Abstract of “Realizing Concurrent Functional Programming Languages” by Eric Larsen McCorkle, Sc.M, Brown University, May, 2008.

This thesis is concerned with a methodology for deriving parallelism from programs where there is little or no coarse–grained concurrency. It presents the case that that concurrent functional programming- a paradigm formed from the assimilation of concurrent programming constructs into higher–order functional languages is the most promising direction for achieving that end. It argues that higher–order functional languages are highly versatile, citing as example the construction of higher–order concurrent and transactional language constructs from very basic primitives using the expressive power of functional languages. It further argues that functional programming, whether purely–functional or merely mostly–functional exhibits a natural concurrency which is scarce at best in other paradigms. The thesis further argues that the effectiveness of concurrent functional programming is limited at the present by two factors: the lack of effective runtime support and the false dependence introduced by existing synchronization structures. On the issue of runtime support, the thesis argues that the traditional runtime structure of programs is ill–suited for functional- and particularly concurrent functional programs. It addresses this by designing a runtime environment specifically intended to support concurrent functional programming from first principles. The description of the runtime system in this thesis is primarily concerned with establishing a foundation for a sound runtime system. As such, the thesis develops formal models of the components of the system and proves correctness conditions about the core algorithms. On the issue of false dependence, the thesis describes a number of techniques and data structures which mitigate the issue of false dependence. These techniques give rise to data structures which may be produced and consumed concurrently in a manner similar to a pipelined processor. They also provide both a method for exposing concurrency in modify–in–place stateful computations, and for eliminating false dependence on the results of these computations. Lastly, the thesis discusses the conclusions, the effort to produce a real implementation of the runtime system I describe, a retrospective on the process of developing these ideas, and directions for possible future work.

Realizing Concurrent Functional Programming Languages

by
Eric Larsen McCorkle
B. S., Georgia Institute of Technology, 2004

Thesis

Submitted in partial fulfillment of the requirements for the Degree of
Master of Science in the Department of Computer Science at Brown University

Providence, Rhode Island
May, 2008

This thesis by Eric Larsen McCorkle is accepted in its present form by the Department of Computer Science as satisfying the thesis requirement for the degree of Master of Science.

Date ________________ ____
Maurice Herlihy, Advisor

Date ________________ ____
Thomas Doeppener, Reader

Approved by the Graduate Council

Date ________________ ____
Sheila Bonde
Dean of the Graduate School

In memory of my grandmother
Joan K. Adkins
Dec 12, 1923 - Feb. 14, 1999

Vita

Eric Larsen McCorkle was born on May 18, 1982 in Spartanburg, SC, as the oldest son of Kent McCorkle and Rhonda (Larsen) McCorkle, moving to the vicinity of Asheville, NC shortly thereafter, where he lived for duration of his childhood. Eric graduated from Milton High School in Atlanta, GA in June of 2000, and enrolled in Georgia Institute of Technology, majoring in Computer Science, with a minor in Psychology. Eric's undergraduate studies in Computer Science focused on operating systems. He graduated in December of 2004 with a Bachelor of Science in Computer Science. Eric lived and worked in the Atlanta area for the following two years. He then moved to Providence, RI to begin his studies at Brown University in September of 2006.

Acknowledgements

First and foremost, I am grateful to my advisor, Maurice Herlihy for his continual insight and advice with regard to my research efforts, and for his aid to my academic career.

I thank my parents, Kent and Rhonda McCorkle for their support and contributions over the course of my endless education. I also acknowledge that my father was, in the end right about a certain statement I made at age seven, concerning the duration of my education.

I am grateful to my younger brother Ryan McCorkle for too many things to possibly list.

I have been quite fortunate to have encountered excellent teachers throughout my education. I recognize the following professors, teachers, and mentors: Ben Raphael, John Savage, Thomas Doeppener, Kim Harrington, Olin Shivers, Michael Reid, Tremayne Brown, Phil Hutto, Milos Prvulovic, Dana Randall, Ross Friedman, Jane Serkedakis, Judy Hammack, Ray Johnson, Joann Stephens, John Hurd, Michael Wallace, Jewel Albright, and Deborah Briggs.

Finally, I thank the following friends, in no particular order, some whom I have known much longer, but all of whom have played some part in my efforts leading up to and during my studies at Brown: Jabulani Barber, Lauren Hutter, Harrison Caudill, Shakti Saran, Frank Rietta, Lisa Dixon, Rosemary Simpson, Anna Waymack, Jennie Rogers and Matthew Duggan.

