257 ms
Number of calls	14,703,523	160,540,046	1,855,875,685
Maximal depth	19	21	25

Table 5.3: Quicksort Single Threaded

Quicksort. As can be seen in Table 5.3, the maximal call depth (including auxiliary functions) is so shallow that all computation remains within the first chunk. With Guard-Page mechanism, a guard page is not hit, and Guard-Page performs consistently better than Look-Ahead. However, the difference is at most 3%.

5.6.4 Impact of Usage Profile in Multi-Threaded Runs

Deep Summation. The times reported in Figure 5.9 are the total running time for all threads for summing from 1 to n, where n is divisible by the number of threads). The Guard-Page mechanism, while starting out with better performance than Caller-cleanup and Look-Ahead, demonstrates the worst scalability, and eventually, the worst performance, as the number of threads exceeds the number of available cores. To isolate the cause, we introduced a *stack chunk reuse* mechanism: rather than deallocating stack chunks; they are placed in queue for future use. On allocation, first chunks from that queue are used before a new chunk is allocated through *memalign* and *mprotect*. Calls to *mprotect* take more than 100 times as long as calls to *memalign* for allocating page-aligned memory, which itself takes about twice as long as *malloc*. Guard-Page needs *mprotect* and *memalign*. Calls to *mprotect* cause the processor's TLB to be flushed, thus incur a heavy penalty. The new mechanism is called "Guard-Page-with-reuse", the old mechanism is renamed to "Guard-Page-without-reuse". As can be seen in Figure 5.10, the Guard-Page-with-reuse mechanism has a better performance than Look-Ahead when the number of threads exceeds 200. The reason is that the concurrency is so high that some threads manage to start their cleanup phase while

the others are still in their growth phase, in this case, the stack chunks are able to be reused, meaning there are fewer calls to *malloc* and *mprotect*. To magnify this effect, when summation is repeated ten times, Guard-Page outperforms Caller-cleanup and Look-Ahead, see Figure 5.11.

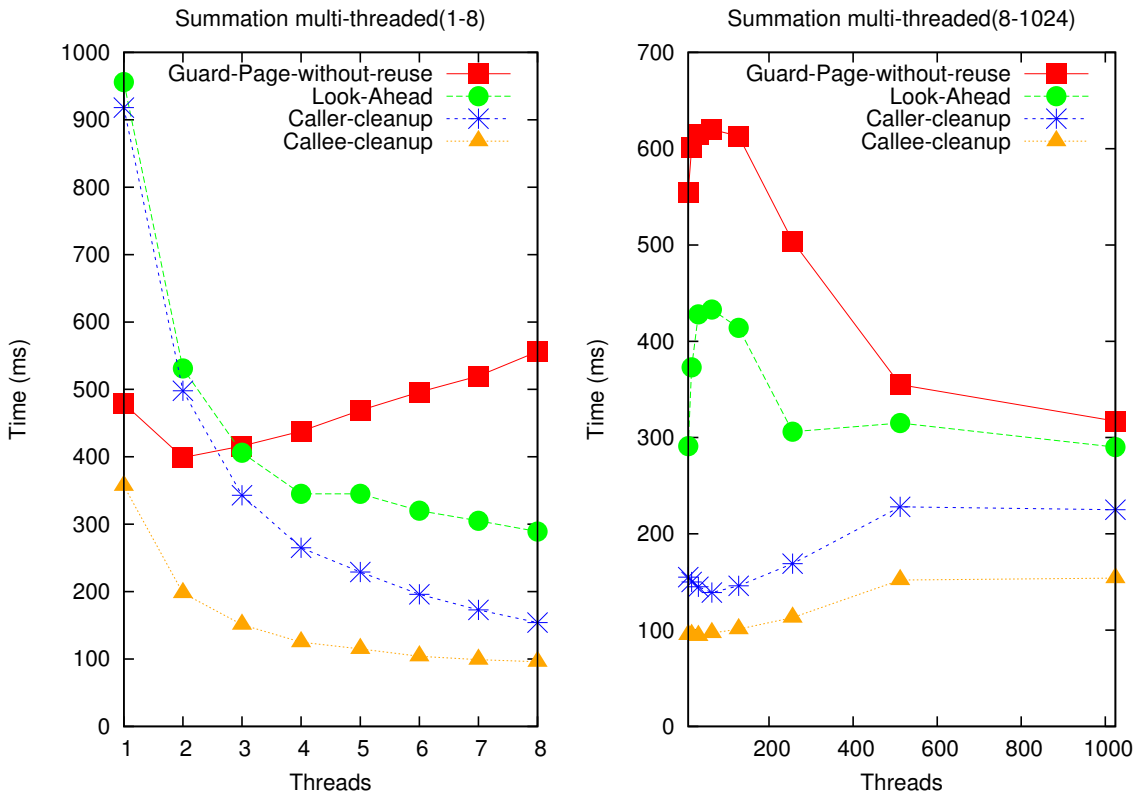

Figure 5.9: Deep Summation Multi-threaded (Guard-Page without reuse)

Big Summation. As evident from Figure 5.12, there is almost a linear increase in the time with the increase for the stack frame size due to the need for allocating memory, despite the same computation taking place. For a single run, Guard-Page performs worse than Look-Ahead because of the overhead of calling *mprotect*. However, if the runs are repeated ten times and chunks are reused, Guard-Page outperforms Look-Ahead significantly.

Unbalanced Binary Tree. The time reported in Figure 5.13 is the total time for all threads to finish. Guard-Page-with-reuse scales identically to Caller-cleanup and Look-Ahead. Guard-Page-with-reuse continues to show better scaling than Look-Ahead.

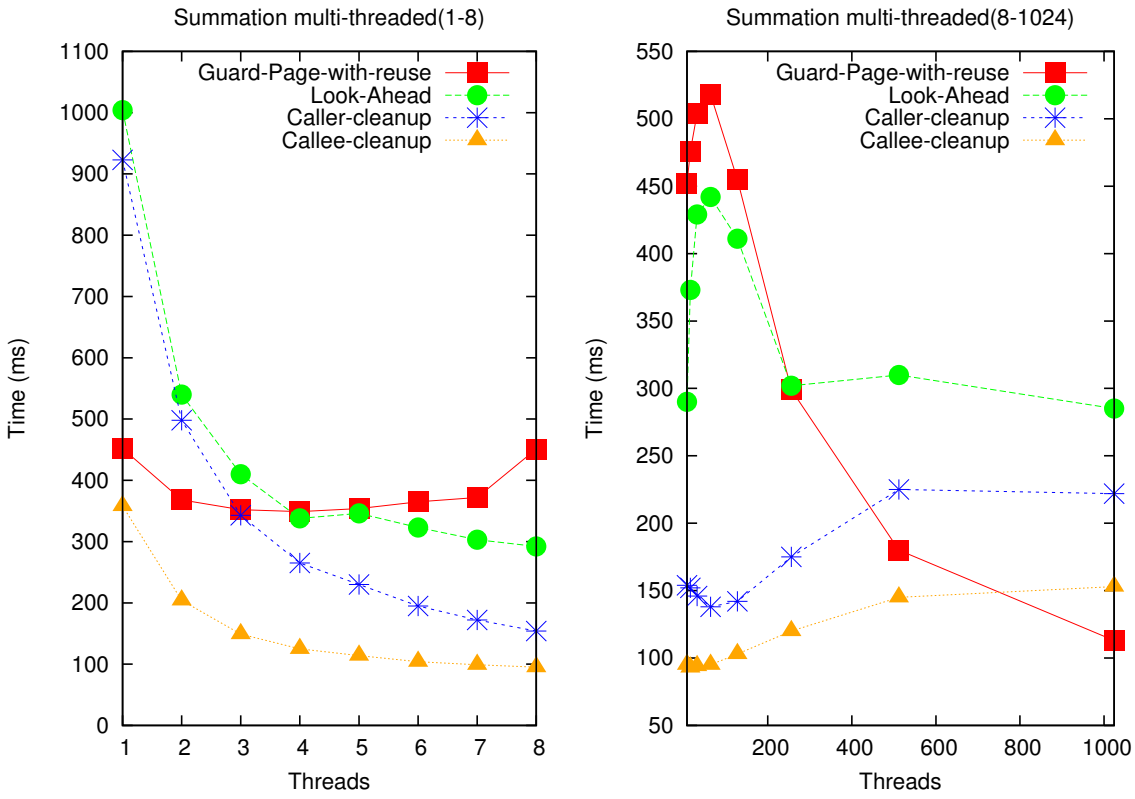

Figure 5.10: Deep Summation Multi-threaded (Guard-Page with reuse)

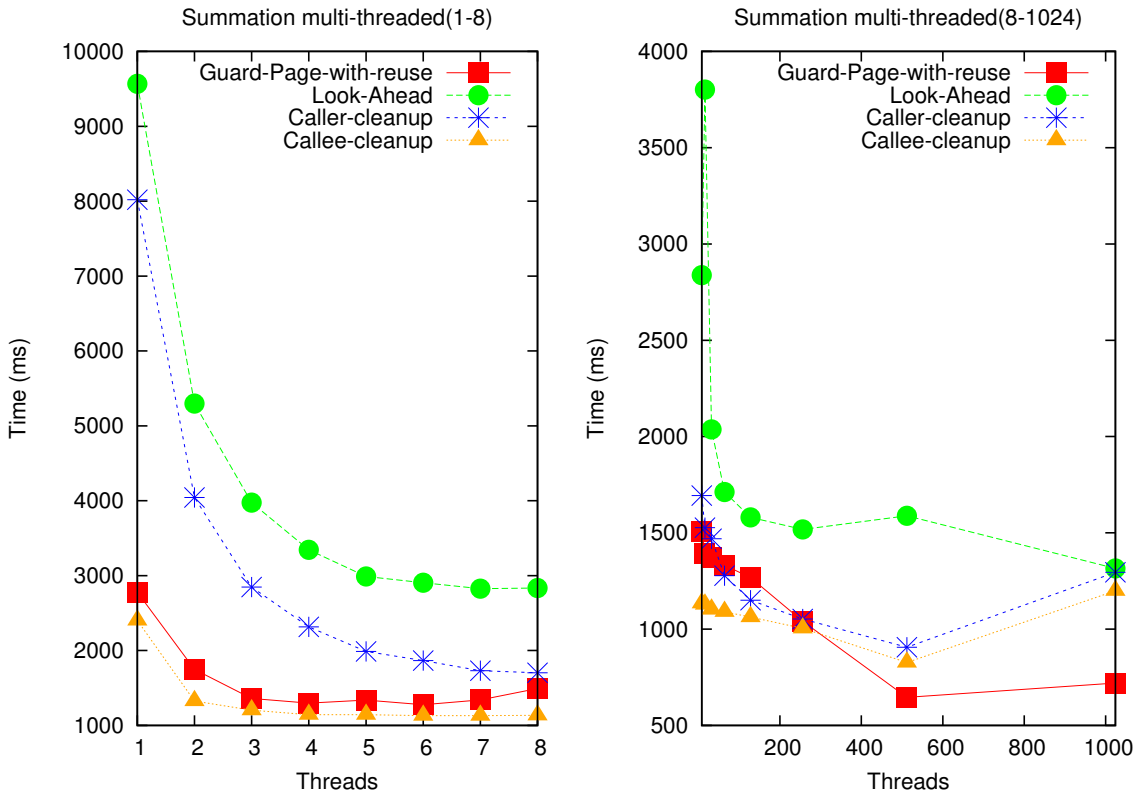

Figure 5.11: Deep Summation Multi-threaded (Guard-Page with reuse), 10 repeats

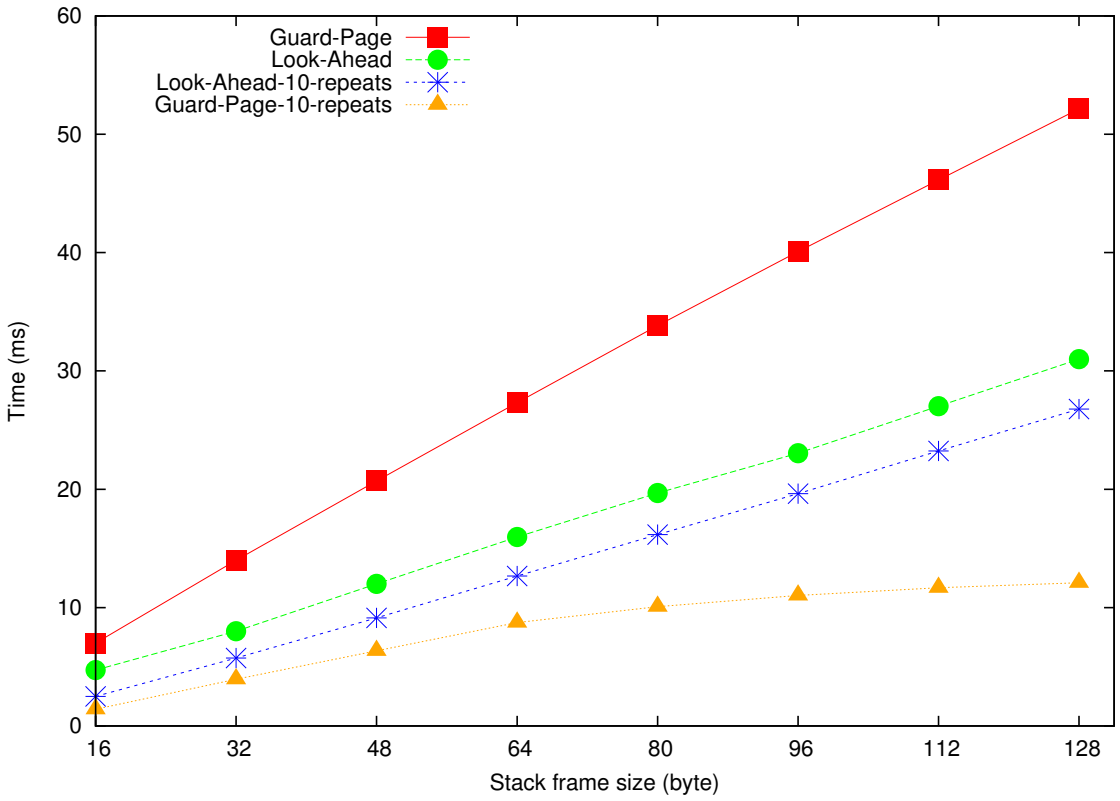

Figure 5.12: Big Summation multi-threaded (Guard-Page with reuse)

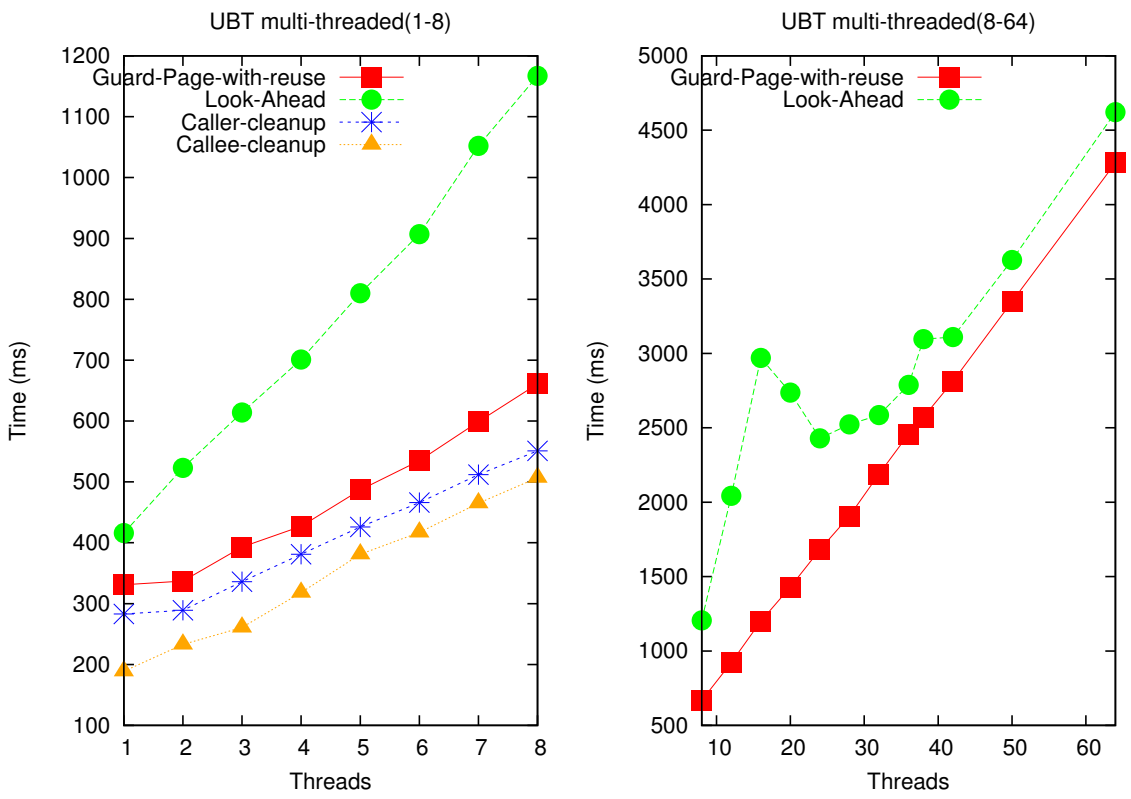

Figure 5.13: Unbalanced Binary Tree Multi-threaded

Neither the Caller-cleanup nor Callee-cleanup was tested when the number of thread exceeds 8, as there is not enough space to share for a large number of threads that require a fixed size and "large enough" stack.

The spike of Look-Ahead in Figure 5.13 results from mutex contention. When threads allocate and free memory at the same time, there could be increased mutex contention in the malloc function. To solve this problem, "Glibc creates additional memory allocation arenas if mutex contention is detected". The number of arenas usually equals to the number of cores (Linux, 2018).

To summarize, Guard-Page stack mechanism tends to perform worse than Look-Ahead, if (1) there is a deep recursion without repeats or there are a large number of short-lived threads (so the overhead of mprotecting does not amortize), (2) the stack frame size is large (so the guard page is more frequently hit). However, none of these situations are typical. Thus, the conclusion we can draw is that Guard-Page performs better in practice, particularly for languages that do not allow arrays to be allocated on the stack and thus have small stack frames.

Chapter 6

Guarded Commands in Lime

In this chapter, we first present an overview over existing lightweight thread implementations. Then, we discuss the previous implementation in Lime for guarded commands. Lastly, we propose the new implementation of guarded commands.

6.1 Lightweight Thread Implementations

In this section, we first discuss the lightweight thread implementations in practical programming languages. Then we examine the lightweight thread implementations.

In this section, we examine two typical implementations of lightweight threads, Erlang's processes and Go's goroutines.

6.1.1 Erlang Process

In Erlang, the concept of the lightweight thread is called a *process*. As the name implies, Erlang processes are similar to OS processes: an Erlang process has its address space and can communicate by sending messages, and a "preemptive" scheduler controls the execution.

A process in Erlang consists of mainly four blocks of memory: a stack, a heap, a message area, and the PCB:

- Stack: stores local variables, arguments and the return address;
- Heap: stores larger structures, such as lists and tuples;
- Mailbox: stores received messages;
- PCB: serves as a repository for the state of the process, similar idea of PCB in an OS.

Erlang dynamically allocates the stack, the heap and the mailbox, which can grow and shrink as needed. Like operating systems, Erlang runtime system allocates the

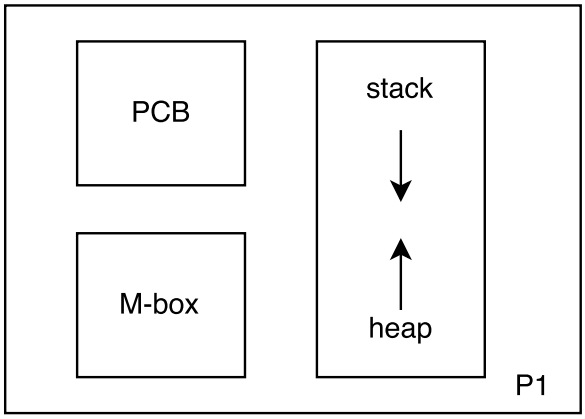

Figure 6.1: Erlang's Process Memory Layout

heap and the stack together, and the stack grows towards lower memory address while the heap towards higher memory address. The default size of this stack-heap memory is 233 words. If the pointers *htop* and *stop* were to meet, it would mean the process has run out of memory and the runtime system has to do a garbage collection to free up memory. It uses a per process copying generational garbage collector.

As a concurrent programming language, Erlang has a preemptive scheduler on top of the cooperative scheduler. The scheduler ensures that an Erlang process can yield within a limited time. A process can only be suspended at well-defined points, such as at a receive or a function call. Strictly speaking, the scheduler of Erlang is cooperative.

There are two reasons that Erlang's scheduler can be regarded as a preemptive scheduler on top of a cooperative scheduler. First, as a functional programming language, there is no way for an Erlang process to run for a long time without calling a function. Second, because there are no other loop constructs than recursion and list comprehensions, it is impossible for an Erlang process to loop forever without a function call. There is a reduction counter for each function. The reduction counter is decreased by one when the function is called. The process is suspended when the reduction counter reaches the limit 0.

6.1.2 Goroutine

In Go, the concept of the lightweight thread is called a *goroutine*. A goroutine includes a stack, an instruction pointer and other information for scheduling goroutines, such as any channel it might be blocked on or the thread it binds to. Goroutines use channels as the primary synchronization and communication primitive, and a cooperative scheduler controls the execution. In Go's runtime system, the G structure (in Listing 6.1) represents a Goroutine, which includes fields necessary to keep track of its status (Google, 2018b).

Listing 6.1: G Structure in Go

```
type g struct {
    stack stack
    stackguard0 uintptr
    stackguard1 uintptr

    _panic *_panic // innermost panic
    _defer *_defer // innermost defer
    m *m // current m;
    ...
}
```

As we have discussed in Section 1.1.1, there are three models for threading. One is $N:1$ where several userspace threads run on one kernel thread. Context switching of this model is fast, but it cannot take advantage of all of the cores on the machine. Another is $1:1$ where one userspace thread matches one kernel thread. It can take advantage of multicore systems, but context switching is slow because it has to trap through the OS.

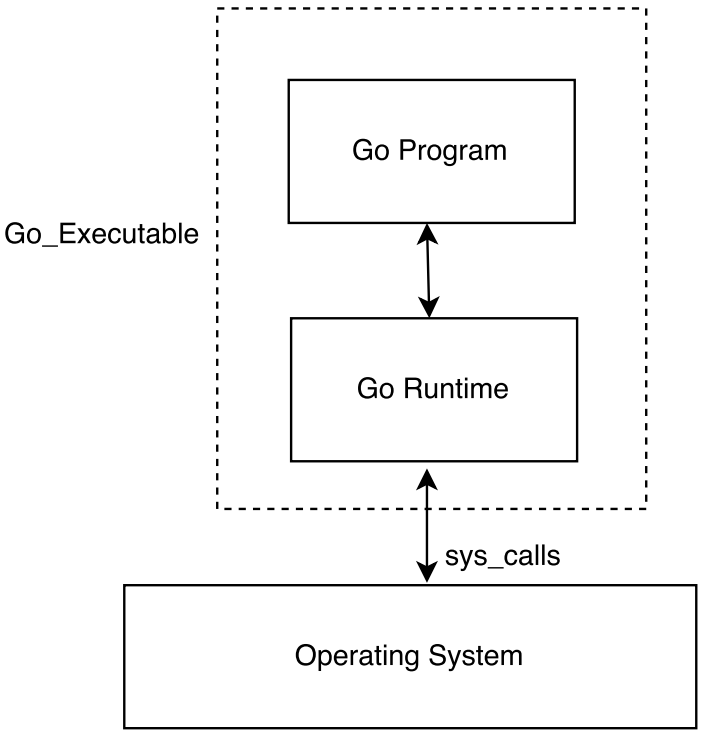

Figure 6.2: Go's Execution Model

To achieve better performance, Go uses an $M:N$ scheduler which maps an arbitrary number of goroutines to a fixed amount (the number of available cores) of kernel threads. This scheduler can achieve quick context switches for goroutines and

takes advantage of all the available cores on the machine. The only disadvantage of this approach is the complexity it adds to the scheduler and the runtime system.

There are three internal types of channels: synchronous channels, asynchronous channels and asynchronous channels with zero-sized elements. Compared with synchronous channels, asynchronous channels contain a ring buffer and position information of the data while asynchronous channels with zero-sized elements only contain a counter which represents the number of items in the channel. In this section, we focus on the synchronous channels.

For each synchronous channel, as illustrated in Listing 6.2, there are two waiting queues for the senders and the receivers. The field *closed* represents whether the channel is closed or not. One lock protects a synchronous channel. When a goroutine sends data to a synchronous channel, it locks the channel first and then checks whether it needs to block or to wake up the receiver.

Listing 6.2: Synchronous Channels in Go (Google, 2018b)

```
type hchan struct {
  qcount uint // total data in the queue
  dataqsiz uint // capacity of the circular queue
  buf unsafe.Pointer // points to an array of dataqsiz elements
  elemsize uint16
  closed uint32
  elemtype *_type
  sendx uint // send index
  recvx uint // receive index
  recvq waitq // list of recv waiters
  sendq waitq // list of send waiters
  lock mutex
}
```

6.2 Previous Implementations of Guarded Commands in Lime

As we have discussed before, guarded commands can provide programmers with a more natural way to prove the correctness of the program. Also, guarded commands can give the system designer a unified and straightforward design view. However, as we have explained, how to efficiently implement guarded commands is our current research focus.

For the ease of discussion, we take the *start_read* method of the Reader and Writer problem (from Section 3.5.3) as an example to compare different implementations. Firstly, we discuss a simple solution by using busy waiting. Secondly, we demonstrate the previous implementations in Section 6.2.2.

Listing 6.3: Start_read Method

```
method start_read()
   when n > 0 do
      n := n - 1
```

6.2.1 Busy Waiting

The simplest solution is to have the disabled methods busy waiting for the guards when the method guards do not hold. When the program enters a guarded method, it checks whether the guard is true or not. If it is not, the program sits in a tight loop waiting until it is. Because of the significant drawback of CPU time-wasting, this solution is never implemented.

Listing 6.4: Reader and Writer Solution Using Busy Waiting

```
void start_read() {
   lock();
   while (rw <= 0) {unlock(); lock();}
   rw--;
   unlock();
}
```

6.2.2 Previous Implementations

Lou (2004) implements the guarded commands by using monitors in Java (Listing 6.5). In principle, all method calls that invoke the other objects' methods need to be *synchronized*. So the call statement is put between the two operations, *monitorenter* and *monitorexit* (*P* and *V* operations discussed in Section 2.1.2). For all guarded classes, the execution of the method may affect the guards of actions or guarded methods. So when a method or an action is successfully executed, all threads need to be notified so that the suspended threads have to re-evaluate the guards. The body of a guarded method can be executed if the guard holds; otherwise, the method is blocked, and the thread which is executing the method gets suspended. If the guard of the action holds and the action is executed, otherwise it skips current action.

Listing 6.5: Reader and Writer Solution Using Monitors (Lou, 2004)

```
public synchronized void start_read() {
   while (rw <= 0) {
     try {wait();}
     catch (InterruptedException e) {}
   }
   rw--;
 }
```

Cui (2009) implements the guarded commands by using condition variables in Pthread. Each object, mapped to a thread, has a private lock and a condition variable. All the methods of the object have an additional parameter which points to the original caller. The thread acquires the lock at the entry of the method and releases the lock at the exit. The method can only be executed when the guard is true. Otherwise, it waits on the condition variables. Before the worker thread exits the method, the related waiting threads are wakened up to re-evaluate the guard. If the guard of the action holds and the action is executed, otherwise it skips current action.

Listing 6.6: Reader and Writer Solution Using Condition Variables

```
void start_read(rw_arbiter *a) {
   pthread_mutex_lock(&a->mutex);
      while (a->rw <= 0)
         pthread_cond_wait(&a->cv, &a->mutex);
      a->rw--;
   pthread_mutex_unlock(&a->mutex);
}
```

A fixed number of threads are created in Lou's implementation and a false deadlock situation may happen when all Java threads are conditionally blocked but there are other objects that are eligible to execute. Cui's implementation maps one object to one thread. The threads are managed by the operating system. These two guarded commands implementation have the same deficiency: all the blocked callers need to re-evaluate the guard when a method or an action is successfully executed. The performance decreases significantly when the number of active objects increases. We can see in Section 6.4.1, for the priority queue example, if 80 active objects are mapped to 80 threads, both monitors and condition variables implementations are around 70 times slower than the lightweight thread implementations.

6.3 Guarded Commands Implementations

Compared with previous implementations of guarded commands, a new implementation of guarded commands is illustrated on the *start_read* method which is from Listing 3.7. The worker thread first unlocks the object and then switches to the

scheduler when the guard does not hold. Otherwise, the worker thread executes the method body and then unlocks the current object. The current implementation of Lime only supports basic Boolean expressions.

Listing 6.7: Reader and Writer Solution Expressed in Lime

```
method start_read()
   while true do
      if lock(this.lock) then
         if this.rw <= 0 then
            this.rw := this.rw - 1
            unlock(this.lock)
            return
         else
            unlock(this.lock)
      transfer(Scheduler.schedule(originator))
```

6.3.1 Translation Schemes

We use the following Lime class in Listing 6.8 to demonstrate the translation scheme for the guards.

Listing 6.8: Lime Class Example

```
class C
   var v: int
   var g1, h1: bool
   init()
      P
   method M1() : int
      when g1 do
         Q1
   method M2()
         Q2
   action A1
      when h1 do
         R1
   action A2
         R2
```

Class *C* represents a Lime class and it contains a local field *v*. There are two methods in this class: *M1* is a guarded method and *M2* is an unguarded method. Method *M1* can be called only when the guard *g1* holds. There are two actions in this class *C*: Action *A1* is a guarded action and *A2* is an unguarded action. *A1* can be executed when the guard *h1* holds. Methods in Lime may have arguments and return values.

The translation scheme of the guarded method is demonstrated in procedure *C_M1*. The worker thread obtains the lock when entering and releases the lock when exiting. If the guard *this.g1* does not hold, the worker thread releases the lock and calls *transfer* function, shown in Listing 6.14, to switch to the scheduler. The method *transfer*(*Scheduler.schedule*(*originator*)) is similar to the *yield* method and the parameter *originator* points to the source active object which executes its actions. The method body *Q1* is translated to *this.Q1* in Listing 6.9 without any semantic changes. The worker thread puts the current object to the local queue before it returns from the method call.

Listing 6.9: Lime Guarded Method Translation Expressed in Lime

```
method C_M1(this) : int
   while true do
      if lock(this.lock) then
         if this.g1 then
            this.Q1
            putrunq(this)
            unlock(this.lock)
            return
         else
            unlock(this.lock)
      transfer(Scheduler.schedule(originator))
```

Unlike the guarded method, an unguarded method only needs to acquire the lock before executing the method body *this.Q2*. The worker thread puts the current object to the local queue before it returns from the method call.

Listing 6.10: Lime Unguarded Method Translation Expressed in Lime

```
method C_M2(this)
   while true do
      if lock(this.lock) then
         this.Q2
         putrunq(this)
         unlock(this.lock)
         return
      else
         transfer(Scheduler.schedule(originator))
```

Actions in Lime are executed automatically and can only be invoked by the scheduler. Actions have no arguments or return values. The translation scheme of the guarded actions is demonstrated in procedure *C_A1*. Unlike methods, actions never return. It switches back to the scheduler at the end of actions' execution.

Listing 6.11: Lime Guarded Action Translation Expressed in Lime

```
method C_A1(this)
   while true do
      if lock(this.lock) then
         if this.h1 then
            this.R1
            unlock(this.lock)
         else
            unlock(this.lock)
      transfer(Scheduler.schedule(originator))
```

An unguarded action is translated as follows:

Listing 6.12: Lime Unguarded Action Translation Expressed in Lime

```
method C_A2(this)
   while true do
      if lock(this.lock) then
         this.R2
         unlock(this.lock)
      transfer(Scheduler.schedule(originator))
```

6.3.2 Context Switches

As we have discussed in Section 1.1.1, the overhead of a thread context switch is still significant. Thus Lime uses a lighter concept — coroutines. Coroutines in Lime are cooperatively scheduled, and arguments are passed on the stack. The context switch between coroutines only happens at well-defined points, when an explicit call is made to the Lime runtime scheduler. The code in Listing 6.13 switches from the scheduler to a coroutine by saving the ESP and EBP registers on the coroutine's stack and restoring current coroutine's ESP and ESP registers. The return address is always stored on the top of the current stack.

Listing 6.13: Switch to Coroutine

```
switch_to:
  MOV ECX, [ESP + 4] ;coro_stack
  MOV EAX, [ECX]
  MOV EDX, [ECX + 4]
  MOV [ECX], EBP
  MOV [ECX + 4], ESP
  MOV EBP, EAX
  MOV ESP, EDX
  RET
```

The code in Listing 6.14 switches from a coroutine back to the Lime scheduler (*transfer* in Section 6.3.1).

Listing 6.14: Switch to Lime Scheduler

```
switch_to_sched:
  MOV EAX, [EBP] ;pre EBP
  MOV EDX, [EBP + 4] ;pre ESP
  MOV [EBP], EBP
  MOV [EBP + 4], ESP
  MOV EBP, EAX
  MOV ESP, EDX
  RET
```

6.3.3 Lime Runtime System

In this section, we discuss Lime's runtime system implementation. First, we illustrate the communication between objects. Second, we introduce the Lime cooperative scheduler. Third, we demonstrate the optimization strategies which are applied to increase the performance.

There is a naive approach to implement the Lime runtime system. All the active objects are evenly stored among worker threads. Worker threads are implemented as daemon threads, and the main thread determines when to terminate the program. Worker threads keep searching for enabled objects to execute. To improve the performance, a *next* field is added to the active node to indicate which active object is waiting for execution.

However, the performance of this naive Lime runtime system implementation decays when the number of active objects increases. The worker threads are searching for enabled objects in a cyclical order. The overhead of context switching is accumulated during the search although the context switching is fast in Lime. The *next* field can only relieve the problem, to some content. The Lime runtime system holds all active objects in the pool, rather than enabled objects (Figure 6.3).