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

The essence of computer science lies in the description, analysis, and implementation of various models of computation. Basic models such as the λ-calculus or the Turing Machine are concerned with the limits on what is computable at all and on the efficiency of algorithms to solve various problems. Other models, such as programming language semantics or compiler intermediate representations concern themselves with analysis and translation, while models such as computer architectures and operating systems are concerned with what can be feasibly constructed and efficiently executed. Most classical models of computation feature a single flow of control, or *thread*. The evaluation or execution of these models is done by performing a series of translations according to the rules of the model.

Parallel or *concurrent*[1] models of computation, on the other hand, concern the case in which there are multiple independent flows of control. These models are equivalent in terms of computability to any single–threaded model of computation; there does not exist any concurrent algorithm capable of deciding a problem which is undecidable by single–threaded computation. Likewise the bounds on *total* resource usage by a given algorithm remains the same for both concurrent and single–threaded programs. However, parallel execution of a concurrent algorithm can drastically reduce its execution time. This is put to use in hardware particularly, but also in all levels of computing, as I will discuss in section 2.

There are also problems which fall strictly into the realm of concurrency, as they involve the communication between or coordination of several independent agents. As with classical single–threaded computation, there exist problems within concurrency theory which are unsolvable by any protocol. Likewise, there exist constraints of feasibility and efficiency governing the actual construction and operation of concurrent models of computation. Ensuring the correctness of concurrent programs is significantly harder than for single–threaded computation in most cases. Lastly, some programs and algorithms do not lend themselves naturally to concurrency.

Concurrency has been and remains a difficult area within computer science for these reasons. The idea of improving performance by doing things in parallel is present in all aspects of computing, from the implementation of computing hardware to the evaluation of higher–order languages. However, these gains do not come easily. As previously mentioned, parallel programming is a difficult thing to do correctly. Even obvious

[1]There is a subtle difference in the meaning of concurrency and parallelism. This being stated, I will use the term "concurrency" to effectively refer to both from here on.

parallelism is not necessarily easy to exploit. More importantly, opportunities for parallelism are not always obvious, nor easily utilized.

This thesis is concerned with a style of programming which is able to take advantage of concurrency in programs which do not readily expose it: namely concurrent functional programming. I begin with further introduction of concurrent computation, its benefits, its limitations, and the two main factors which limit the degree to which a program can benefit from parallelism. Chapter 2 discusses the existing body of work concerning concurrency and parallelism, starting with hardware and operating systems, then discussing theories of concurrency, synchronization, and finally, concurrent programming languages. Chapter 3 presents the main focus of the thesis: concurrent functional languages. It presents the core concept of concurrent functional programming, followed arguments in favor of it which focus on its versatility, natural concurrency, and correctness. The chapter then presents the two main problems which impede the viability of concurrent functional programming, and the solutions this thesis proposes.

The second part of the thesis focuses on the design and implementation of a new execution model, or *runtime system* which is built from the start to support the assumptions made by concurrent functional languages. Chapter 5 analyzes the requirements, then provides an overall description of the runtime system I designed and implemented based on these requirements, as well as arguments supporting the design decisions I made. Chapters 6, 7, and 8 describe the various components of the runtime system in greater detail.

The third part of the thesis describes several new constructs for improving concurrent functional programming by eliminating false data–dependence. Chapter 10 describes the effective use of *futures*: a concept central to concurrent functional programming. Chapter 11 describes how to implement futures for more complicated data structures, such as sets and maps. Chapter 12 describes how to extend this to represent futures on the results of a stateful computation.

The final part concludes the thesis. Chapter 13 briefly covers the essence of each part. Chapter 14 is a personal retrospective on the thought processes that gave rise to this work, as well as a brief discussion of several ideas which I developed early on, but did not include. Finally, chapter 15 describes several ideas for continuing the work described in this thesis.

Chapter 2

Background

Concurrency and parallelism are present in almost all levels of computation, ranging from hardware up to the highest levels of programming language abstractions. This chapter discusses the history and existing work regarding concurrency and parallelism in various aspects of computing. It begins by briefly discussing parallelism in hardware, following this with a more detailed overview of parallelism from an operating systems standpoint. It then covers the history of the theory of concurrent computation, various models of concurrency, and notions of synchronization. Finally, the chapter covers concurrency in programming languages, with an emphasis on concurrent functional languages.

2.1 Parallelism in Hardware

Hardware is quite arguably the first real instance of parallel computing. Parallelism is naturally present in hardware; electrical circuits are capable of computing any number of logic functions at once[1]. Most hardware is comprised of pure logical functions, which can be evaluated very efficiently because of this parallelism. By contrast, evaluating the same functions in a non–parallel fashion would be quite expensive.