In the current implementation of Lime, each worker thread has two FIFO queues for active objects. The suspended object queue is for active objects which "got stuck" at method calls. The enabled object queue is for the active objects which have enabled actions. The disabled objects are not added to the enabled objects queue until the objects become enabled through method calls. These two object queues are initially empty. When an active object has enabled actions, a pointer to it is added to the enabled object queue. If the enabled object "gets stuck" at a method call, this object is added to the suspended object queue. A worker thread requests a reference from these two object queues and evaluates the guard. If the guard holds, the worker thread executes the method call, respectively the enabled action. For the enabled object, the guard needs to be evaluated because an enabled action can be disabled by a method call (for example, the Santa Clause Problem discussed in (Sekerinski and Yao, 2018)). If the guard does not hold or the object is accessed by another worker thread, the current worker thread yields to the scheduler. Before the worker thread exits an

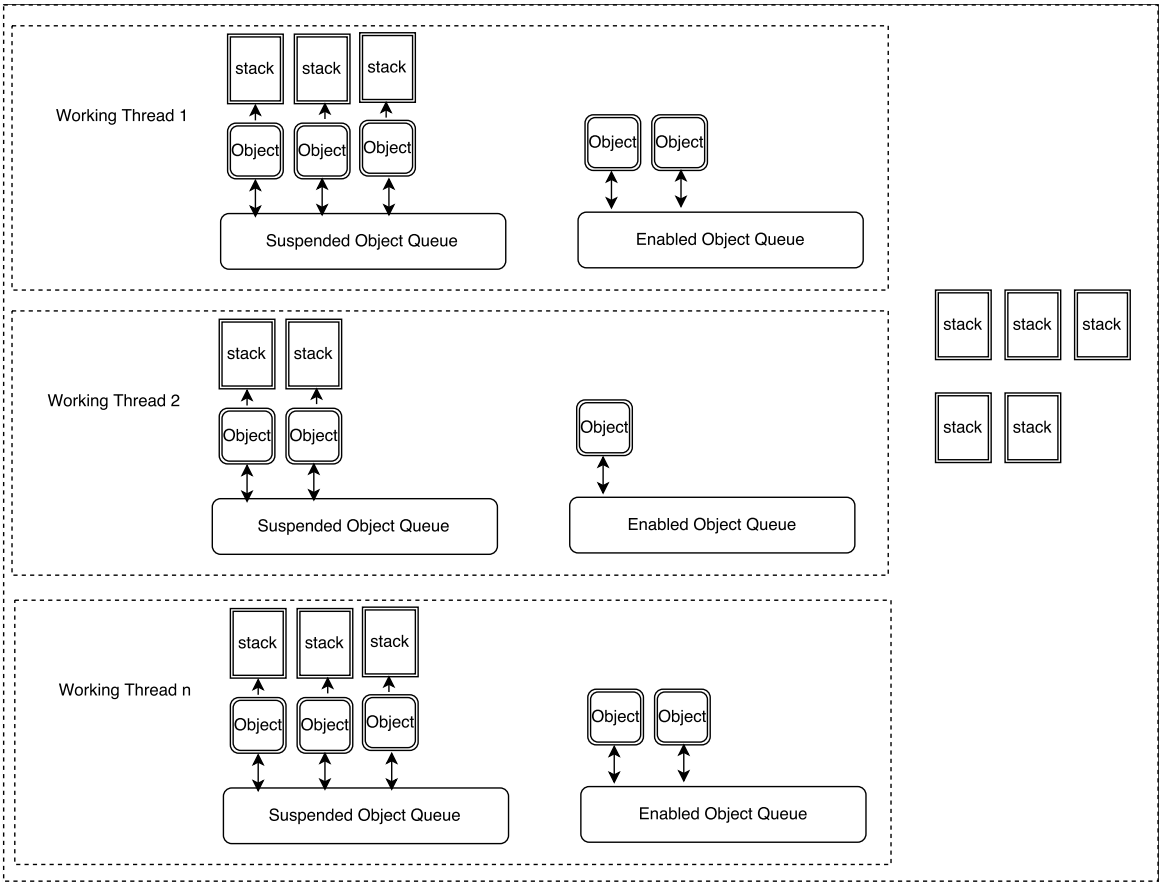

Figure 6.3: Lime Runtime System

active object, it checks whether this active object is enabled or not. If it is enabled, the worker thread adds the current object to the enabled object queue again. The number of the worker threads is defined as a constant (equals the number of cores). The worker threads are implemented as regular threads, meaning that a termination mechanism (will discuss in Section 6.3.3) has to be introduced to determine when to terminate the whole program.

Communication Between Objects in Lime

In Lime, the communication between objects is through synchronous method calls. Condition synchronization is achieved through guards. As we have discussed before, to provide the programmers with a simple design view, Lime allows method calls getting stuck. In this case, the lock should be released when entering the method call while the lock should be acquired when returning from the method call. For example, the method call $a.b(arg1)$ can be translated into Listing 6.15. There are two arguments

that are added to the method call: *a* and *originator*. The argument *a* points to the target object while the argument *originator* points to the source object which invokes the method *b* of the object *a*. All these parameters are stored on the stack. The worker thread puts the *originator* object to its local queue when the current method call gets stuck.

Listing 6.15: Method Call Translation

```
unlock(this.lock)
   b(arg1, a, originator)
lock(this.lock)
```

In Lime, method calls to other objects can be *open* as the exclusive access to the first object is dropped and only regained when the call returns. By comparison, method calls in Java are *closed* as exclusive access to all objects in the call chain is retained. It is already discussed in (Andrews, 2000) that closed calls allow less concurrency and are more prone to deadlocks.

Lime's Cooperative Scheduler

We implement a simple but efficient cooperative scheduling strategy for the Lime runtime system. We explain how this cooperative scheduler works in Figure 6.4.

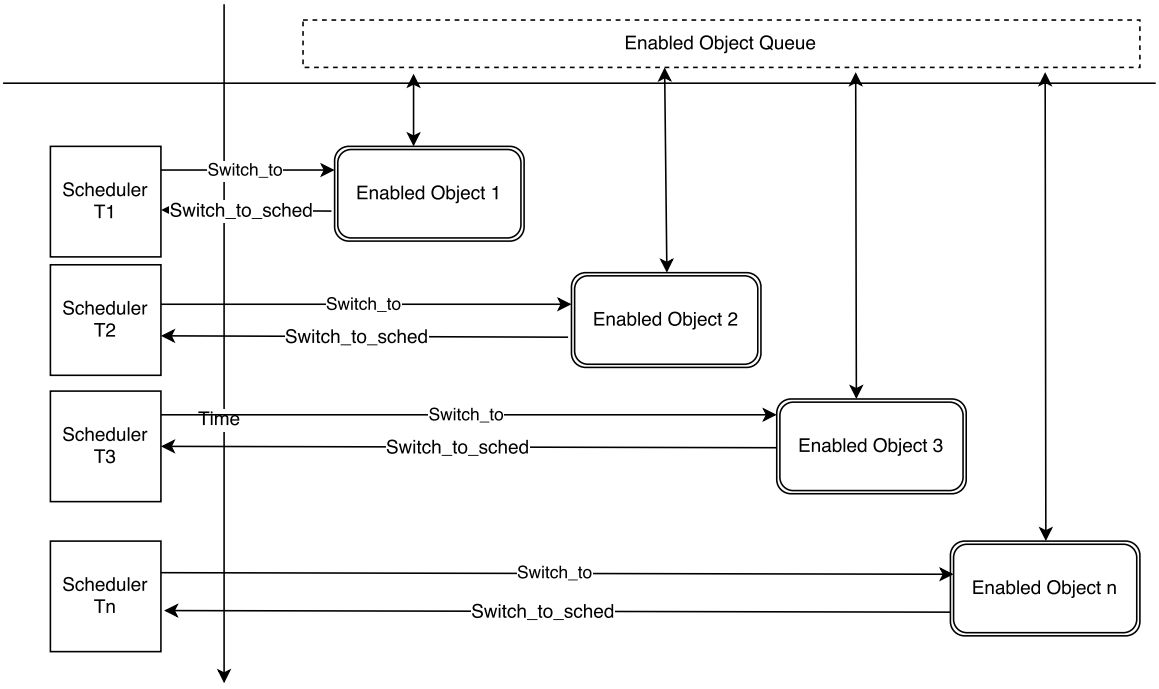

Figure 6.4: Lime Scheduler

	Threading Model	Scheduling	Communication
Erlang	M:N	Preemptive	Asynchronous message passing
Go	N:1/M:N	Cooperative	Channels
Lime	M:N	Cooperative	Synchronized method calls

Table 6.1: Scheduling Strategy Comparison

There are a fixed number of worker threads in Lime, which equals the number of the cores. Each worker thread has a scheduler stack on which the thread starts. The active objects run on their stacks. The worker thread switches to the active object's stack when the active object is selected. When the active object suspends for the method call or exits from the action's execution, the worker thread switches to the scheduler's stack and continues to execute the next active object. The worker thread repeatedly switches to active objects as long as there are available active objects.

The runtime system dynamically adjusts worker threads as needed. For example, if there are no enabled objects left in the enabled object queue, the worker thread could sleep. If there are some new enabled objects are created, the sleeping threads should wake up.

To illustrate Lime scheduling, we use a tabular overview in Table 6.1 to make a comparison among Erlang scheduling, Go scheduling and Lime scheduling. Lime employs cooperative scheduling, same as Go, other than preemptive scheduling in Erlang. The communication in Lime is through synchronous method calls. Communication in Erlang and Go are using asynchronous message passing and channels, respectively. Erlang and Lime use hybrid threading while Go applied user-level threads and moves to hybrid threading from version 1.5.

Lime Runtime Optimizations

Lime implements a global enabled object queue. The enabled object is added to the global queue when the local enabled object queue is full. When the local enabled object queue is empty, the worker thread first tries to fetch a random number of the enabled objects from the global queue.

The Lime runtime system needs to balance between speed and efficiency. On the one hand, to increase efficiency, it has to keep enough running worker threads to utilize hardware parallelism. On the other hand, it has no reason to keep excessive running worker threads to conserve CPU resources and power. We implement schedule algorithms which can keep enough worker threads when the system has enough work to finish and let the excessive threads sleep when the system does not have enough work.

The system has:

- at most W worker threads,
- currently w threads are working, meaning executing actions, and

- currently s threads are seeking, that is, waiting for the objects from the global queue, where s is 0 or 1.

So, $W - w - s$ threads are sleeping, meaning doing nothing. The invariant is:

- either $w = W$, i.e., all threads are working, where $s = 0$, or
- if $w < W$, then $s = 1$, i.e., if not all are working, one thread must be seeking.

The whole program terminates when one thread is seeking and all the others threads are sleeping.

Lock-Free Local Queue and Work Stealing

In the current implementation of Lime, each worker thread has one private local queue. At first, there is only one worker thread working in the runtime system. The second worker thread waits until there are some objects in the global queue. To reduce this delay, we introduce a second implementation (Lime-LF in Section 6.4) of Lime which contains a lock-free local queue for each worker thread and enables idle worker threads to steal work from the others, rather than waiting for the objects from the global queue. The load-acquire and store-release instructions are used to pass the information between worker threads cooperatively and implement the lock-free local queue. This work is inspired by Golang's implementation (Google, 2019). In addition, one active object can be accessed by multiple objects. In this case, the enabled active objects can be distributed more efficiently. There is a visible performance improvement when the system contains sufficient enabled objects (in Section 6.4.2). However, the overhead of the lock-free local queue can become significant if there are a limited number of enabled objects. More implementation details are discussed in the gitlab repository (`Documents/Lime/Lime_Runtime_LockFree.md`).

The global object queue has a lock, and only one worker thread can access it at a time. The worker thread can only access its private local queue. For the lock-free local queue implementation, the local queue may be accessed by multiple worker threads (work-stealing) at the same time. The load-acquire and store-release instructions can achieve the memory barrier's effect. Please note that the current lock-free implementation only works on X86's memory model. The worker thread needs to acquire the object's lock before entering the object and release it after exiting from the object. In this case, each object can be accessed by at most one worker thread at a time.

The performance of the lock-free local queue varies from case to case. The lock-free local queue improves the performance if there are a large number of active objects, for example, the test cases discussed in Section 6.4.1 and 6.4.2. However, it decreases the performance if the number of the available active objects is small, for example, the test cases discussed in Section 6.4.3 and 6.4.4.

6.4 Experiments

To isolate the overhead of guarded commands implementation from other computations, three programs with little computation but representing different concurrent models were selected as usage profiles, each with different characteristics: priority queue, MapReduce and leaf-oriented tree. Priority queue is a linear structure while MapReduce and leaf-oriented tree have a general tree structure.

Each experiment version has two variations: The "thread" variation tests scalability between the normal thread and the lightweight thread implementations. The "lightweight thread" variation compares scalability among the lightweight thread implementations when the number of the objects significantly exceeds available cores in the system. Each experiment contains two typical thread implementations: Java threads and Pthreads and five lightweight thread implementations: Go, Erlang, Haskell, Lime and Lime-LF. Lime-LF represents the Lime runtime system with the lock-free local queue. For each test case, we use the same random numbers for all implementations.

For Java's implementation, active objects are mapped to Java threads, and guarded commands are implemented by using monitors. For Pthread's implementation, active objects are mapped to Pthreads, and guarded commands are implemented by using condition variables. In Go and Erlang, active objects are mapped to Goroutines and Processes, respectively. Glasgow Haskell Compiler (GHC) (GHC, 2018) is a Haskell compiler supporting concurrency by implementing a lightweight thread system. In Lime, active objects are mapped to coroutines.

This section describes the performance of different implementations. We use Java, Pthread, Erlang, Go, Haskell and Lime to implement the priority queue, MapReduce, and leaf-oriented tree examples. The full listings for these programs could be obtained from Gitlab (https://gitlab.cas.mcmaster.ca/yaos4/thesis_code.git). The experiments were run on AMD Ryzen Threadripper 1950X 16-Core Processor (2.09GHz). All measurements were with Ubuntu 16.4 LTS in single-user mode. Each timing measurement is run thirty times. The results reported here are the average with a 95% confidence interval. The difference between the maximum and minimum value was small enough, so only the average value is reported. For Erlang, because there is a constant overhead (around 1000 ms) for the virtual machine to start and stop, the tests are repeated for 1000 times to amortize this overhead. For Java, the tests are repeated for 10 times to amortize this overhead. The software environments are as follows:

Component	Version
Operating System	Ubuntu 16.4 LTS
Gcc	5.4.0
Java HotSpot	1.8.0
Go	1.8
Erlang	20
GHC	8.4.2

6.4.1 Priority Queue

This experiment is an implementation of the example discussed in Section 3.5.5. This experiment aims to test the performance in a linear structure for different programming languages. A priority queue offers a method $add(e)$ for storing positive integer e, a method *remove* for removing the least integer stored so far, and a method *empty* for testing whether the priority queue is empty. This priority queue is implemented as a linked list. Elements are stored in field m in ascending order.

In this priority queue example, the *head* node is the data entrance. Each node has actions that would eventually insert or remove the element in the priority queue. These actions need to hold the locks on the current node and *next* node. Thus, the *insert* method and *remove* method can proceed concurrently in the different part of the queue (the gray nodes).

Test Program

Listing 6.16: Priority Queue Test Program in Lime

```
class Start
  var head: PQ
  var i: int
  var num: int
  init()
       num := getArg(1)
       setRand(num)
       head := new PQ()
       for i := 0 to num - 1 do
          head.add(getRand(i))
       for i := 0 to num - 1 do
          head.remove()
```

The test program in Listing 6.16 is a simple and multi-threaded program that first adds *num* elements to the priority queue and then removes all the elements. The *num* varies first from 10 to 80 and in the second test the *num* is from 1000 to 9000. The method $getRand(i)$ return the i-th random numbers which are generated by the method $setRand(num)$. All the implementations use the same random numbers.

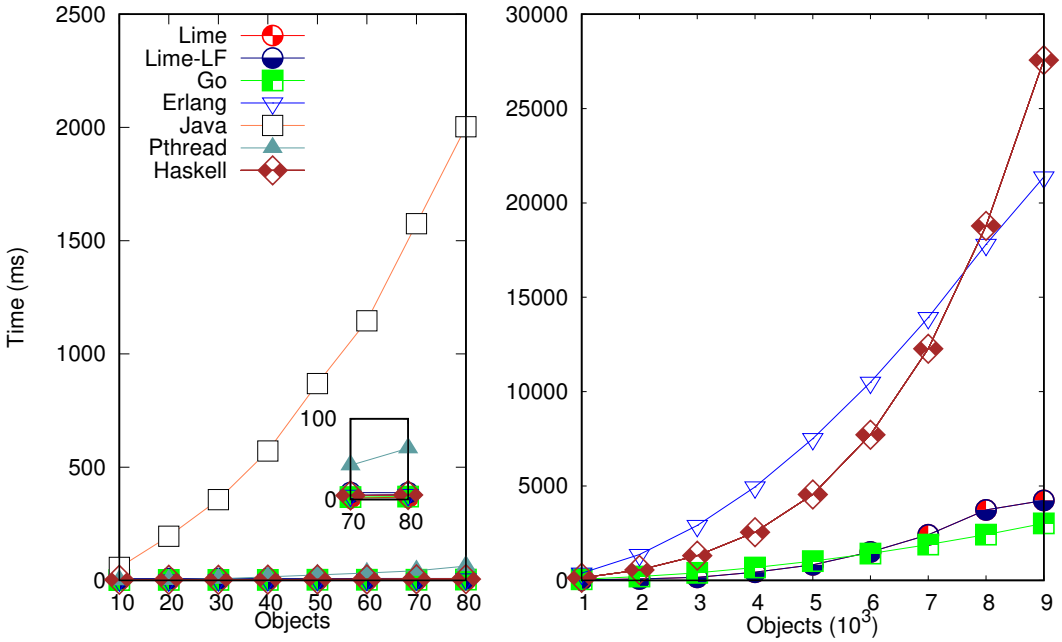

Figure 6.5: Priority Queue Results

Analysis

Figure 6.5 displays the execution time for adding and removing all of the elements. The small box in the left figure of Figure 6.5 is the zoomed graph when the number of the objects is from 70 to 80. The results show that the lightweight thread implementations, such as Go, Erlang, Haskell and Lime coroutines outperform the heavyweight thread implementations, such as Java thread and Pthread. Second, Go, Lime and Lime-LF outperform the other lightweight implementations.

First of all, the creation of a Lime coroutine only requires 8KB memory for the stack space. It becomes possible to run millions of coroutines even in a 32-bit address space. Second, Lime coroutine setup and teardown are efficient by creating the stacks in advance and reusing the stacks. The runtime system creates one coroutine per active object and maintains object pools for each worker threads, which makes the setup and teardown of a coroutine easy, requiring only a few lines of assembly code. Third, the most critical factor that affects system performance is the cost of stack switching. In the Lime coroutine implementation, coroutines are scheduled cooperatively, and when a stack switch occurs, only three registers need to be saved and

restored. The switch between Lime coroutines happens at well-defined points when an explicit call is made to the runtime scheduler. Places, where coroutines may yield back to the scheduler, are method guard evaluation, action guard evaluation, and lock acquirement.

6.4.2 MapReduce

MapReduce represents a programming model. A MapReduce program consists of a *Map* method which takes the input data, and a *Reduce* method takes the output data from the *Map* method and combines all the data until generating the final result.

In this MapReduce example, the *Mapper* nodes take the input data. Both *Mapper* and *Reducer* nodes would eventually call the *Reduce* method. Thus, the *Reduce* and *Map* methods can proceed concurrently in the different parts of the tree.

Test Program

Listing 6.17: MapReduce Test Program in Lime

```
class Reducer
   var index: int
   var next: Reducer
   var a1, a2: bool
   var e1, e2: int
   init(i: int, r: Reducer)
      index, a1, a2, next := i, false, false, r
   method reduce1(x: int)
      when not a1 do
         e1, a1 := x, true
   method reduce2(x: int)
      when not a2 do
         e2, a2 := x, true
   action doReduce
      when a1 and a2 do
         if index = 1 then
            print(e1 + e2)
            e1, e2 := 0, 0
         elif index % 2 = 0 then
            next.reduce1(e1 + e2)
         else
            next.reduce2(e1 + e2)
         a1, a2 := false, false
class Mapper
   var next: Reducer
   var a: bool
   var e, index: int
```

```
   init(i: int, r: Reducer)
      index, a, next := i, false, r
   method map(n: int)
      when not a do
         e, a := n, true
   action doMap
      when a do
         if index % 2 = 0 then
            next.reduce1(e * e)
         else
            next.reduce2(e * e)
         a := false


class Start
   var i: int
   var num: int
   var repeat: int
   var marray: array of Mapper
   var rarray: array of Reducer
   init()
      num := getArg(1)
      repeat := getArg(2)
      marray := new Mapper[num]
      rarray := new Reducer[num]
      rarray[0] := nil
      for i := 1 to num - 1 do
         rarray[i] := new Reducer(i, rarray[i / 2])
      for i := 0 to num - 1 do
         marray[i] := new Mapper(i, rarray[(i + num) / 2])
      while repeat > 0 do
         for i := 0 to num - 1 do
            marray[i].map(i)
         repeat := repeat - 1
```

The test program 6.17 computes the sum of squares by taking $map(x) = x^2$ and $reduce(x, y) = x + y$ for integers x, y. The input list is the integers from 0 to $num - 1$. The computation is repeated r times to "fill the pipeline". The output is the square pyramidal number of $num - 1$. The number of num is the power of 2, which varies first from 8 to 64 and in the second test is from 128 to 4096.

Analysis

Figure 6.6 displays the execution time for computing sum of squares. The small box in the left figure of Figure 6.6 is the zoomed graph when the number of the objects is

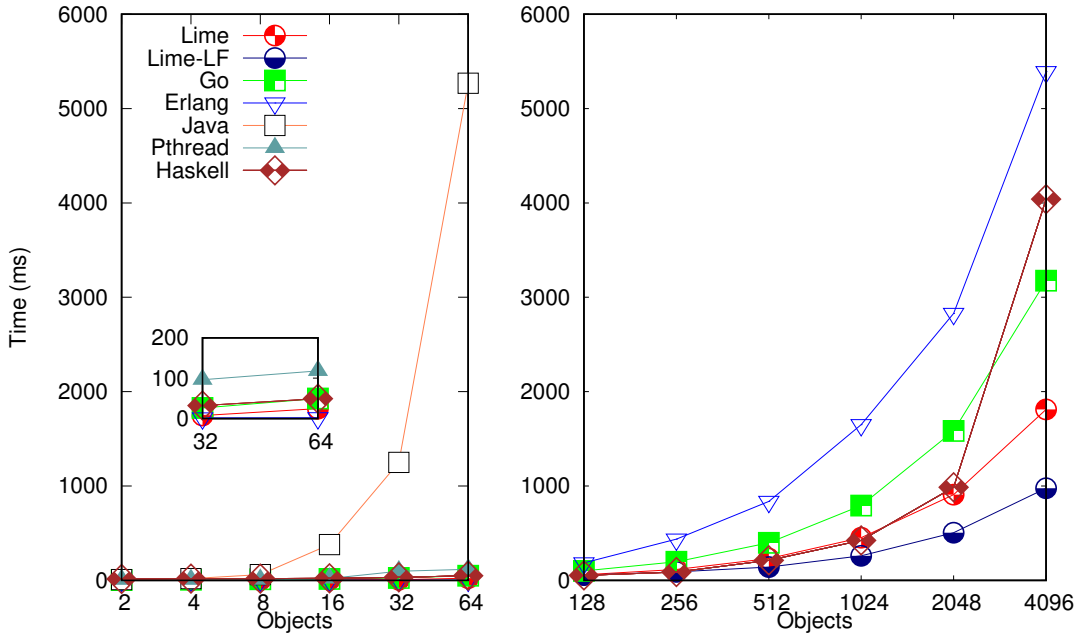

Figure 6.6: MapReduce Results

from 32 to 64. The results show that the lightweight thread implementations, such as Go, Erlang, Haskell and Lime coroutines outperform the heavyweight thread implementations, such as Java thread and Pthread. Second, Lime and Lime-LF continue to outperform the other lightweight implementations. Lime-LF has better performance than Lime because of the fast work distribution among worker threads.

6.4.3 Leaf-Oriented Tree

In the leaf-oriented tree example, the leaves contain all the elements stored, and internal nodes include only key values to guide the search. Insertion either creates two new leaves or only deposits a component of an internal node. Each node has an action that would eventually move the stored element one level closer to its final position.

For the leaf-oriented tree example, the *Root* is the only node which can get the input data. Although insertions can proceed concurrently in different parts of the tree, the real opportunity for the concurrency is low. Take 10000-leaf leaf-oriented tree as an example: in theory, 5000 parents can be executed concurrently. However, the *Root*

node is the "bottleneck". Suppose each node spends the same time in passing the data, and the tree is a complete tree, in this case, only 0.14% (14/10000) of nodes in the tree can be executed concurrently because the approximate depth of the tree is 14.

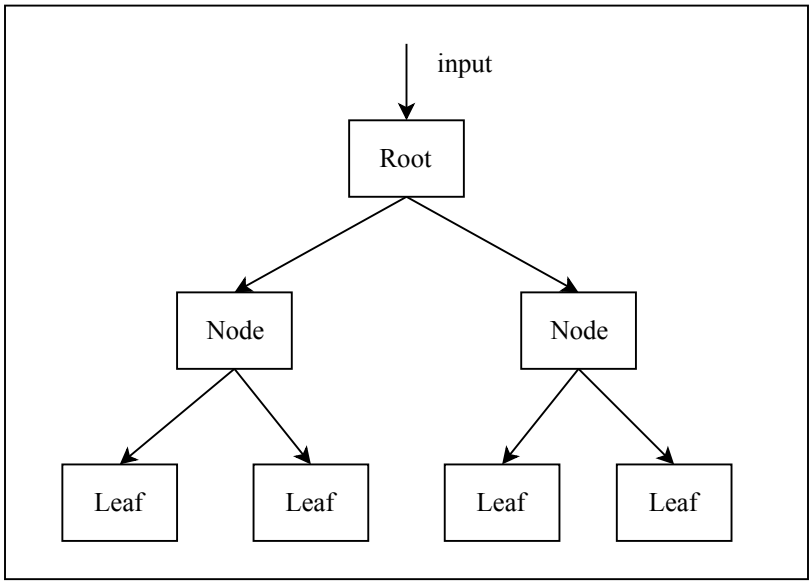

Figure 6.7: Leaf-Oriented Tree

Test Program

Listing 6.18: Leaf-oriented Tree Test Program in Lime

```
class Start
   var root: Node
   var num: int
   var i: int
   init()
      num := getArg(1)
      setRand(num)
      root := new Node(5000)
      for i := 0 to num - 1 do
         root.add(getRand(i))
      for i := 0 to num - 1 do
         if not root.has(getRand(i)) then
            root.add(getRand(i))
```

The test program in Listing 6.18 is a simple and multi-threaded program that first adds *num* random elements to the leaf-oriented tree and then searches *num* times. An

element is added to the tree if it does not already exist. The number of *num* varies first from 10 to 80 and in the second test the number of *num* is from 1000 to 9000. The method *getRand*(i) return the i-th random numbers which are generated by the method *setRand*(*num*). All the implementations use the same random numbers.

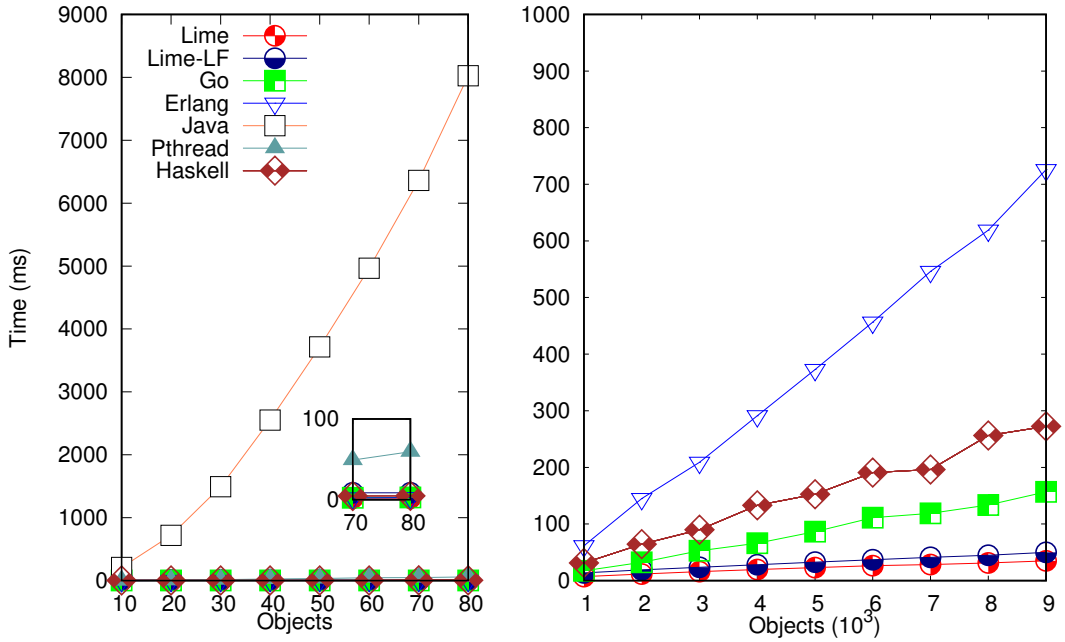

Figure 6.8: Leaf-oriented Tree Results

Analysis

Figure 6.8 displays the execution time for inserting data to the tree and searching for data from the tree. The small box in the left figure of Figure 6.8 is the zoomed graph when the number of the objects is from 70 to 80. The results show that the lightweight thread implementations, such as Go, Erlang, Haskell and Lime coroutines outperform the heavyweight thread implementations, such as Java thread and Pthread. Second, Lime and Lime-LF continue to beat the other lightweight implementations. However, Lime-LF does not outperform Lime in this experiment because there are around 14 enabled objects that existed in the system. In this case, the overhead of the lock-free queue becomes significant when there are a limited number of enabled objects which can be executed simultaneously.

6.4.4 The Santa Claus Problem

In 1994, Trono proposed the Santa Claus Problem as an exercise in concurrent programming (Trono, 1994):

> "Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise... This could also explain the quickness in their delivering of presents, since the reindeer can not wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh."

This problem requires an implementation of multi-party rendezvous with priority.

Test Program

In this section, we only discuss the result. The details of this Santa Claus Problem are in (Sekerinski and Yao, 2018). The main program in Listing 6.19 creates active objects for Santa, reindeer, and elves; these use the passive sleigh and shop objects for synchronization. The times are reported as the average real/user/system times of 20 runs. Only a single run was used for Java with 1,000,000 repetitions of Santa.

Listing 6.19: Santa Claus Problem Test Program in Lime

```
class Santa
   var s: {Sleeping, Harnessing, Riding, Welcoming, Consulting}
   var b: boolean
   var p: int
   init()
      s, b, p := Sleeping, false, 0
   method back()
      b := true
   method harness()
      when s = Harnessing do
```

```
            s := Riding
      method pull()
         when s = Riding do
            s, b := Sleeping, false
      method puzzled()
         p := 3
      method enter()
         when s = Welcoming do
            s := Consulting
      method consult()
         when s = Consulting do
            p := p - 1
            if p > 0 then
               s := Welcoming
            else
               s := Sleeping
      action action1
         when s = Sleeping and b do
            s := Harnessing
      action action2
         when s = Sleeping and p = 3 and not b do
            s := Welcoming

class Sleigh
      var s: {Back, Harnessing, Pulling}
      var c: int
      var st: Santa
      init(santa: Santa)
         s, c, st := Back, 9, santa
      method back()
         when s = Back do
            c := c - 1
            if c = 0 then
               s, c := Harnessing, 9
               st.back()
      method harness()
         when s = Harnessing do
            c := c - 1
            if c = 0 then
               s, c := Pulling, 9
               st.harness()
      method pull()
         when s = Pulling do
            c := c - 1
            if c = 0 then
```

```
            s, c := Back, 9
            st.pull()

class Reindeer
   var sl: Sleigh
   init (sleigh: Sleigh)
      sl := sleigh
   action action1
      sl.back()
      sl.harness()
      sl.pull()

class Shop
   var s: {Puzzled, Entering, Consulting}
   var c: int
   var st: Santa
   init(santa: Santa)
      s, c, st := Puzzled, 0, santa
   method puzzled()
      when s = Puzzled do
         c := c + 1
         if c = 3 then
            s := Entering
            st.puzzled()
   method enter()
      when s = Entering do
         s := Consulting
         st.enter()
   method consult()
      when s = Consulting do
         c := c - 1
         if c > 0 then
            s := Entering
         else
            s := Puzzled
         st.consult()

class Elf
   var sh: Shop
   init(shop: Shop)
      sh := shop
   action action1
      sh.puzzled()
      sh.enter()
      sh.consult()
```

Repetitions of Santa	Lime (guards)	C (semaphores)	Go (channels)	Java (monitors)
10,000	0.04/0.04/0.00	0.87/0.26/1.18	0.08/0.12/0.01	6.38/2.48/5.30
100,000	0.30/0.30/0.00	8.82/2.50/12.0	0.77/1.18/0.06	60.3/21.6/52.0
1,000,000	2.91/2.90/0.01	93.0/24.8/123	7.51/11.6/0.55	≈ 534/159/509

Table 6.2: The Santa Claus Problem Results: the average real/user/system times in seconds of 20 runs

```
class Start
   var st: Santa
   var sl: Sleigh
   var sh: Shop
   init()
      st := new Santa()
      sl := new Sleigh(st)
      sh := new Shop(st)
      for i := 1 to 9 do new Reindeer(sl)
      for i := 1 to 20 do new Elf(sh)
```

Analysis

Table 6.2 shows the running times for Santa with 9 reindeer and 20 elves. Santa's division of work is that for 10,000 rounds until retirement, he rides the sleigh 2,000 times and helps 8,000 times groups of three elves, or for 20 elves, each elf on average 1,200 times. For 100,000 and 1,000,000 rounds until Santa's retirement the ratio is the same. The results show that lightweight thread implementations, such as Go and Lime coroutines, outperform the heavyweight thread implementations, such as Java thread and Pthread. Secondly, at most 29 objects (9 reindeer and 20 elves) can execute concurrently. The overhead of the synchronization becomes the critical factor. Lime outperforms Go's implementation.

6.4.5 The Chameneos Game

The Chameneos game was proposed by Kaiser and Pradat-Peyre (2003) for comparing programming styles with Ada rendezvous, Java monitors, and C/Pthread semaphores. The Chameneos game contains one shared resource, the Mall object, which can be accessed by all the Chameneos objects. The benchmark can be regarded as a "sequential" concurrency benchmark.

> "Consider a population of N chameneos that have a cyclic behaviour. A chameneos usually lives lonely eating honeysuckle leaves in the training. After a while when feeling ready for competition, it enters a mall where a nice spring babbles and where it occasionally plays pall mall with another chameneos and possibly mutates before leaving the mall and returning in the forest. Given an initial

population, examine its evolution towards a final state in which all chameneos have the same colour, and therefore in which no one can mutate anymore. ”

Test Program

Listing 6.20: The Chameneos Game in Lime

```
class Chameneos
   var s: {InForest, DoneAtMall, WaitingAtMall}
   var col: int
   var mall: Mall
   init(c: int, m: Mall)
      col, mall, s:= c, m, InForest
   method meet(otherCol: int)
      if col != otherCol then col := 3 - col - otherCol
      s := DoneAtMall
   action GoingToMall
      when s = InForest do
         s := WaitingAtMall
         mall.arrive(this, col)
   action BackToForest
      when s = DoneAtMall do
         s := InForest

class Mall
   var s: {ZeroCham, OneCham, TwoCham, Done}
   var firstCol, sndCol: int
   var firstCham, sndCham: Chameneos
   var repeat, N: int
   init(arg: int)
      s, repeat, N := ZeroCham, 0, arg
   method arrive(ch: Chameneos, c: int)
      when s = ZeroCham or s = OneCham do
         if s = ZeroCham then
            firstCol, firstCham, s := c, ch, OneCham
         elif s = OneCham then
            sndCol, sndCham, repeat := c, ch, repeat + 1
            s := TwoCham
   action mutate
      when s = TwoCham do
         firstCham.meet(sndCol); sndCham.meet(firstCol)
         if repeat < N then s := ZeroCham
         else exit()

class Start
   var ma: Mall
```

```
var N, rounds: int
var i: int
init()
    N := getArg(1)
    rounds := getArg(2)
    ma := new Mall(N * rounds / 2)
    for i := 1 to N do new Chameneos(i % 3, ma)
```

The test program in Listing 6.20 is a multi-threaded program that creates one Mall and N chameneos. The number of *rounds* is set to 1000. The population of chameneos varies first from 10 to 90 and in the second test is from 1000 to 9000. We use Lime-LF in this benchmark.