Most of the progress in computer architecture[57] has exploited this parallelism in some way. Pipelining overcomes the problems which arise from sequential, multi–stage logic by permitting the varying “stages” of operations to overlap. Speculative execution simultaneously computes all possibilities arising from some decision, and keeps the one that actually occurs. Superscalar architectures execute multiple instructions in parallel. Other tactics similarly exploit parallelism to avoid waiting, to shorten critical paths, or to overlap multi–stage operations.

Dataflow architectures[15, 112] are an alternative to von Neumann–style processors, which were designed for the execution of functional programs with very high levels of parallelism. Dataflow architectures are very efficient at exploiting fine–grained parallelism. However, the lack of emphasis on parallelism at the time, coupled with the reluctance to adopt both functional programming *and* a new architecture prevented their widespread adoption.

The advent of VLSI and modern CPU’s brought about a phenomenon in processor performance known

[1]In reality, there are constraints, such as maximum fan–out or power consumption, but the available parallelism, even under these constraints is considerable.

as Moore's Law. Fabrication technology has been able to (and at the time of writing, is still able to) continue to shrink the feature size at a steady rate. In semiconductor technology, shrinking the size of a transistor corresponds to a growth in the speed at which the transistor is able to change states. The result of this was an exponential growth in the sequential speed of processors from the 1970's continuing through the year 2004. At this point, the feature size was so small that the physical laws limiting increases in sequential speed were fundamentally different from the ones which originally enabled this exponential growth.

During the period in which sequential processing power doubled every 18 months, there was little demand for parallel execution, except as a means for pushing the limits of available processing power as far as they would go. However, as it became evident that sequential power would not continue its rapid growth, hardware design began to see new innovations aimed to exploit the natural parallelism of circuits.

VLIW, or very long instruction word architectures cluster instructions into "packets" which are executed in parallel. Most VLIW architectures also include speculative instructions to allow explicit speculation. The central idea of VLIW is to allow as many of the decisions regarding speculation and scheduling to be made statically, rather than decided during execution. Symmetric Multi–Threading is an approach which attempts to better use execution units in a processor. Most processors contain many instances of various kinds of execution units, such as integer or floating point units. Processors anticipate the greatest possible use of any kind of unit, and as a result, have more total units than they can actually use. Symmetric Multi–Threading simulates multiple processors by maintaining multiple execution states (threads), and allocating execution units to each thread on demand.

The most recent trends in hardware parallelism take advantage of the continued growth in transistor counts. Multi–core architectures place several whole processing units on a single die (usually with a single, shared cache). At the time of writing, multi–core architectures are the most promising direction for continued growth of processor power for general–purpose processors. Lastly, the success of graphics processing units and stream processing has given rise to the notion of array processors, heterogeneous architectures, and other approaches. To give some examples: stock graphics processors have outperformed high-performance CPU's on several occasions, the recent Cell architecture incorporates eight stream–processors in addition to its main execution units, and modern supercomputers are making increasing use of both array- and stream–processing concepts.

Unless some unforeseen boon restores the explosive growth of sequential processing power for longer than a decade, parallelism is and will remain the primary mechanism for improving hardware performance this side of nanotechnology or quantum computing.

2.2 Parallelism in Systems

One of the first instances of parallelism in the realm of systems came with the advent of the Multics operating system and its successor, the UNIX time sharing system. Though timesharing systems did not imply true parallelism, they created an environment in which processes *appear* to execute in parallel[2] This gave rise to a number of notable errors, including concurrent access to files, and the race conditions which plagued signal handling semantics in the early days of UNIX. These early problems foretold the difficulties which arise from

[2]There are subtle errors which can arise in a truly–parallel system, but are impossible under this sort of simulated parallelism.

the combination of parallelism and stateful programming.

UNIX System V introduced the notion of interprocess communication with a variety of structures and system calls that mimicked the synchronization constructs prevalent at the time. This opened the door to parallel programming by degrees, though creating processes as an execution resource was a significant overhead. The concept of “lightweight processes”, or “threads” arose from the desire for explicit parallelism at a lower cost. The first threads packages[38, 30] worked completely within the scope of a single process. User–level threads implement lightweight explicit parallelism, but they only make use of a single execution resource (the process which creates them). Later threading implementations based on this design introduced the concept of system–level threads, which can make use of multiple execution resources. The term 1:N threading came to refer to user–level threading, while 1:1 (or sometimes N:N) threading referred to system–level threading.