Analysis

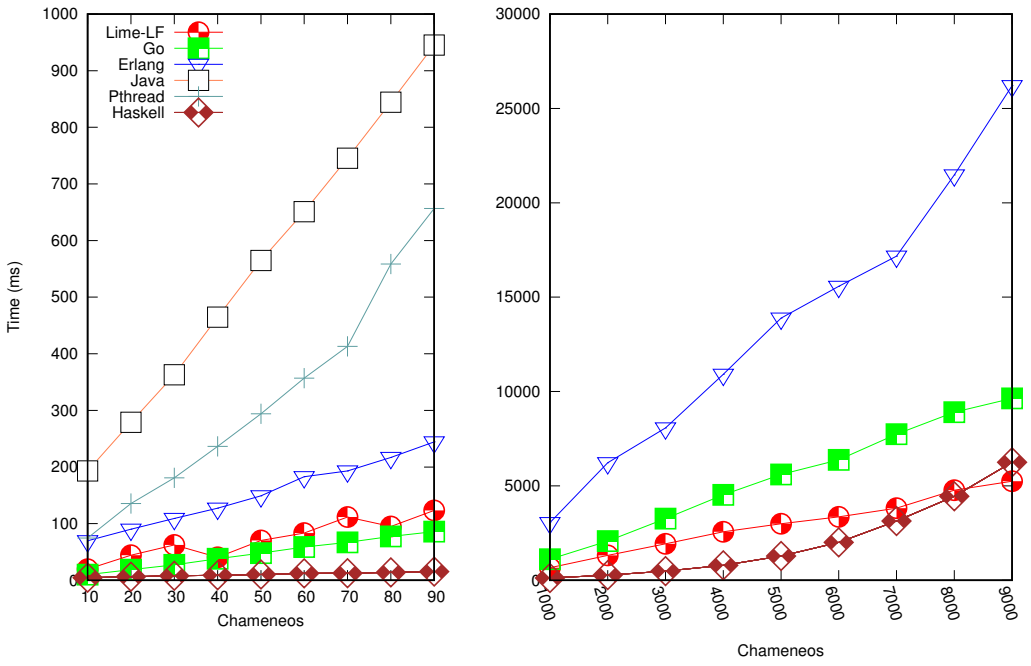

Figure 6.9: The Chameneos Game Results

Table 6.9 shows the running times for the chameneos game benchmark with one mall and N chameneos. Compared with the Santa Claus problem, this benchmark only allows at most two chameneos to enter the mall simultaneously and has even fewer concurrency

opportunities. The number of threads for Lime is first set to 2 and is increased to 3 in the second test. The results show that lightweight thread implementations, such as Go and Lime coroutines, continually outperform the heavyweight thread implementations, such as Java thread and Pthread. The overhead of the synchronization becomes the critical factor. The Haskell implementation only uses one core, and it outperforms the other implementations. Lime outperforms Go's implementation when the population of chameneos is from 1000 to 9000 and surpasses Haskell's implementation when the population of chameneos reaches 9000.

6.4.6 Summary

We compare the performance of Lime with the other three lightweight threads: Go, Erlang, and Haskell. The current implementation of Lime tends to perform better than the other programming languages' implementations in most cases. There are four different communication patterns discussed in this chapter: linear structure, tree structure with top-down dataflow, tree structure with bottom-up dataflow and star structure.

We implement a simple but efficient runtime system for Lime. The feature of garbage collection is not implemented in Lime for the sake of efficiency. More efficient memory models will be considered for Lime in future work. To support garbage collection, Go, Erlang and Haskell implement a large and complicated runtime system. Besides, cooperative scheduling is implemented in Lime. Erlang and Haskell implement preemptive scheduling, and the latest version of Go also implements preemptive scheduling by allowing the scheduler to send a POSIX signal to stop a running goroutine. Compared with cooperative scheduling, the preemptive scheduling introduces more context switch overhead during the execution of the program.

The priority queue benchmark has a linear structure. In principle, the adjacent nodes cannot be executed simultaneously. That is, at most 50% nodes could be executed concurrently. In this benchmark, Lime and Go are better than Haskell and Erlang. The implementation of Go outperforms Lime's implementation when the number of objects exceeds 6000.

The leaf-oriented tree benchmark has a tree structure with top-down dataflow. As we have discussed before, the leaf-oriented tree has a "bottleneck", and there are a limited number of objects that can be executed concurrently. We use the leaf-oriented tree benchmark to examine how efficient the runtime system is at giving priority to bottlenecks. In this benchmark, Lime outperforms all the other programming languages' implementations.

The MapReduce benchmark has a tree structure with bottom-up dataflow. All of the nodes in the MapReduce benchmark can be executed concurrently as long as "the pipeline is fully loaded". Lime and Lime-LF outperform the other lightweight implementations. Lime-LF has better performance than Lime because of the fast work distribution among worker threads.

The Santa Claus problem has an star structure, and Santa Claus is the center node. There are a limited number of objects that can be executed concurrently. In this benchmark, the overhead of the synchronization becomes a critical factor. The current implementation of Lime performs better than the other programming languages' implementations.

The chameneos game benchmark has fewer concurrency opportunities than the Santa

Claus problem. There are at most two chameneos that can be executed concurrently. The efficiency of the synchronization becomes a critical factor again. The current implementation of Lime outperforms the other implementations.

Chapter 7

Conclusions

Lime is an action-based object-oriented concurrent programming language. An object in Lime is a natural "unit" of concurrency, which provides programmers with a simple and unified design view for large concurrent programs. Besides, it is easier for the designers to reason about the code formally, or to prove the correctness of the concurrent programs.

To avoid the rollback mechanism or undesired restrictions, Lime allows that an action or method "gets stuck" at the point where a method is called. Shortcomings with the previous implementation of Lime include the overhead of the repeated guard evaluations and the inefficient implementation of Lime, including the guards translation scheme and the Lime runtime system.

This work takes a multi-pronged approach to addressing these concerns. First of all, every object being concurrent in principle can quickly lead to programs with thousands of threads. Stack overflow concern is addressed by proposing the guard-page stack mechanism, which does not introduce any overhead during the runtime if there is no stack overflow. This new stack mechanism starts with a small stack, grows as needed. It becomes possible to run thousands of threads even in a 32-bit address space.

Since every object is regarded as a natural "unit" of concurrency in Lime, mapping every object to a regular thread is not suitable because the overhead of context switching of the regular threads is too significant. Our implementation maps objects to user-level coroutines, instead of ordinary threads. Underneath the user-level coroutine, the Lime runtime system limits the number of worker threads to the number of CPUs. To reduce the cost of stack switching, Lime implements a fast stack switching mechanism which only needs to save and restore three registers.

The simple and unified design view in Lime stems from the combination of object-oriented concepts and action systems. To improve the performance, Lime implements a fast, cooperative scheduler and maintains local queues and a global queue. The method and action translation schemes and atomicity rules proposed in this work give the Lime runtime system the ability to execute the enabled objects efficiently.

Lime not only provides the programmers with a unified and straightforward OOP model for concurrency but also accomplishes a better performance than concurrent programming languages such as Erlang and Go, in fine-grained, highly concurrent benchmarks (Figure 6.5, Figure 6.6, Figure 6.8, Figure 6.9). This accomplishment comes from the guard commands

transaction scheme and the efficient runtime system.

Bibliography

Agha, G. A. (1985). Actors: a model of concurrent computation in distributed systems. Technical Report 844, MIT Cambridge Artificial Intelligence Lab. http://hdl.handle.net/1721.1/6952.

AMD (2019). AMD EPYC™. https://www.amd.com/en/products/cpu/amd-epyc-7h12. Accessed: 2020-04-23.

Andrews, G. R. (1991). *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company.

Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*, Volume 11. Addison-Wesley Reading.

Apt, K., F. S. De Boer, and E.-R. Olderog (2010). *Verification of sequential and concurrent programs*. Springer Science & Business Media. https://doi.org/10.1007/978-1-84882-745-5.

Apt, K. R. and E.-R. Olderog (2019). Fifty years of hoare's logic. *Formal Aspects of Computing 31*(6), 751–807. https://doi.org/10.1007/s00165-019-00501-3.

Armstrong, J., R. Virding, C. Wikström, and M. Williams (1996). *Concurrent Programming in ERLANG (2nd Edition)*. Prentice Hall International (UK) Ltd.

Asanovic, K., R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick (2009). A view of the parallel computing landscape. *Communications of ACM 52*(10), 56–67. https://doi.org/10.1145/1562764.1562783.

Back, R.-J., M. Büchi, and E. Sekerinski (1997). Action-based concurrency and synchronization for objects. In *International AMAST Workshop on Aspects of Real-Time Systems and Concurrent and Distributed Software*, pp. 248–262. Springer. https://doi.org/10.1007/3-540-63010-4_17.

Back, R.-J. and R. Kurki-Suonio (1989). Decentralization of process nets with centralized control. *Distributed Computing 3*(2), 73–87. https://doi.org/10.1007/BF01558665.

Baker, H. C. and C. Hewitt (1977). The incremental garbage collection of processes. In *ACM Sigplan Notices*, Volume 12, pp. 55–59. ACM. https://doi.org/10.1145/800228.806932.

Berris, D. M., A. Veitch, N. Heintze, E. Anderson, and N. Wang (2016). XRay: A function call tracing system. https://research.google/pubs/pub45287/.

Blake, G., R. G. Dreslinski, and T. Mudge (2009). A survey of multicore processors. *IEEE Signal Processing Magazine 26*(6), 26–37. https://doi.org/10.1109/MSP.2009.934110.

Bonsangue, M. M., J. N. Kok, and K. Sere (1998). An approach to object-orientation in action systems. In *International Conference on Mathematics of Program Construction*, pp. 68–95. Springer. https://doi.org/10.1007/BFb0054286.

Brinch Hansen, P. (1975). The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering SE-1*(2), 199–207. https://doi.org/10.1109/TSE.1975.6312840.

Briot, J.-P., R. Guerraoui, and K.-P. Lohr (1998). Concurrency and distribution in object-oriented programming. *ACM Computing Surveys (CSUR) 30*(3), 291–329. https://doi.org/10.1145/292469.292470.

Büchi, M. and E. Sekerinski (2000). A foundation for refining concurrent objects. *Fundamenta Informaticae 44*(1-2), 25–61. https://dl.acm.org/doi/10.5555/2372549.2372551.

Chandy, K. M. (1989). Parallel Program Design. In *Opportunities and Constraints of Parallel Computing*, pp. 21–24. Springer. https://link.springer.com/chapter/10.1007/978-1-4613-9668-0_6.

Chandy, K. M. and J. Misra (1988). *Parallel Program Design: A Foundation*. Addison-Wesley Longman.

Charles, J., P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova (2009). Evaluation of the Intel® Core™ i7 turbo boost feature. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 188–197. IEEE. https://doi.org/10.1109/IISWC.2009.5306782.

Cheney, D. (2014). Five things that make Go fast. https://dave.cheney.net/2014/06/07/five-things-that-make-go-fast/. Accessed: 2020-04-23.

Courtois, P.-J., F. Heymans, and D. L. Parnas (1971). Concurrent control with "readers" and "writers". *Communications of the ACM 14*(10), 667–668. https://doi.org/10.1145/362759.362813.

Cui, X.-L. (2009). An experimental implementation of action-based concurrency. Master's thesis, McMaster University. http://hdl.handle.net/11375/21409.

Deitel, H. M. (1990). *An Introduction to Operating Systems*. Addison-Wesley Longman Publishing Co., Inc.

Dijkstra, E. W. (1962). Over de sequentialiteit van procesbeschrijvingen (English). https://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html. circulated privately.

Dijkstra, E. W. (1967). The structure of the THE multiprogramming system. In *Proceedings of the First ACM Symposium on Operating System Principles*, pp. 10.1–10.6. ACM. https://doi.org/10.1145/800001.811672.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*(8), 453–457. https://doi.org/10.1145/360933.360975.

Dijkstra, E. W. (1986). A solution designed by A. Blokhuis. http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD979.PDF. circulated privately.

Dijkstra, E. W. (1987). Twenty-eight years. http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1000.PDF. circulated privately.

EETimes and Embedded (2019). 2019 embedded markets study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf. Accessed: 2020-05-23.

Faes, M. and T. R. Gross (2018). Concurrency-aware object-oriented programming with roles. *Proceedings of the ACM on Programming Languages 2*(130), 30. https://doi.org/10.1145/3276500.

Forgy, C. L. (2018). Ops5 user's manual. https://kilthub.cmu.edu/articles/OPS5_user_s_manual/6608090/1. Accessed: 2020-04-23.

Friedman, D. P. and D. S. Wise (1978). Aspects of Applicative Programming for Parallel Processing. *IEEE Transaction on Computers 27*(4), 289–296. https://dl.acm.org/doi/10.1109/TC.1978.1675100.

Geer, D. (2005). Chip makers turn to multicore processors. *Computer 38*(5), 11–13. https://doi.org/10.1109/MC.2005.160.

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam (1994). *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press.

GHC (2018). Glasgow Haskell Compiler User's Guide. https://downloads.haskell.org/ghc/latest/docs/html/users_guide/. Accessed:2020-04-28.

Google (2009). The Go programming language. https://golang.org/. Accessed: 2020-04-23.

Google (2018a). Escape analysis and Inlining. https://github.com/golang/go/wiki/CompilerOptimizations. Accessed: 2020-04-23.

Google (2018b). Source file src/runtime/runtime2.go. https://golang.org/src/runtime/runtime2.go. Accessed: 2020-04-23.

Google (2019). Goroutine Scheduler. https://github.com/golang/go/blob/master/src/runtime/proc.go. Accessed: 2020-04-23.

Harris, T. and K. Fraser (2003). Language support for lightweight transactions. *SIGPLAN Notices 38*(11), 388–402. https://doi.org/10.1145/949305.949340.

Heußner, A., C. M. Poskitt, C. Corrodi, and B. Morandi (2015). Towards practical graph-based verification for an object-oriented concurrency model. In *Proceedings of the First Workshop on Graphs as Models*, Volume 181 of *EPTCS*, pp. 32–47. https://doi.org/10.4204/EPTCS.181.3.

Hewitt, C. (1971). Procedural embedding of knowledge in Planner. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pp. 167–182. https://dl.acm.org/doi/10.5555/1622876.1622895.

Hoare, C. A. R. (1972). Towards a Theory of Parallel Programming. In *Operating Systems Techniques, Proceedings of Seminar at Queen's University. Also available in (Hoare, 2002)*, pp. 61–71. Academic Press.

Hoare, C. A. R. (1974). Monitors: An operating system structuring concept. *Communications of the ACM 17*(10), 549–557. https://doi.org/10.1145/355620.361161.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM 21*(8), 666–677. https://doi.org/10.1145/359576.359585.

Hoare, C. A. R. (2002). Towards a theory of parallel programming. In *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pp. 231–244. Springer-Verlag. https://dl.acm.org/doi/10.5555/762971.762978.

Holt, R. C. and J. R. Cordy (1985). The Turing Plus report. http://research.cs.queensu.ca/home/cordy/pub/downloads/tplus/Turing_Plus_Report.pdf.

Holt, R. C. and J. R. Cordy (1988). The Turing programming language. *Communications of the ACM 31*(12), 1410–1423. https://doi.org/10.1145/53580.53581.

IBM (2010). Language Environment Vendor Interfaces. http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss?CTY=US&FNC=SRX&PBL=SA22-7568-11. Accessed: 2020-04-23.

INMOS Limited (1984). *Occam Programming Manual*. Series in Computer Science. Prentice-Hall International.

Intel (2019). Intel®Xeon®Platinum 9282 Processor. https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html. Accessed: 2020-04-23.

Ishikawa, Y. and M. Tokoro (1984). The design of an object oriented architecture. *ACM SIGARCH Computer Architecture News 12*(3), 178–187. https://doi.org/10.1145/800015.808181.

ITRS (2007). International technology roadmap for semiconductors, system drivers. Semiconductor Industry Association. https://www.semiconductors.org/wp-content/uploads/2018/08/2007System-Drivers.pdf. Accessed: 2020-04-23.

Joshi, R. (1998). Seuss for Java — Language Reference. http://rjoshi.org/bio/papers/SeussForJava.pdf. Accessed: 2020-04-29.

Kaiser, C. and J. . Pradat-Peyre (2003). Chameneos, a concurrency game for java, ada and others. In *ACS/IEEE International Conference on Computer Systems and Applications, 2003. Book of Abstracts.* https://doi.org/10.1109/AICCSA.2003.1227495.

Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming Language*. Prentice Hall Inc.

Knuth, D. E. (1973). *Fundamental Algorithms: The Art of Computer Programming*. Addison-Wesley.

Koa, C. and C. Hwang (1987). A dietary recommendation expert system using OPS5. In *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pp. 658–663. IEEE Computer Society Press. https://dl.acm.org/doi/10.5555/42040.42144.

Krüger, I. H. (1996). An experiment in compiler design for a concurrent object-based programming language. Master's thesis, The University of Texas at Austin. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=55BFDAF3045A3757D449ED0337AAEB47?doi=10.1.1.30.2842&rep=rep1&type=pdf.

Kurki-Suonio, R. and H.-M. Järvinen (1989). Action system approach to the specification and design of distributed systems. *SIGSOFT Software Engineer Notes 14*(3), 34–40. https://doi.org/10.1145/75199.75205.

Lalis, S. and B. A. Sanders (1994). Adding concurrency to the Oberon system. In *Proceedings of the International Conference on Programming Languages and System Architectures*, pp. 328–344. Springer-Verlag New York. https://dl.acm.org/doi/10.5555/184716.184735.

Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*(3), 872–923. https://doi.org/10.1145/177492.177726.

Lea, D. (2000). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Professional.

Linux (2012). pthread_create man page, The Linux *man-pages* project. http://man7.org/linux/man-pages/man3/pthread_create.3.html. Accessed: 2020-04-23.

Linux (2018). malloc (3) - linux man pages. http://man7.org/linux/man-pages/man3/malloc.3.html. Accessed: 2020-04-23.

Liskov, B. (1988). Keynote address - data abstraction and hierarchy. *SIGPLAN Notices 23*(5), 17–34. https://doi.org/10.1145/62138.62141.

Lou, G. (2004). A compiler for an action-based object-oriented programming. Master's thesis, McMaster University. http://www.cas.mcmaster.ca/~emil/Students_files/Lou04ActionOOPL.pdf.

Marr, D. T., F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton (2002). Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal 6*, 12.

Meyer, B. (1997). *Object-Oriented Software Construction (2nd Edition)*. Prentice Hall.

Microsoft (2013). Thread Stack Size. http://msdn.microsoft.com/en-us/library/windows/desktop/ms686774(v=vs.85).aspx. Accessed: 2020-04-23.

Microsoft (2017). Virtual address spaces. https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces. Accessed: 2020-04-23.

Microsoft (2019). Asynchronous programming with async and await. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/. Accessed: 2020-04-23.

Middha, B., M. Simpson, and R. Barua (2008). MTSS: Multitask stack sharing for embedded systems. *ACM Transactions on Embedded Computing Systems 7*(4), 37. https://doi.org/10.1145/1376804.1376814.

Misra, J. (2001). A discipline of multiprogramming. In *A Discipline of Multiprogramming: A Programming Theory for Distributed Applications*, pp. 1–12. Springer. https://doi.org/10.1007/978-1-4419-8528-6.

Moore, B. J. (2010). Parallelism generics for Ada 2005 and beyond. In *ACM SIGAda Ada Letters*, Volume 30, pp. 41–52. ACM. https://doi.org/10.1145/1879063.1879078.

Moore-Oliva, J., E. Sekerinski, and Y. Shucai (2014). A comparison of scalable multi-threaded stack mechanisms. Technical report, McMaster University. http://www.cas.mcmaster.ca/cas/0reports/CAS-14-07-ES.pdf.

Moore-Oliva, J. I. (2010). A comparison of scalable multi-threaded stack mechanisms. Master's thesis, McMaster University. http://hdl.handle.net/11375/9011.

Mozilla (2019). Async function. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function. Accessed: 2020-04-23.

OPS5 (2013). OPS5 Production System. https://github.com/briangu/OPS5. Accessed: 2020-04-23.

Owicki, S. and D. Gries (1976). An axiomatic proof technique for parallel programs i. *Acta informatica 6*(4), 319–340.

Parr, T. J. (2004). Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, pp. 224–233. ACM. https://doi.org/10.1145/988672.988703.

Parr, T. J. (2012). Stringtemplate. http://www.stringtemplate.org/. Accessed: 2020-04-23.

Pizka, M. (1999). Thread segment stacks. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1964–1970. CSREA Press. http://www.oldlinux.org/Linux.old/study/sabre/os/files/Misc/Thread_Segment_Stacks.pdf.

Rupp, K. (2018). 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/. Accessed: 2020-04-23.

Sekerinski, E. (1996). Deriving control programs by weakest preconditions. Technical Report 4, Turku Centre for Computer Science.

Sekerinski, E. (2002). Concurrent object-oriented programs: From specification to code. In *International Symposium on Formal Methods for Components and Objects*, pp. 403–423. Springer. https://doi.org/10.1007/978-3-540-39656-7_17.

Sekerinski, E. (2003). A simple model for concurrent object-oriented programming. In *International Conference Internet, Processing, Systems, Interdisciplinaries, IPSI 2003*, Sveti Stefan, Montenegro, pp. 1–4.

Sekerinski, E. and S. Yao (2018). Refining santa: An exercise in efficient synchronization. In Proceedings *18th Refinement Workshop,* Oxford, UK, Volume 282 of *Electronic Proceedings in Theoretical Computer Science*, pp. 68–86. Open Publishing Association. https://doi.org/10.4204/EPTCS.282.6.

Shankar, A. (2003). Implementing Coroutines for .NET by wrapping the unmanaged Fiber API. https://docs.microsoft.com/en-us/archive/msdn-magazine/2003/september/implement-coroutines-for-net-by-wrapping-the-unmanaged-fiber-api. Accessed: 2020-04-23.

Siegel, S. F., M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers (2015). CIVL: The concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15. Association for Computing Machinery. https://doi.org/10.1145/2807591.2807635.

Silberschatz, A., P. B. Galvin, and G. Gagne (2014). *Operating System Concepts Essentials*. John Wiley & Sons, Inc.

Slotnick, D. L., W. C. Borck, and R. C. McReynolds (1962). The SOLOMON computer. In *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*, pp. 97–107. IEEE. https://doi.org/10.1145/1461518.1461528.

Sun Microsystems (2008). Multithreaded Programming Guide. http://docs.oracle.com/cd/E19253-01/816-5137/816-5137.pdf. Accessed: 2020-04-23.

SunSoft (1997). Java on Solaris 2.6-A White Paper. https://docs.oracle.com/cd/E19504-01/index.html. Accessed: 2020-04-23.

Talla, D. (1990). Notes on termination of OCCAM processes. *SIGPLAN Notices 25*(9), 17–24. https://doi.org/10.1145/101344.101348.

Tismer, C. (2000). Continuations and stackless Python. In *Proceedings of the 8th International Python Conference*. http://www.stackless.com/spcpaper.htm.

Trono, J. A. (1994). A new exercise in concurrency. *ACM SIGCSE Bulletin 26*(3), 8–10. https://doi.org/10.1145/187387.187391.

Van Rossum, G. and F. L. Drake (2011). *The Python Language Reference Manual*. Network Theory Ltd.

von Behren, R., J. Condit, F. Zhou, G. C. Necula, and E. Brewer (2003). Capriccio: scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, pp. 268–281. ACM. https://doi.org/10.1145/945445.945471.

Weber, V., C. Bekas, T. Laino, A. Curioni, A. Bertsch, and S. Futral (2014). Shedding light on lithium/air batteries using millions of threads on the bg/q supercomputer. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 735–744. IEEE. https://doi.org/10.1109/IPDPS.2014.81.

West, S., S. Nanz, and B. Meyer (2015). Efficient and reasonable object-oriented concurrency. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 734–744. ACM. https://doi.org/10.1145/2786805.2786822.

Whitby-Strevens, C. (1985). The transputer. *ACM SIGARCH Computer Architecture News 13*(3), 292–300. https://doi.org/10.1145/327070.327269.

Wilding, M. F. and D. A. Wood (2008). Heap and stack layout for multithreaded processes in a processing system. US Patent 744782. http://www.patentlens.net/patentlens/patent/US_7447829/en/.

Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM 20*(11), 822–823. https://doi.org/10.1145/359863.359883.

Wong, K.-F. and B. Dagevill (1994). Supporting thousands of threads using a hybrid stack sharing scheme. In *Proceedings of the 1994 ACM Symposium on Applied Computing*, pp. 493–498. ACM New York. https://doi.org/10.1145/326619.326822.

Appendix A

Priority Queue Examples

A.1 Erlang

```
; erlc -W pq.erl
; erl -noshell -s -run pq start -s init stop -inputdir ../
   InputData/ -extra 10

-module(pq).
-export([start/0, node/3]).
start()->
   Root = spawn(pq, node, [0, self(), 0]),
   {ok, [Path | _]} = init:get_argument(inputdir),
   [Arg1 | _] = init:get_plain_arguments(),
   File = string:concat(Path, Arg1),
   {ok, Binary} = file:read_file(File),
   Lines = string:tokens(erlang:binary_to_list(Binary), "\n"),
   Numbers = lists:map(fun(X) -> {Int, _} = string:to_integer(X
      ), Int end, Lines),
   add_num(Numbers, Root),
   remove(list_to_integer(Arg1), Root).

remove(0, _)->
   true;
remove(Num, Root)->
   Root ! remove,
   receive
      {remove_ret, Val, _}->
         io:format("~p~n", [Val])
   end,
   remove(Num-1, Root).

add_num([], _)->
```

```
   true;
add_num([F | Rest], Root)->
   Root ! {add, F},
   add_num(Rest, Root).

node(Val, From, To)->
   receive
      {add, Value_add}->
         if To == 0->
            node(Value_add, From, spawn(pq, node, [0, self(), 0
               ]));
         true->
            To ! {add, max(Val, Value_add)},
            node(min(Val, Value_add), From, To)
         end;
      remove->
         if To == 0->
            From ! {remove_ret, Val, 0};
         true->
            From ! {remove_ret, Val, self()},
            To ! remove,
            receive
               {remove_ret, V, To_new}->
                  node(V, From, To_new)
            end
         end
   end.
```

A.2 Go

```
// go build -o PQ_GO PQ.go
// ./PQ_GO 10 ../InputData/ 4

package main
import (
   "fmt"
   "runtime"
   "os"
   "strconv"
   "log"
   "bufio"
)
```

```
func link(left_input <- chan int, left_output chan <- int) {
   m := 0 // value of the node
   v := 0 // temp for receiving
   var right_output chan int
   var right_input chan int
   for {
      select {
         case v = <- left_input:
            if m == 0 {
               m = v
               right_output = make(chan int)
               right_input = make(chan int)
               go link(right_output, right_input)
            } else {
               if v < m {
                     right_output <- m
                     m = v
               } else {
                     right_output <- v
               }
            }
         case left_output <- m:
               m = <- right_input
      }
   }
}

func main() {
   if len(os.Args) != 4 {
      fmt.Printf("Usage: %s num inputdata_dir thread_num\n",
         os.Args[0])
      return
   }

   m := os.Args[1]
   num,err:= strconv.Atoi(m)
   file_name := os.Args[2] + "/"+m
   file,err := os.Open(file_name)
   if err!=nil {
      log.Fatal(err)
   }
   defer file.Close()
   var input [] int
   scanner := bufio.NewScanner(file)
   for scanner.Scan() {
```

```
      //fmt.Println(scanner.Text())
      tmp, err := strconv.Atoi(scanner.Text())
      if err != nil{
         log.Fatal(err)
      }
      input = append(input, tmp)
   }

   n := os.Args[3]
   if err != nil {
      fmt.Println(err)
      os.Exit(2)
   }
   thread_num,err := strconv.Atoi(n)
   runtime.GOMAXPROCS(thread_num)

   right_output := make(chan int)
   right_input := make(chan int)
   go link(right_output, right_input)
   for _, element := range input {
      //fmt.Println( index)
      right_output <- element
   }
   for i := 1; i<= num; i++{
      //<- right_input
      fmt.Println(<- right_input)
      //fmt.Println(i)
   }
   //fmt.Printf("Done \n" )
   return
}
```

A.3 Haskell

```
-- ghc -threaded -o PQ_HS --make -O -rtsopts PQ.hs
-- ./PQ_HS 10 ../InputData/

import Control.Concurrent
import System.Environment

data PQcmd = Add !Int | Remove | RemoveRet !Int | LastNode
```

```
pqnode :: (MVar PQcmd) -> (MVar PQcmd) -> (MVar PQcmd) -> (MVar
    PQcmd) -> Int -> IO ()
pqnode left_input left_output right_output right_input key = do
   cmd <- takeMVar left_input
   case cmd of
      Add x -> do
         putMVar right_output (Add (max key x))
         pqnode left_input left_output right_output right_input
             (min key x)
      Remove -> do
         putMVar left_output (RemoveRet key)
         putMVar right_output Remove
         k <- takeMVar right_input
         case k of
            RemoveRet r -> do
               pqnode left_input left_output right_output
                  right_input r
            LastNode -> do
               pqnode_lastnode left_input left_output

pqnode_lastnode :: (MVar PQcmd) -> (MVar PQcmd) -> IO ()
pqnode_lastnode left_input left_output = do
   right_input <- newEmptyMVar
   right_output <- newEmptyMVar
   cmd <- takeMVar left_input
   case cmd of
      Add a -> do
         forkIO (pqnode_lastnode right_output right_input)
         pqnode left_input left_output right_output right_input
             a
      Remove -> do
         putMVar left_output LastNode
         return ()

main = do
   args <- getArgs
   let path = (args!!1) ++ (args!!0)
   file <- readFile path
   let datas = lines file
   let nums = map read datas

   right_output <- newEmptyMVar
   right_input <- newEmptyMVar
   forkIO (pqnode_lastnode right_output right_input)
   mapM_ (\ x -> putMVar right_output (Add x)) nums
```

```
   mapM_ (\ x -> do
      putMVar right_output Remove
      y <- takeMVar right_input
      case y of
         RemoveRet r -> print r
      ) nums
   return ()
```

A.4 Java

```
// javac PQ.java
// java PQ 10 ../InputData/

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

class PQ extends Thread{
   private int m, p;
   private boolean a, r;
   private PQ next;
   private volatile boolean isRunning = true;
   PQ(){
      setDaemon(true) ;
      this.m = 0;
      this.p = 0;
      this.a = false;
      this.r = false;
   }

   public synchronized boolean empty(){
      while(this.r){
         try {
            wait();
         }
         catch (InterruptedException e) {}
      }
      return this.next == null;

   }

   public synchronized void add(int n) {
      while(this.a || this.r){
```

```
            try{
                wait();
            }
            catch (InterruptedException e) {}
        }
        if (this.next == null){
            this.m = n;
            this.next = new PQ();
            this.next.start();
        }else{
            this.p = n;
            this.a = true;
        }
        notifyAll();
    }

    public synchronized int remove() {
        while(this.a || this.r){
            try{
                wait();
            }
            catch (InterruptedException e) {}
        }
        this.r = true;
        notifyAll();
        return this.m;
    }


    private synchronized void doAdd(){
        if(this.a){
            if(this.m < this.p){
                next.add(this.p);
            }else{
                next.add(this.m);
                this.m = this.p;
            }
            this.a = false;
            notifyAll();
        }
    }
    private synchronized void doRemove(){
        if(this.r){
            if(this.next == null){
                this.r = false;
```

```
            return;
         }else if(this.next.empty()){
            this.next = null;
         }
         else{
            this.m = this.next.remove();
         }
         this.r = false;
         notifyAll();
      }
   }

   public void run(){
      while(true){
         doAdd();
         doRemove();
         //yield to the scheduler
         Thread.yield();
      }
   }

   /*public void kill() {
     isRunning = false;
   }*/
   public static void main(String[] args) {
      try {
         int i = 0;
         if(args.length < 2){
            System.err.println("Usage: java -cp ./bin/PQ num
               inputdata_dir");
            return;
         }

         int num = Integer.parseInt(args[0]);
         int [] input = new int[num];
         Scanner scanner = new Scanner(new File(args[1],args
            [0]));
         while (scanner.hasNextLine()) {
            //System.out.println(scanner.nextLine());
            input[i] = Integer.parseInt(scanner.nextLine());
            i++;
         }
         scanner.close();

         PQ head = new PQ();
```

```
            head.start();
            for (i = 0; i < num; i++){
               head.add(input[i]);
            }
            for (i = 0; i < num; i++){
               System.out.println(head.remove());
               //head.remove();
            }
         } catch (FileNotFoundException e) {
            e.printStackTrace();
         }

      }
```

A.5 Pthread

```
// gcc -pthread -o PQ_C PQ.c
// ./PQ_C 10 ../InputData/

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

typedef struct PriorityQueue{
   pthread_mutex_t lock;
   pthread_cond_t cv;
   struct PriorityQueue *next;
   int a, r, m, p;
   int num_actions;
   int (*actions[2])(void *);
}PriorityQueue;

int pq_doAdd(void * self);
int pq_doRemove(void *self);
void * pq_doActions(void * self);

void *pq_init(){
   pthread_t thread_id;
   PriorityQueue * pq = (PriorityQueue *)malloc(sizeof(
      PriorityQueue));
```

```
   if(pq == NULL){
      printf("Create priority queue failed\n");
      exit(1);
   }
   pthread_mutex_init(&pq->lock,0);
   pthread_cond_init(&pq->cv,0);
   pq->next = NULL;
   pq->a = 0;
   pq->r = 0;
   pq->m = 0;
   pq->p = 0;
   pq->num_actions = 2;
   pq->actions[0] = pq_doAdd;
   pq->actions[1] = pq_doRemove;
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
   pthread_create(&thread_id, &attr, pq_doActions, (void *)pq);
   return pq;
}

int pq_empty(PriorityQueue * self){
   int tmp;
   pthread_mutex_lock(&self->lock);
   while(self->r){
      pthread_cond_wait(&self->cv, &self->lock);
   }
   tmp = (self->next == NULL);
   pthread_mutex_unlock(&self->lock);
   return tmp;
}

int pq_remove(PriorityQueue * self){
   //when not a and not r
   int tmp;
   pthread_mutex_lock(&self->lock);
   while(self->a || self->r){
      pthread_cond_wait(&self->cv, &self->lock);
   }
   self->r = 1;
   pthread_cond_signal(&self->cv);
   tmp = self->m;
   pthread_mutex_unlock(&self->lock);
   return tmp;
}
```

```
void pq_add(int e, PriorityQueue * self){
   // when not a and not r
   pthread_mutex_lock(&self->lock);
   while(self->a || self->r){
      pthread_cond_wait(&self->cv, &self->lock);
   }
   if(self->next == NULL){
      self->m = e;
      self->next = pq_init();
   }else{
      self->p = e;
      self->a = 1;
      pthread_cond_signal(&self->cv);
   }
   pthread_mutex_unlock(&self->lock);
}

int pq_doAdd(void * self){
   //when a do
   int done;
   PriorityQueue *this = (PriorityQueue *)self;
   if(this->a){
      if(this->m < this->p){
         pq_add(this->p, this->next);
      }else {

         pq_add(this->m, this->next);
         this->m = this->p;
      }
      this->a = 0;
      pthread_cond_signal(&this->cv);
      done = 1;
   }else{
      done = 0;
   }
   return done;
}

int pq_doRemove(void * self){
   int done;
   PriorityQueue *this = (PriorityQueue *)self;
   if(this->r){
      if(this->next == NULL){
         this->r = 0;
```

```
            return 1;
        }else if(pq_empty(this->next)){
            this->next = NULL;
        }else {
            this->m = pq_remove(this->next);
        }
        this->r = 0;
        pthread_cond_signal(&this->cv);
        done = 1;
    }else{
        done = 0;
    }
    return done;
}

void * pq_doActions(void * self){
    int i = 0, done = 0;
    PriorityQueue *this = (PriorityQueue *)self;
    while(1){
        pthread_mutex_lock(&this->lock);
        for(i=0; i < this->num_actions; i++)
            done += this->actions[i](this);
        pthread_mutex_unlock(&this->lock);
        //yield to the scheduler
        pthread_yield();
    }
    return NULL;
}

int main(int argc, char* argv[]){
    if(argc < 3){
        printf("The Usage: %s num inputdata_dir\n", argv[0]);
        exit(1);
    }
    char *dir = argv[2];
    char filename[100];
    char * line = NULL;
    size_t len = 0;
    ssize_t read;
    snprintf(filename, 100, "%s/%s", dir, argv[1]);
    int num = atoi(argv[1]);
    FILE *file = fopen(filename, "r");
    if (file == NULL) exit(EXIT_FAILURE);
    int *input = (int *)malloc(sizeof(int)*num);
    int i = 0;
```

```
    while ((read = getline(&line, &len, file)) != -1) {
        input[i] = atoi(line);
        i++;
    }
    fclose(file);

    PriorityQueue * head = (PriorityQueue *)pq_init();
    for (i = 0; i < num; i++){
        pq_add(input[i], head);
    }

    for (i = 0; i < num; i++){
        //pq_remove(head);
        printf("%d\n", pq_remove(head));
    }
    return 0;
}
```