The SunOS operating system (which became Solaris) adopted a different, hybrid approach. The direct correlation between user and kernel threads makes for a significant thread–creation overhead (although not nearly as high as the overhead for process–creation). Furthermore, assigning a kernel thread to each user thread requires that applications tune themselves to the number of available processors in the system. The M:N threading model works by creating a near–constant number of kernel threads (called “lightweight processes” in Solaris parlance), onto which user threads are mapped by a user–level scheduler. Early attempts at this were fraught with difficulties arising from the lack of effective communication between user–space and the kernel.

The microkernel era, ushered in by Mach[128] placed much more emphasis on interprocess (or more accurately, interthread) communication and lightweight threading. Later efforts, such as L4[85, 86] refined these approaches. The microkernel era also saw efforts to fix the problems with M:N threading through better communication between kernel- and user–space[88], or new models for execution resources[8].

In the modern era, virtually all major operating systems have implemented multithreaded kernels, the only significant exception of which I am aware being the security–conscious OpenBSD[3]. Both FreeBSD[42] and Linux[21] have implemented mechanisms for M:N threading based on a combination of activations and asynchronous communication between kernel- and user–space. Solaris, meanwhile, has abandoned its approach to M:N threading[100], preferring 1:1 threading instead.

Given recent developments in hardware, it seems likely that systems will see a period of reorganization with the goal of better supporting parallelism. There is already an existing body of work in this direction, including a lock–free kernel implementation[89] and a simple wait–free scheduling algorithm[34]. There have also been more recent efforts in this direction in the context of industrial research.

2.3 Theory of Concurrency and Parallelism

The theory of parallel and concurrent computation has been host to a more fragmented approach than has been seen in conventional computational theory. Indeed, even defining the exact meaning of concurrency and parallelism has proven more difficult than one might expect. To add to this, seemingly subtle variations in semantics can produce a drastically different model.

The Actors model[70] has had the greatest influence on concurrency theory. Originally proposed as a

[3]This should not be interpreted as criticism or admonition, but rather as praise for OpenBSD’s single–minded focus on security

formalism for artificial intelligence, the actors model was developed into a model of concurrent computation by further work[28, 50, 3]. This model would later influence work on process calculi, asynchronous computability theory, and wait–free synchronization.

The PRAM model[44] was developed within the classical theory community at the same time as the Actors model, and served as the basis for research into parallel algorithms[79]. The PRAM model assumes a global clock and synchronization property which is considerably stronger than what exists within the Actors model[4]. The model was criticized by researchers in the concurrency theory community[47], modified to support asynchronous computation[29, 114] and ultimately abandoned in favor of an axiomatic approach.

Various process calculi have been developed alongside ongoing work in concurrency theory. The Calculus of Communicating Systems[101] and Communicating Sequential Processes[73] both served as a basis for early work in concurrency theory. More recently, the π–calculus[133, 102] provides a simple, yet Turing–complete model of concurrent processes. π–calculus models features found in a number of programming languages, such as Concurrent ML[130, 132], Bluespec, and others. This model has also seen use outside concurrency as well, having applications in security, business process modeling, and biology, among other areas.

The most successful theory of concurrency has taken an approach beginning from an axiomatic foundation based on the realities of asynchronous computation[104, 68, 117] and deriving increasingly powerful models, whose behavior can be understood using theories of sequential execution. Lamport's safe, regular, and atomic register models[83, 84] and later work[6, 23, 5] study the construction of reliable shared memory from a very unreliable base component. Herlihy and Wing's concept of linearizability[67] is arguably the single most important concept in concurrency theory. Linearizability provides a method for proving the atomicity of a given protocol, demonstrating that there is a discrete atomic operation at which it can be considered to "take effect". This form of composable atomicity is used to prove the correctness of numerous concurrent protocols. This approach has also led to significant results in asynchronous computability. Herlihy's proofs of the universality and impossibility of consensus[58] established the consensus heirarchy. Later, Herlihy applied algebraic topology[64] to asynchronous computation, eventually demonstrating the impossibility of the Renaming Problem along with Shavit[65]. Recent work includes progress towards formal logic for verification of concurrent programs[144], and toward type systems for verifying concurrent programs[110]

2.4 Synchronization

Synchronization refers to the act of controlling execution of a concurrent program. This is typically done to ensure that certain guarantees can be made, thus enabling reasoning about the correctness of the program. Synchronization is often the most easily recognizable feature of a concurrent programming language. As with the theory of concurrency, there have been several approaches.

Synchronization is a non–issue in the PRAM model, as there is a strong guarantee of ordering in the execution steps of processes. In the terms of more practical models of parallel computation, PRAM executes a barrier synchronization between every execution step. The implication of this is that PRAM assumes the

[4]Indeed, the PRAM model assumes a barrier synchronization between every execution step, which permits it to solve consensus in a single execution step.

ability to solve a consensus among any number of threads in a single execution step. Since consensus can be used to implement an arbitrarily complex structure[58], there is no need for synchronization.