Appendix B

Leaf-oriented Tree Examples

B.1 Erlang

```
% erlc -W lot.erl
% erl -noshell -s -run lot start -s init stop -inputdir ../
   InputData/ -extra 10

-module(lot).
-export([start/0, node/3]).

start()->
   Root = spawn(lot, node, [5000,0,0]),
   {ok, [Path | _]} = init:get_argument(inputdir),
   [Arg1 | _] = init:get_plain_arguments(),
   File = string:concat(Path,Arg1),
   {ok, Binary} = file:read_file(File),
   Lines = string:tokens(erlang:binary_to_list(Binary), "\n"),
   Numbers = lists:map(fun(X) -> {Int, _} = string:to_integer(X
      ), Int end, Lines),
   add_num(Numbers, Root),
   Range = lists:seq(0, 10000),
   has_num(Range, Root).


has_num([H], Root)->
   Root !{has, H, self()},
   receive
      {has_result, 1}->
         io:format("~b ~n", [H]);
      {has_result, 0}->
         ok
      end;
```

```
has_num([H | Rest], Root)->
   Root !{has, H, self()},
   receive
      {has_result, 1}->
         io:format("~b ~n", [H]);
      {has_result, 0}->
         ok
      end,
   has_num(Rest, Root).

add_num([F], Root)->
   Root ! {add, F};
add_num([F | Rest], Root)->
   Root ! {add, F},
   add_num(Rest, Root).

node(Key, Left, Right)->
   receive
      {add, X}->
         if Left /= 0 ->
            if X =< Key ->
               Left ! {add, X},
               node(Key, Left, Right);
            true ->
               Right ! {add, X},
               node(Key, Left, Right)
            end;
         true->
            if X > Key ->
               node(Key, spawn(lot, node, [Key, 0, 0]), spawn(
                  lot, node, [X, 0, 0]));
            X < Key ->
               node(X, spawn(lot, node, [X, 0, 0]), spawn(lot,
                  node, [Key, 0, 0]));
            true ->
               node(Key, Left, Right)
            end
         end;
      {has, Y, Ret}->
         if Left == 0->
            if Key == Y->
               Ret ! {has_result, 1};
            true->
               Ret ! {has_result, 0}
```

```
            end,
            node(Key, Left, Right);
         true->
            if Y =< Key->
               Left ! {has, Y, Ret},
               node(Key, Left, Right);
            true->
               Right ! {has, Y, Ret},
               node(Key, Left, Right)
            end
         end
   end.
```

B.2 Go

```
// go build -o LOT_GO LOT.go
// ./LOT_GO 10 ../InputData/ 4

package main

import (
   "bufio"
   "fmt"
   "log"
   "os"
   "runtime"
   "strconv"
)

var found chan bool

func node(k int, parent chan int, find chan int) {
   key := k // value of the current node
   p := 0 // temp for receiving
   var leftchild_parent chan int
   var rightchild_parent chan int
   var leftchild_find chan int
   var rightchild_find chan int

   for {
      select {
      case p = <- parent:
         if leftchild_parent != nil {
```

```
            if p <= key {
               leftchild_parent <- p
            } else {
               rightchild_parent <- p
            }

         } else if p < key {
            leftchild_parent = make(chan int)
            leftchild_find = make(chan int)
            go node(p, leftchild_parent, leftchild_find)

            rightchild_parent = make(chan int)
            rightchild_find = make(chan int)
            go node(key, rightchild_parent, rightchild_find)
            key = p
         } else if p > key {
            leftchild_parent = make(chan int)
            leftchild_find = make(chan int)
            go node(key, leftchild_parent, leftchild_find)
            rightchild_parent = make(chan int)
            rightchild_find = make(chan int)
            go node(p, rightchild_parent, rightchild_find)
         }
      case p = <- find:
         if leftchild_parent == nil {
            found <- p == key
         } else if p <= key {
            leftchild_find <- p
         } else {
            rightchild_find <- p
         }
      }
   }
}

func main() {
   if len(os.Args) != 4{
      fmt.Printf("Usage: %s num inputdata_dir thread_num\n",
         os.Args[0])
      return
   }
   m := os.Args[1]
   num, err := strconv.Atoi(m)
   if err != nil {
      log.Fatal(err)
```

```
}
dir := os.Args[2]
filePath := dir + "/" + m
file, err := os.Open(filePath)
if err != nil {
   log.Fatal(err)
}
defer file.Close()
if err != nil {
   log.Fatal(err)
}
defer file.Close()
var input []int
scanner := bufio.NewScanner(file)
for scanner.Scan() {
   //fmt.Println(scanner.Text())
   tmp, err := strconv.Atoi(scanner.Text())
   if err != nil {
      log.Fatal(err)
   }
   input = append(input, tmp)
}
n := os.Args[3]
if err != nil {
   fmt.Println(err)
   os.Exit(2)
}
thread_num, err := strconv.Atoi(n)
runtime.GOMAXPROCS(thread_num)
root := make(chan int)
find := make(chan int)
found = make(chan bool)
go node(5000, root, find)

for i := 0; i < num; i++ {
   root <- input[i]
}
for j := 0; j <= 10000; j++ {
   find <- j
   if <- found {
      fmt.Printf("%d\n", j);
   }
}
//t1 := time.Now()
//fmt.Printf("Go impl: %v\n", t1.Sub(t0))
```

```
}
```

B.3 Haskell

```
-- ghc -threaded -o LOT_HS --make -O -rtsopts LOT.hs
-- ./LOT_HS 10 ../InputData/

import Control.Concurrent
import System.Environment
import qualified Data.Text as Text
import qualified Data.Text.IO as Text

data LOTchannel = LOTchannel !(MVar LOTcmd)
data LOTcmd = Add !Int | Search !Int | Found !Bool
  deriving (Show)

lotnode :: LOTchannel -> LOTchannel -> LOTchannel -> LOTchannel
    -> Int -> IO ()
lotnode parent_C@(LOTchannel parent) found left_child_C@(
   LOTchannel left) right_child_C@(LOTchannel right) key = do
   cmd <- takeMVar parent
   case cmd of
      Add a -> do
         if a > key
            then do
               putMVar right (Add a)
               lotnode parent_C found left_child_C
                  right_child_C key
            else if a < key
               then do
                  putMVar left (Add a)
                  lotnode parent_C found left_child_C
                     right_child_C key
               else do
                  lotnode parent_C found left_child_C
                     right_child_C key
      Search b -> do
         if b <= key
            then do
               putMVar left (Search b)
               lotnode parent_C found left_child_C
                  right_child_C key
            else do
```

```
                putMVar right (Search b)
                lotnode parent_C found left_child_C
                   right_child_C key


lotnode_leaf :: LOTchannel-> LOTchannel -> Int -> IO ()
lotnode_leaf parent_C@(LOTchannel parent) found_C@(LOTchannel
   found) key = do
   cmd <- takeMVar parent
   case cmd of
      Add a -> do
         if a /= key
            then do
               l_child <- newEmptyMVar
               r_child <- newEmptyMVar
               let left = LOTchannel l_child
               let right = LOTchannel r_child
               let smaller = min key a
               let larger = max key a
               forkIO lotnode_leaf left found_C smaller
               forkIO lotnode_leaf
                  right found_C larger
               lotnode parent_C found_C left right smaller
            else do
               lotnode_leaf parent_C found_C key
      Search b -> do
         if b == key
            then do
               putMVar found (Found True)
            else do
               putMVar found (Found False)
         lotnode_leaf parent_C found_C key

f :: [String] -> [Int]
f = map read

main = do
   args <- getArgs
   let path = (args!!1) ++ (args!!0)
   file <- readFile path
   let datas = lines file
   let nums = f datas

   r <- newEmptyMVar
   f <- newEmptyMVar
   let root = LOTchannel r
```

```
   let find = LOTchannel f
   forkIO (lotnode_leaf root find 5000)
   mapM_ (\ x -> putMVar r (Add x)) nums
   mapM_ (\ x -> do
      putMVar r (Search x)
      y <- takeMVar f
      case y of
        Found True -> do
           print x
        _ -> return ()
    ) [0..10000]
   return ()
```

B.4 Java

```
// javac LOT.java
// java LOT 10 ../InputData/

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
class LOT extends Thread{
   private int key, p;
   private boolean a;
   public LOT right, left;

   LOT(int i){
      setDaemon(true) ;
      this.key = i;
      this.p = 0;
      this.left = null;
      this.right = null;
      this.a = false;
   }

   public synchronized void add(int x){
      while(this.a){
         try{wait();} catch (InterruptedException e) {}
      }
      if(this.left != null){
         this.a = true;
         this.p = x;
      }else if(x < this.key){
```

```
            this.left = new LOT(x);
            this.left.start();
            this.right = new LOT(key);
            this.right.start();
            this.key = x;
        }else if(x > this.key){
            this.left = new LOT(key);
            this.left.start();
            this.right = new LOT(x);
            this.right.start();
        }
    }

    public synchronized boolean has(int x){
        while(this.a){
            try{wait();} catch (InterruptedException e) {}
        }
        if(this.left == null) return x == this.key;
        else if(x <= this.key) return left.has(x);
        else return right.has(x);
    }

    private synchronized void doAdd(){
        if(this.a == true){
            if(this.p <= this.key) { left.add(this.p);}
            else right.add(this.p);
            this.a = false;
            notifyAll();
        }
    }
    public void run(){
        while(true){
            doAdd();
            Thread.yield();
        }
    }

    public static void main(String[] args) {
        int i = 0;
        int num = Integer.parseInt(args[0]);
        int [] input = new int[num];
        try {
            Scanner scanner = new Scanner(new File(args[1],args
                [0]));
            while (scanner.hasNextLine()) {
```

```
            //System.out.println(scanner.nextLine());
            input[i] = Integer.parseInt(scanner.nextLine());
            i++;
         }
         scanner.close();
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      }

      LOT root = new LOT(5000);
      root.start();
      for(i = 0; i < num; i++){
         root.add(input[i]);
      }

      for(i = 0; i <= 10000; i++){
         if(root.has(i)){
            System.out.println(i);
         }
      }
      return;
   }
}
```

B.5 Pthread

```
// gcc -pthread -o LOT_C LOT.c
// time ./LOT_C 80 ../InputData

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>
#include <errno.h>

typedef struct Node{
   pthread_mutex_t lock;
   pthread_cond_t cv;
   int key, p, a;
   struct Node *left;
   struct Node *right;
```

```
   int num_actions;
   void (*actions[1])(void *);
}node_t;

void * LOT_doActions(void * self);
void LOT_doAdd(void * this);

struct Node * LOT_init(int x){
   pthread_t thread_id;
   struct Node * tmp = (struct Node *)malloc(sizeof(struct Node
      ));
   if(tmp == NULL){
      printf("Create Node failed\n");
      exit(1);
   }
   pthread_mutex_init(&tmp->lock,0);
   pthread_cond_init(&tmp->cv,0);
   tmp->key = x;
   tmp->left = NULL;
   tmp->right = NULL;
   tmp->a = 0;
   tmp->num_actions = 1;
   tmp->actions[0] = LOT_doAdd;
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
   pthread_create(&thread_id, &attr, LOT_doActions, (void *)tmp
      );
   return tmp;
}

void LOT_add(int x, struct Node * self){
   pthread_mutex_lock(&self->lock);
   while(self->a){
      pthread_cond_wait(&self->cv, &self->lock);
   }
   if(self->left != NULL){
      self->a = 1;
      self->p = x;
   }else if(x < self->key){
      self->left = LOT_init(x);
      self->right = LOT_init(self->key);
      self->key = x;
   }else if(x > self->key){
      self->left = LOT_init(self->key);
```

```
      self->right = LOT_init(x);
   }
   pthread_mutex_unlock(&self->lock);
}

int LOT_has(int x, struct Node * self){
   int ret = 0;
   pthread_mutex_lock(&self->lock);
   while(self->a){
      pthread_cond_wait(&self->cv, &self->lock);
   }
   if(self->left == NULL){
      ret = (x == self->key);
   }else if(x <= self->key){
      ret = LOT_has(x, self->left);
   }else {
      ret = LOT_has(x, self->right);
   }
   pthread_mutex_unlock(&self->lock);
   return ret;
}

void LOT_doAdd(void * self){
   struct Node * this = (struct Node *)self;
   if(this->a){
      if(this->p <= this->key){
         LOT_add(this->p, this->left);
      }else{
         LOT_add(this->p, this->right);
      }
      this->a = 0;
      pthread_cond_signal(&this->cv);
   }
}

void * LOT_doActions(void * self){
   struct Node * this = (struct Node *)self;
   while(1){
      pthread_mutex_lock(&this->lock);
      for(int i = 0; i < this->num_actions; i++)
         this->actions[i](this);
      pthread_mutex_unlock(&this->lock);
      pthread_yield();
   }
   return NULL;
```

```
}

int main(int argc, char* argv[]){
   if(argc < 3){
      printf("The Usage: %s pq_num inputdir\n", argv[0]);
      exit(1);
   }
   char *dir = argv[2];
   char filename[100];
   char * line = NULL;
   size_t len = 0;
   ssize_t read;
   snprintf(filename, 100, "%s/%s", dir, argv[1]);
   int num = atoi(argv[1]);
   FILE *file = fopen(filename, "r");
   if (file == NULL) exit(EXIT_FAILURE);
   int *input = (int *)malloc(sizeof(int)*num);
   int i = 0;
   while ((read = getline(&line, &len, file)) != -1) {
      input[i] = atoi(line);
      i++;
   }
   fclose(file);

   struct Node * root = LOT_init(5000);

   for(i = 0; i < num; i++){
      LOT_add(input[i], root);
   }

   for(i = 0; i <= 10000; i++){
      if(LOT_has(i, root) == 1){
         printf("%d\n", i);
      }
   }
   return 0;
}
```

Appendix C

MapReduce Examples

C.1 Erlang

```
% erlc -W mr.erl
% erl -noshell -s -run mr initial 1 1 16 -s init stop
-module(mr).

-export([initial/1, rep/2, start/1, start/2]).

initial([RR, R, N])->
   rep(list_to_integer(RR), [R,N]).

rep(1, [R, N])->
   start([R, N]);
   %io:format("Done ~n", []);
rep(RR, [R, N])->
   start([R, N]),
   rep(RR - 1, [R, N]).

start([R, N]) ->
   Result = start(list_to_integer(R), list_to_integer(N)),
   io:format("~B~n", [hd(Result)]).

start(R, N) ->
   Self = self(),
   Reducer = start(Self, R, 1, N),
   [receive {Reducer, Result} -> Result end || _ <- lists:seq
      (1, R)].

start(Parent, R, N, N) ->
   spawn_link(fun() -> mapper(Parent, R, N) end);
start(Parent, R, From, To) ->
```

```
   spawn_link(fun() -> reducer(Parent, R, From, To) end).

mapper(Parent, R, N) ->
   [Parent ! {self(), N * N} || _ <- lists:seq(1, R)].

reducer(Parent, R, From, To) ->
   Self = self(),
   Middle = (From + To) div 2,
   A = start(Self, R, From, Middle),
   B = start(Self, R, Middle + 1, To),
   [Parent ! {Self, receive {A, X} -> receive {B, Y} -> X + Y
      end end} || _ <- lists:seq(1, R)].
```

C.2 Go

```
// go build -o MR_GO MR.go
// ./MR_GO 16 1 4

package main

import (
   "fmt"
   "os"
   "runtime"
   "strconv"
   //"time"
)

func mapper(in chan int, out chan int) {
   for v := range in {
      out <- v * v
   }
}

func reducer(in1, in2 chan int, out chan int) {
   for i1 := range in1 {
      i2 := <- in2
      out <- i1 + i2
   }
}

func main() {
   if len(os.Args) != 4{
```

```
    fmt.Printf("Usage: %s num repeat thread_num\n", os.Args
        [0])
    return
}
num := os.Args[1]
N, errw := strconv.Atoi(num)
if errw != nil {
    fmt.Println(errw)
    os.Exit(2)
}
R, errr := strconv.Atoi(os.Args[2]) // number of repetitions
if errr != nil {
    fmt.Println(errr)
    os.Exit(2)
}
// strconv.FormatInt(n, 2)
worker_num, _ := strconv.Atoi(os.Args[3])
runtime.GOMAXPROCS(worker_num)
//start := time.Now()
//const N = 1 << 10 // calculate P(N), N = 2 ** 10

r := make([]chan int, N * 2)
for i := range r {
    r[i] = make(chan int)
}
//var m [N]chan int
m := make([]chan int, N)
for i := range m {
    m[i] = make(chan int)
}
for i := 0; i < N; i++ {
    go mapper(m[i], r[i + N])
}
for i := 1; i < N; i++ {
    go reducer(r[i * 2], r[i * 2 + 1], r[i])
}

go func() {
    for j := 0; j < R; j++ {
        for i := 0; i < N; i++ {
            m[i] <- i + 1
        }
    }
}()
```

```
    for j := 0; j < R; j++ {
        fmt.Println(<- r[1])
        //fmt.Println(<- r[1], (N * (N + 1) * (2 * N + 1)) / 6)
    }
    //fmt.Println(time.Since(start))
}
```

C.3 Haskell

```
-- ghc -threaded -O -rtsopts --make -o MR_HS MR.hs
-- ./MR_HS 16 1

import Control.Concurrent
import Control.Monad
import System.Environment

mapper :: MVar Int -> MVar Int -> IO ()
mapper left right = do
    v <- takeMVar left
    putMVar right (! v * v)
    mapper left right

reducer :: MVar Int -> MVar Int -> MVar Int -> IO ()
reducer left_1 left_2 right = do
    v1 <- takeMVar left_1
    v2 <- takeMVar left_2
    putMVar right (! v1+v2)
    reducer left_1 left_2 right

repeats:: Int -> [MVar Int] -> [MVar Int] ->IO ()
repeats 0 m r = do
    mapM_ (\ (e,x) -> putMVar e (x+1)) (zip m [0..])
    result <- takeMVar (r!!1)
    putStrLn (show result)
    return ()

repeats n m r = do
    mapM_ (\ (e,x) -> putMVar e (x+1)) (zip m [0..])
    result <- takeMVar (r!!1)
    repeats (n-1) m r

main = do
    args <- getArgs
```

```
   let num = read (args!!0)
   let reps = read (args!!1)
   m <- replicateM num newEmptyMVar
   r <- replicateM (num*2) newEmptyMVar

   mapM_ (\ (mapMVar, x) -> forkIO (mapper mapMVar (r!!(num+x)
      ))) (zip m [0..])
   mapM_ (\ x -> forkIO (reducer (r!!(x*2)) (r!!(x*2+1)) (r!!x)
      )) [1..(num-1)]
   repeats (reps - 1) m r
```