Early synchronization concepts focused on the problem of mutual exclusion[81, 82] and the concept of critical sections. Dijkstra's semaphores, as well as similar approaches by Lamport and others[66] established the concept of *locks*. Hoare's monitors[72] further developed these ideas to allow conditional synchronization.

Synchronous channels are an alternate approach which grew out of process calculi[11, 101, 133]. Synchronous channels are a message–passing mechanism, by which one thread may "send" a value to another. Synchronous channels guarantee that both the send and receive action take place simultaneously, blocking both senders and receivers until both are present. An *asynchronous* channel, by contrast, blocks only the receiver until a message is present.

A similar concept, the incomplete–structure was developed by Arvind[16] as part of data–flow computing. These structures are also known as *futures*[5]. Incomplete–structures are used extensively in concurrent functional programming to represent the results of functions which are being computed by another thread.

Both channels and i–structures (as well as a mutex–like construct: the m–structure[20]) view synchronization as a means of communication, rather than a protection mechanism. Under this view, blocking is a necessary condition for continued execution, as the blocking operation produces some value which is used in future computations. This view of synchronization became popular in concurrent functional languages. Reppy further developed the concept, giving rise to the notion of higher–order concurrency[132] by treating synchronous operations as first–class entities[131], which could be composed using a number of operators.

Locking synchronization, however, has a number of deficiencies. First and foremost, it does not scale well. The performance gains of programs with coarse–grained locks typically fall off considerably when the number of processors grows beyond even eight. Secondly, the association between locks and the data they protect is entirely ephemeral; there is generally no *guarantee* of protection. It is possible to write a program which protects all data with locks, but still has race conditions. Lastly, locks introduce new, difficult errors such as deadlocks.

The alternative to locking synchronization is to develop protocols which are tolerant of concurrent access- a technique known as lock–free programming. One of the first instances of this style of programming was Lamport's and others' constructions of atomic registers from weaker primitives[83, 84, 6, 23, 5]. Atomic snapshots[2] are another example of this style of programming. Herlihy's contributions to this area are considerable, including the first impossible problem[58], as well as others[65], formal definitions of wait–free synchronization[59] and further theoretical work[64, 144], and a method for implementing arbitrary concurrent structures using consensus[60]. Following the understanding of the consensus heirarchy, a number of efficient lock–free data structures have been designed. These include, among others, lock–free queues[99, 108], stacks[56], lists [145], hash–tables[135], and other algorithms such as a memory allocator[98].

Desiging efficient lock–free structures using only a single–consensus, however, is difficult, and there is a limit to what is possible without stronger primitives[90][6]. Transactional memory was first proposed by

[5]The exact definition of the term "future" varies. In some terminologies, futures do not denote a synchronization mechanism, but merely a delayed execution, as found in lazy evaluation.

[6]At the time of writing, I have very recently proven a number of theorems which demonstrate the limits on what can be done efficiently with the ability to solve a given number of consensus problems simultaneously.

Herlihy and Moss[63], followed by a pure–software implementation by Shavit and Touitou[137]. Transactional memory permits the clustering of an arbitrary number of accesses and writes to memory into a single atomic action. Transactional memory has seen a great deal of activity lately in both industry and academic research. Notable approaches include hardware–based approaches[150], library–based approaches[87, 53], compiler–based approaches[1], and hybrid approaches[134, 31, 93, 127].

Given the lack of scalability of locking synchronization, it seems likely that non–blocking approaches such as transactional memory or lock–free data structures will become the dominant paradigm for synchronization as it relates to safety. In the case of synchronization–as–communication, blocking is inevitable, so the focus becomes the elimination of false data dependence. This is one of the issues addressed by this thesis.

2.5 Concurrency in Programming Languages

The history of concurrent languages has taken several paths, each following a particular paradigm in programming language design. As this thesis is concerned primarily with concurrent functional programming, I will give a considerably more exhaustive treatment to the functional paradigm than to others.

In general, side–effects and concurrency are anathema to one another. Purely–functional languages exhibit a property known as the *Church–Rosser property*, which implies that any order of evaluation of a given set of expressions yields the same results. This property enables the aggressive optimizations and lazy evaluation strategies found in purely–functional languages such as Haskell. As a corollary, any time the Church–Rosser property holds for some set of expressions, those expressions can also be evaluated in parallel[149][7].