C.4 Java

```
// javac MR.java
// java MR 64 1

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
class Reducer extends Thread{
   private int e1, e2, index;
   private boolean a1, a2;
   public Reducer next;
   Reducer(int idx){
      setDaemon(true) ;
      this.index = idx;
      this.e1 = this.e2 = 0;
      this.a1 = this.a2 = false;
      this.next = null;
   }

   public synchronized void reduce1(int input){
      while(this.a1){
         try{wait();} catch (InterruptedException e) {}
      }
      this.e1 = input;
      this.a1 = true;
   }
   public synchronized void reduce2(int input){
      while(this.a2){
         try{wait();} catch (InterruptedException e) {}
      }
      this.e2 = input;
```

```
      this.a2 = true;
   }

   private synchronized void doreduce(){
      if(this.a1 && this.a2){
         if(this.index == 1){
            System.out.printf("%d\n", e1 + e2);
            e1 = e2 = 0;
         }else{
            if(this.index % 2 == 0){
               this.next.reduce1(e1 + e2);
            }else{
               this.next.reduce2(e1 + e2);
            }
         }
         this.a1 = false;
         this.a2 = false;
         notifyAll();
      }
   }

   public void run(){
      while(true){
         doreduce();
         Thread.yield();
      }
   }
}

class Mapper extends Thread{
   private int e, index;
   private boolean a;
   public Reducer next;
   Mapper(int idx){
      setDaemon(true) ;
      this.index = idx;
      this.e = 0;
      this.a = false;
      this.next = null;
   }
   public synchronized void map(int input){
      while(this.a){
         try{wait();} catch (InterruptedException e) {}
      }
      this.e = input;
```

```
      this.a = true;
   }

   private synchronized void domap(){
      if(this.a){
         if(this.index % 2 == 0){
            this.next.reduce1(this.e * this.e);
         }else{
            this.next.reduce2(this.e * this.e);
         }
         this.a = false;
         notifyAll();
      }
   }

   public void run(){
      while(true){
         domap();
         // yield to the scheduler
         Thread.yield();
      }
   }
}

class MR{
   public Mapper[] m ;
   private Reducer[] r;
   int N;
   MR(int x){
      N = x;
      m = new Mapper[N];
      r = new Reducer[N];
   }
   private void MR_init(){
      m[0] = new Mapper(0);
      for(int i = 1; i < N; ++i){
         m[i] = new Mapper(i);
         r[i] = new Reducer(i);
      }
      for(int i = 1; i < N / 2; ++i){
         r[i * 2].next = r[i];
         r[i * 2 + 1].next = r[i];
      }
      for(int i = 0; i < N; ++i){
         m[i].next = r[(i + N) / 2];
```

```
      }
      for(int i = 0; i < N; i++){m[i].start();}
      for(int i = 1; i < N; i++){r[i].start();}
   }

   public static void main(String[] args) {
      if(args.length < 2){
         System.err.println("Usage: java MR num repeat\n");
         return;
      }
      int N = Integer.parseInt(args[0]);
      int repeat = Integer.parseInt(args[1]);
      int i;
      MR tree = new MR(N);
      tree.MR_init();
      while(repeat > 0){
         for(i = 1; i <= N; i++){
            tree.m[i - 1].map(i);
         }
         repeat--;
      }
      try {
         Thread.sleep(2000);
      } catch (InterruptedException e){}
      return;
   }
}
```

C.5 Pthread

```
// gcc -o MR_C -pthread MR.c
// ./MR_C 64 1

#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

typedef struct Reducer{
   pthread_mutex_t lock;
```

```
   pthread_cond_t cv;
   int index, a1, a2, e1, e2;
   struct Reducer *next;
   int num_actions;
   void (*actions[1])(void *);
}reducer_t;

typedef struct Mapper{
   pthread_mutex_t lock;
   pthread_cond_t cv;
   int index, e, a;
   struct Reducer *next;
   int num_actions;
   void (*actions[1])(void *);
}mapper_t;

typedef struct Tree{
   mapper_t ** m;
   reducer_t ** r;
}tree_t;

void Reducer_doReduce(void * this);
void Mapper_doMap(void * this);
void * Reducer_doActions(void * self);
void * Mapper_doActions(void * self);

struct Reducer * Reducer_init(int e){
   pthread_t thread_id;
   struct Reducer *tmp = (struct Reducer *)malloc(sizeof(struct
       Reducer));
   if(tmp == NULL){
      printf("Create reducer failed\n");
      exit(1);
   }
   pthread_mutex_init(&tmp->lock, 0);
   pthread_cond_init(&tmp->cv, 0);
   tmp->index = e;
   tmp->a1 = 0;
   tmp->a2 = 0;
   tmp->e1 = 0;
   tmp->e2 = 0;
   tmp->next = NULL;
   tmp->num_actions = 1;
   tmp->actions[0] = Reducer_doReduce;
```

```
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
    pthread_create(&thread_id, &attr, Reducer_doActions, (void
        *)tmp);
    return tmp;
}

struct Mapper * Mapper_init(int e){
    pthread_t thread_id;
    struct Mapper * tmp = (struct Mapper *)malloc(sizeof(struct
        Mapper));
    if(tmp == NULL){
        printf("Create mapper failed\n");
        exit(1);
    }
    pthread_mutex_init(&tmp->lock, 0);
    pthread_cond_init(&tmp->cv, 0);
    tmp->index = e;
    tmp->e = 0;
    tmp->a = 0;
    tmp->next = NULL;
    tmp->num_actions = 1;
    tmp->actions[0] = Mapper_doMap;

    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
    pthread_create(&thread_id, &attr, Mapper_doActions, (void *)
        tmp);
    return tmp;
}

void Reducer_reduce1(int e, struct Reducer * self){
    pthread_mutex_lock(&self->lock);
    while(self->a1){
        pthread_cond_wait(&self->cv, &self->lock);
    }
    self->e1 = e;
    self->a1 = 1;
    pthread_mutex_unlock(&self->lock);
}

void Reducer_reduce2(int e, struct Reducer *self){
    pthread_mutex_lock(&self->lock);
```

```
   while(self->a2){
      pthread_cond_wait(&self->cv, &self->lock);
   }
   self->e2 = e;
   self->a2 = 1;
   pthread_mutex_unlock(&self->lock);
}

void Reducer_doReduce(void * this){
   struct Reducer * self = (struct Reducer *)this;
   if(self->a1 && self->a2){
      if(self->index == 1){
         printf("%d\n", self->e1 + self->e2);
         self->e1 = 0;
         self->e2 = 0;
      }else{
         if(self->index % 2 == 0){
            Reducer_reduce1(self->e1 + self->e2, self->next);
         }else{
            Reducer_reduce2(self->e1 + self->e2, self->next);
         }
      }
      self->a1 = 0;
      self->a2 = 0;
      //pthread_cond_signal(&self->cv);
      pthread_cond_broadcast(&self->cv);
   }
}

void * Reducer_doActions(void * self){
   int i = 0;
   struct Reducer *this = (struct Reducer *)self;
   while(1){
      pthread_mutex_lock(&this->lock);
      for(i=0; i < this->num_actions; i++)
         this->actions[i](this);
      pthread_mutex_unlock(&this->lock);
      //yield to the scheduler
      pthread_yield();
   }
   return NULL;
}

void Mapper_map(int e, struct Mapper * self){
   pthread_mutex_lock(&self->lock);
```

```
   while(self->a){
      pthread_cond_wait(&self->cv, &self->lock);
   }
   self->e = e;
   self->a = 1;
   pthread_mutex_unlock(&self->lock);
}

void Mapper_doMap(void * self){
   struct Mapper * this = (struct Mapper *)self;
   if(this->a){
      if(this->index % 2 == 0){
         Reducer_reduce1(this->e * this->e, this->next);
      }else{
         Reducer_reduce2(this->e * this->e, this->next);
      }
      this->a = 0;
      pthread_cond_signal(&this->cv);
   }
}

void * Mapper_doActions(void * self){
   int i;
   struct Mapper * this = (struct Mapper *)self;
   while(1){
      pthread_mutex_lock(&this->lock);
      for(i=0; i<this->num_actions; i++)
         this->actions[i](this);
      pthread_mutex_unlock(&this->lock);
      //yield to the scheduler
      //pthread_yield();
   }
   return NULL;
}

tree_t * tree_init(int object_num){
   tree_t * n = (tree_t *) malloc (sizeof(tree_t));
   n->m = (mapper_t **)malloc(sizeof(mapper_t *) * object_num);
   n->r = (reducer_t **)malloc(sizeof(reducer_t *) * object_num
      );
   n->m[0] = (mapper_t *)Mapper_init(0);
   for(int i = 1; i < object_num; ++i){
      n->m[i] = (mapper_t *)Mapper_init(i);
      n->r[i] = (struct Reducer *)Reducer_init(i);
   }
```

```
   for(int i = 1; i < object_num / 2; ++i){
      n->r[i * 2]->next = n->r[i];
      n->r[i * 2 + 1]->next = n->r[i];
   }
   for(int i = 0; i< object_num; ++i){
      n->m[i]->next = n->r[(i + object_num) / 2];
   }
   return n;
}

int main(int argc, char* argv[]){
   if(argc < 3){
      printf("The Usage: %s num repeat\n", argv[0]);
      exit(1);
   }
   int i;
   int num = atoi(argv[1]);
   int repeat = atoi(argv[2]);
   tree_t *t = tree_init(num);
   while(repeat > 0){
      for(i = 1; i <= num; i++){
         Mapper_map(i, t->m[i - 1]);
      }
      repeat--;
   }
   sleep(2);
   return 0;
}
```

Appendix D

The Chameneos Game Benchmark

D.1 Erlang

```
-module(chameneos).
-export([start/0]).

-import(lists, [foreach/2]).

chameneos(Mall, Color) ->
   Mall ! {self(), Color},
   receive
      {OtherColor} ->
         if Color == OtherColor ->
            chameneos(Mall, Color);
         true ->
            chameneos(Mall, 3 - Color - OtherColor)
         end
   end.

mall(0, Diff, Main) ->
   io:fwrite("Color changes: " ++ integer_to_list(Diff) ++ "\n"
      ),
   Main ! {done,self()};

mall(N, Diff, Main) ->
   receive
      {Pid1, C1} -> nil
   end,
   receive
      {Pid2, C2} ->
         Pid1 ! {C2},
         Pid2 ! {C1},
```

```
         if C1 == C2 ->
            mall(N - 1, Diff, Main);
         true ->
            mall(N - 1, Diff + 1, Main)
         end
   end.
start() ->
   Main = self(),
   [Chame | _] = init:get_plain_arguments(),
   C = list_to_integer(Chame),
   Mall = spawn(fun () -> mall(C*500, 0, Main) end),
   foreach(fun(Color) -> spawn(fun() -> chameneos(Mall, Color
      rem 3) end) end, lists:seq(1, C)),

   receive
      {done, _} -> nil
   end.
```

D.2 Go

```
package main
import ("fmt"
      "os"
     "strconv")

type Color int
const (blue Color = 0; red; yellow)

type Request struct{col Color; reply chan Color}

func chameneos(col Color, mall chan Request) {
  reply := make(chan Color)
  for {
    // in forest
    mall <- Request{col, reply}
    // waiting to meet
    otherCol := <- reply
    // changing color
    if col != otherCol {col = 3 - col - otherCol}
  }
}
func mall(cham chan Request, done chan bool, Chams, Rounds int)
   {
```

```
  diff := 0
  for r := 0; r < Chams * Rounds / 2; r++ {
    fst := <- cham; snd := <- cham
    fst.reply <- snd.col; snd.reply <- fst.col
    if fst.col != snd.col {diff += 1}
  }
  fmt.Println("Color changes:", diff)
  done <- true
}
func main() {
  if len(os.Args) != 3{
    fmt.Printf("Usage: %s numChams Rounds\n", os.Args[0])
    return
  }
  ArgChams := os.Args[1]
  ArgRounds := os.Args[2]
  Chams, errC := strconv.Atoi(ArgChams)
  if errC != nil {
    fmt.Println(errC)
    os.Exit(2)
  }

  Rounds, errR := strconv.Atoi(ArgRounds)
  if errR != nil {
    fmt.Println(errR)
    os.Exit(2)
  }

  req := make(chan Request); done := make(chan bool)
  go mall(req, done, Chams, Rounds)
  for i := 0; i < Chams; i++ {go chameneos(Color(i % 3), req)}
  <- done
}
```

D.3 Haskell

```
import Control.Concurrent
import System.Environment

data RequestChan = RequestChan (MVar (Int, (MVar Int)))
data DoneChan = DoneChan (MVar Bool)

chameneos :: (MVar (Int, (MVar Int))) -> Int -> IO()
```

```
chameneos mall col = do
   re <- newEmptyMVar
   putMVar mall (col, re)
   othercol <- takeMVar re
   if col /= othercol
      then do
         chameneos mall (3 - col - othercol)
      else do
         chameneos mall col

mall :: Int -> (MVar (Int, (MVar Int))) -> MVar Bool -> IO()
mall reps cham done = do
   (fstColor, fstReply) <- takeMVar cham
   (scdColor, scdReply) <- takeMVar cham
   putMVar fstReply scdColor
   putMVar scdReply fstColor
   if reps > 1
      then do
         -- print reps
         mall (reps-1) cham done
      else do
         -- putStr "done"
         putMVar done True

main = do
   args <- getArgs
   let num = read (args!!0)
   let reps = read (args!!1)
   r <- newEmptyMVar
   d <- newEmptyMVar
   request <- newMVar (0, r)
   forkIO (mall (num * reps `div` 2) request d)
   mapM_ (\ x-> forkIO (chameneos request (x `rem` 3))) [1..num
      ]
   tmp <- takeMVar d
   return ()
```

D.4 Java

```
class Chameneos extends Thread {
  final int Blue = 0, Red = 1, Yellow = 2;
  int col;
```

```
  Mall mall;
  Chameneos(int c, Mall m) {
    col = c; mall = m; setDaemon(true);
  }
  synchronized void meet(int otherCol) {
    // changing color
    if (col != otherCol) col = 3 - col - otherCol;
    notify();
  }
  synchronized public void run() {
    while (true) {
      // in forest
      mall.arrive(this, col);
      // waiting to meet
      try {wait();} catch (InterruptedException e) {}
    }
  }
}
class Mall extends Thread {
  int reps;
  int chams;
  Chameneos fstCham, sndCham;
  int fstCol, sndCol;
  Object mutate = new Object();
  Mall(int r) {
    reps = r;
  }
  synchronized void arrive(Chameneos ch, int c) {
    while (chams == 2) {
      try {wait();} catch (InterruptedException e) {}
    }
    chams += 1;
    if (chams == 1) {fstCham = ch; fstCol = c;
    } else {
      sndCham = ch; sndCol = c;
      synchronized(mutate) {mutate.notify();}
    }
  }
  public void run() {
    int diff = 0;
    synchronized(mutate) {
      for (int r = 0; r < reps; r++) {
        try {mutate.wait();} catch (InterruptedException e) {}
        synchronized(this) {
          fstCham.meet(sndCol); sndCham.meet(fstCol);
```

```
        if (fstCol != sndCol) diff += 1;
        chams = 0; notifyAll();
       }
     }
   System.out.println("Color changes: " + diff);
   }
  }
  public static void main(String args[]) {
       if(args.length < 2){
          System.err.println("Usage: java Mall Chams Rounds\n");
          return;
       }
       int Chams = Integer.parseInt(args[0]);
       int Rounds = Integer.parseInt(args[1]);
   Mall m = new Mall(Chams * Rounds / 2);
   m.start();
   for (int i = 0; i < Chams; i++) new Chameneos(i % 3, m).
       start();
  }
}
```

D.5 Pthread

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <stdlib.h>

int Chams = 0;
int Rounds = 0;
typedef enum {blue, red, yellow} Color;

sem_t req_meet, sent_meet, req_other;
Color color;
Color *reply;

void *chameneos(void *colptr) {
  Color col = *((int *) colptr);
  Color othercol;
  for (;;) {
    // in forest
    sem_wait(&req_meet);
    color = col;
```

```
    reply = &othercol;
    sem_post(&sent_meet);
    // waiting to meet
    sem_wait(&req_other);
    // changing color
    if (col != othercol) col = 3 - col - othercol;
  }
}

void *mall(void *ptr) {
  int diff = 0;
  for (int r = 0; r < Chams * Rounds / 2; r++) {
    sem_wait(&sent_meet);
    Color fstcol = color;
    Color *fstreply = reply;
    sem_post(&req_meet);
    sem_wait(&sent_meet);
    Color sndcol = color;
    Color *sndreply = reply;
    sem_post(&req_meet);
    *fstreply = sndcol;
    *sndreply = fstcol;
    sem_post(&req_other);
    sem_post(&req_other);
    if (fstcol != sndcol) diff += 1;
  }
  printf("Color changes: %d\n", diff);
}

int main(int argc, char *argv[]) {
  if(argc < 3){
       printf("The Usage: %s Chams Rounds\n", argv[0]);
       exit(1);
    }

  Chams = atoi(argv[1]);
  Rounds = atoi(argv[2]);
  pthread_t mid;
  pthread_t *cid = (pthread_t *)malloc(sizeof(pthread_t) * Chams
     ); // id's of mall, chameneos
  sem_init(&req_meet, 0, 1); // semaphore req_meet = 1
  sem_init(&sent_meet, 0, 0); // semaphore sent_meet = 0
  sem_init(&req_other, 0, 0); // semaphore req_other = 0
  pthread_create(&mid, NULL, mall, NULL);
  for (int i = 0; i < Chams; i++) {
```

```
    int col = i % 3;
    pthread_create(&cid[i], NULL, chameneos, &col);
  }
  pthread_join(mid, NULL);
}
```