In imperative languages, each statement implicitly takes an additional argument: specifically the state of the world at–large. Likewise, it also yields an additional result, namely the new state of the world[8]. The result of this is that the order of execution of statements in an imperative language *does* matter. Where functional (or rather mostly–functional) languages are "innocent until proven guilty", imperative languages are just the opposite.

Concurrency in an imperative language must be explicitly stated. Attempts to automatically derive parallelism from an imperative language have been studied, but failed to produce satisfactory results. Furthermore, any shared state must be protected through some synchronization mechanism. Early concurrent languages were based on explicitly–parallel control structures, such as a parallel do statement[36], or parallel loop structures, with synchronization done via lock/unlock[9]. Most languages of this kind were C–like imperative languages into which concurrency was forcefit, rather than languages designed explicity to be concurrent. This fact, combined with the fact that threading was dramatically different on various operating systems, due to the lack of a standard API gave rise to several external threading libraries. Ultimately, these converged into the modern pthread library[75]. The library approach to threading has recently come under criticism[22], the case being that compilers cannot properly account for concurrency in their program transformations without knowledge of threads and synchronization.

Imperative languages have made little progress beyond the pthread–style concurrency. While various

[7]This is the natural state of affairs in hardware, which accounts for the high degree of inherent parallelism there.

[8]This model of stateful computations is called a "monad", which I relegate to a footnote so as to minimize the frightening effects that word has on some readers.

[9]Or, in the archaic terminology of Dijkstra, *P* and *V*

language constructs exist for better managing concurrency and synchronization, such as Java's `synchronize` constructs or Eiffel's object–based synchronization, the model remains more or less the same.

Concurrent functional languages have a history of better integration with the *concept* of concurrency, but a lack of suitable implementation. While functional languages are more amenable to concurrency, there are relatively few actually parallel implementations of concurrent functional languages for various reasons (which happen to be one of the primary topics of this thesis). Literature describes many "abstract machines"[52], and there are a several implementations of concurrent functional languages with very lightweight, elegant 1:N threading models[130, 120]. However, research efforts in this area for the most part seem content to imagine but not to realize; real, parallel implementations of concurrent functional languages are lacking.

One of the first major functional languages with a parallel implementation was the Multilisp language. This Lisp dialect included two main explicitly–parallel constructs. Most notable of these were futures, which were similar to the I–Structures of dataflow languages[16].

The mostly–functional language ML (particularly Standard ML) saw several constructs for concurrency emerge, ultimately leading up to the development of the concept of *higher–order concurrency*, which is the cornerstone of the Concurrent ML[130] language. Concurrent ML lacks a truly parallel implementation, unfortuantely[10]. Other parallel ML implementations include Facile and ParaML[18], both of which focus on coarse–grained, cluster–style distributed parallelism. Similar systems include the π–RED$^+$ engine and the S/NET system[24].

The parallel evaluation of lazy languages has received considerable attention in literature; however, the majority of work in this area has focused on the design of various abstract machines[52] rather than real implementations. Lazy languages, when evaluated precisely according to lazy semantics tend to produce numerous short–lived threads, which perform poorly in traditional parallel runtimes. While this is a problem with concurrent functional languages in general, it is even more pronounced in lazy languages. The lazy functional language Haskell has seen a considerable effort to incorporate concurrent and parallel programming constructs, including CML–style constructs[120], and transactional memory[53]. Additionally, efforts have been made to realize a parallel Haskell implementation[54].

There are a number of languages based more or less on the functional paradigm, which are designed for a concurrent or distributed execution model. The Erlang language[13] was produced explicitly for concurrent and distributed programming, and is based on the message–passing paradigm. The Haskell–based pH[113] language is an attempt to produce an implicitly–parallel language, using the natural concurrency of functional languages.

Data–parallel languages model parallelism by applying the same operation to all elements in collection types such as arrays, sets, and others. This is tantamount to the applicative style of programming seen in higher–order functional languages. This style of programming also appears in specific applications such as GPU shader languages.

Dataflow languages, such as Id[111], also bear a significant similarity to concurrent functional languages. Dataflow languages make use of on–demand evaluation via I–structures[16] to exploit fine–grained parallelism. These languages were originally designed in conjunction with dataflow architectures[15]. The effective use of fine–grained parallelism, however, is not supported by traditional pthread–style runtime systems[11],

[10]The implementation of the runtime in this thesis aims to address this deficiency.

[11]The concepts from dataflow languages are one of the primary influences of this thesis, and enabling the effective use of fine–grained

so these languages have fallen into disuse in general programming. Dataflow languages have found use in other areas, such as stream- and event–processing and database systems.

Lastly the process–calculi CCS, and more notably π–calculus have also served as the foundation for several languages. Most interesting is the Turing–complete π–calculus, which has served as the foundation of the language PICT[121]. The hardware–centric Bluespec language also shows a striking similarity to the polyadic π–calculus, although it was not consciously designed that way[14].

Concurrent languages have evolved significantly from their first incarnations. Recently, concurrent languages have seen considerable interest following the end of Moore's law. As effective use of parallelism becomes more important, new languages with ever more powerful expressions of concurrency are likely to emerge.

parallelism is the focus of part II.

Chapter 3

Concurrent Functional Programming Languages

Functional programming exhibits a natural concurrency which can expose parallelism where other methods fail to do so. Concurrent functional languages- in theory -create parallel implementations of programs which are quite difficult to parallelize effectively using conventional methods. Despite its expressive power, concurrent functional programming is faced with a number of problems which prevent it from realizing its full potential in terms of fine–grained parallelism and performance.

This thesis is concerned with addressing the limitations on the viability of concurrent functional programming. The first major limitation is the lack of runtime support. Concurrent functional programming deals primarily in fine–grained concurrency: a coin not readily accepted by existing runtime systems.

The second factor addressed in this thesis is that of false dependence. Data dependence is the traditional foil of any sort of parallelism. No program may be made to execute in a shorter time than it takes to execute its critical path using any technique of parallelism. While data dependence is insurmountable, naïve programming or language constructs can give rise to a large amount of *false* data–dependence, which unnecessarily reduces the parallelism of the program.

This chapter serves as a statement of the problem addressed in this thesis. It first presents a case for concurrent functional programming, arguing that it represents the most promising style of programming for developing robust, highly–concurrent programs. It then describes the two factors which limit the viability of concurrent functional programming. Lastly, it gives an overview of the solutions proposed by the thesis.

3.1 A Case for Concurrent Functional Languages

Concurrent functional programming has emerged from the integration of concurrent programming constructs into purely- and mostly–functional languages such as Haskell or ML, and from the resulting acculturation of those constructs to the paradigms of higher–order functional programming. This style of programming lends itself very naturally to concurrency, and does not lend itself to the kind of errors which emerge in concurrent programming with shared–state. Concurrent functional programming has considerable expressive power as a result of its derivation from functional programming. It is able to express both fine- and coarse–grained concurrency in a succinct and simple fashion. Due to the versatility of functional programming, it can be adapted to various tasks, and is able to take advantage of the innovations of various concepts which have emerged from dataflow, data–parallel, and other research directions.

Concurrent functional languages tend to produce much more robust and error–free programs as a result of their reduced emphasis on stateful computation, as well as their type–systems. Shared state and parallel execution are known to be a problematic pairing, as evidenced by nearly half a century's worth of research on the topic, and by innumerable software problems. However, in concurrent functional programming, synchronization functions more as a mechanism of communication than one of protection. Indeed, in languages which lack side–effects such as Haskell, or which lack shared state like Erlang, synchronization is *strictly* a communication mechanism. Similarly, in the π–calculus (or its predecessor, CCS), there is no concern for concurrent access to shared state.

Furthermore, in languages which possess some degree of side–effects such as Standard ML, or which model stateful computation as does Haskell via monads, type systems can be employed to guarantee varying degrees of soundness. Transactional Haskell, for instance ensures that no shared variable is accessed outside of a transaction[53], and can enforce arbitrary invariants at runtime. Recent work on Hoare Type Theory[110] extends the concept of dependent types[122] to concurrent programs which use transactions to mutate shared state. There has also been other work on type–systems to guarantee various properties of concurrent programs. While the use of type–systems is not strictly limited to functional programs, it is a known fact that pure functions are far more amenable to reasoning than are imperative programs. For this reason, concurrent functional programming lends itself to greater verification though both automated and manual reasoning.

Functional programming languages have proven their ability to host very powerful abstractions for concurrent programming given only a bare–minimum set of primitives. Higher–order concurrency[132] implements the ability to treat synchronous operations as first–class values, to compose events into arbitrarily complex configurations using various combinators, and to create generalized protocol implementations using these events. Monadic transactional memory[53] (which might also be titled "higher–order transactions") implements a similar abstraction for transactional computing, treating transactions as first–class objects, which can be composed in the same way as events. Both of these abstractions can be built from a very stark set of primitives entirely within the context of the programming language for which they are being implemented[1]. Indeed, this concept of *derived forms* is quite common in functional languages. Many functional languages are formally defined in terms of a very small "core", on top of which the actual language is built.

This versatility enables concurrent functional programming to take advantage of the techniques devised by other parallel language research fields, such as dataflow, stream or array processing, data–parallelism, and others. The concept of "little languages" dates back to the early days of Lisp. The concept has proven popular in the Haskell world, whose considerably powerful monad combinators have been instrumental in the implementation of several such sub–languages.

Lastly, the techniques employed in concurrent functional programming can be applied to both coarse- and fine–grained concurrency. Numerous projects have produced data–parallel variants of a traditional functional language. Indeed, the applicative style of programming common in higher–order functional languages lends itself naturally to data–parallelism and coarse–grained concurrency.

[1] More efficient implementations would likely care to do a great deal more in the compiler.

Likewise, the Church–Rosser property, which holds completely in purely–functional languages, and for vast majority of a program in mostly–functional languages exposes the separable control flows in the program. Purely–functional languages can be parallelized automatically, simply by identifying the separation of control flows. Other languages, like ML, have stateful primitives but discourage their use unless it is actually beneficial. In any case, large portions of programs written in these languages are entirely pure, which serves to expose both coarse- and fine–grained concurrency in them.

For these reasons, concurrent functional languages represent the most promising foundation for concurrent programming available in the modern world. However, concurrent functional languages are, at the present, restrained by several limitations. The remainder of this chapter discusses these problems, as well as an overview of the solutions presented in this thesis.

3.2 Limitations on Concurrent Functional Programming

Despite its advantages, concurrent functional programming faces two limitations which impede its viability severely. The first of these limitations is the lack of runtime support; the second is the false data–dependence introduced by the naïve use of futures. The lack of runtime support is the more severe of the two, and is likely responsible for the lack of a solution to the second.

Functional languages are naturally amenable to parallelism, as I have argued in section 3.1, and as is evidenced by a considerable amount of research on the topic. However, in reality, concurrent functional languages have failed to produce the amount of parallelism which they seem capable, in theory, of producing. Indeed, truly *parallel* implementations of concurrent functional languages are rare enough.

The root of the problem lies in the fact that functional programming thrives on fine–grained concurrency, while virtually all runtime systems provide only coarse–grained parallelism. The traditional notion of a program's structure at runtime largely comes from the implementation of languages such as C and concurrency in the fashion of the pthread API. This should come as little surprise, given that sequential programming and coarse–grained parallelism have been viable, and arguably the most effective methods for achieving performance from the advent of VLSI forward until fairly recently.

Functional languages, however, are ill–suited for execution on such a runtime. The basic assumptions in the design of a program runtime for a C–like language are fundamentally different from those made for a language like Haskell. Nonetheless, functional languages are sometimes forcefit into such an environment, with varying degrees of success. Other functional language implementations venture by degrees from the traditional structure. However, in the area of concurrency, this disconnection of the language and runtime is particularly deleterious. Rapidly creating large numbers of threads with short lifespans is particularly detrimental to the performance of a pthread–style program, yet this is exactly what concurrent functional programs do- and must do to exploit fine–grained concurrency.

The second limitation is the introduction of false data–dependence. Concurrent functional programs make use of synchronization constructs such as futures and synchronous channels for communication. However, without care, these constructs can needlessly introduce false dependence. For instance, if a thread blocks on a future of a simple integer, this is wholly necessary, as the integer's value is needed to continue and cannot be subdivided in any way. If the thread blocks on a future of a complex data structure, the thread falsely depends upon the entire structure, when it may only need a small portion. Concurrent functional programs which make naïve use of futures will suffer from this false dependence. Programs structured in the form of the π–calculus (and by association, languages which derive from it) may suffer from similar maladies if channels are used to send complex data structures. The problem of false dependence is relatively unexplored, as a result of the lack of a truly parallel implementation of a concurrent functional language. However, given one it, is likely that this would become a much more obvious problem.

3.3 Proposed Solutions

This thesis addresses both of the limitations stated in section 3.2, proposing a completely new runtime system to address the first, and new algorithms and data structures which address the second. The remainder of this chapter gives a brief overview of both approaches.

Part II describes the solution to the problem of runtime support for concurrent functional programming. I propose a new structure for processes executing concurrent functional programs, and a new runtime system to provide the necessary primitives. This runtime system makes use of a number of concepts which are considered in the conventional wisdom of systems to be suboptimal or expensive. However, when these concepts are implemented in concert, they act to accentuate eachother's advantages and mitigate eachother's costs.

Part II first presents a case and rationale for these design decisions. As the implementation of this runtime which I have created makes heavy use of lock–free algorithms and data structures (and indeed, some of the rationale depends on lock–freedom), the following chapters then present formal descriptions and proofs of the correctness of the core algorithms. While the treatment of the new runtime system by this thesis is largely theoretical, I do describe a system which can be implemented on common hardware, using common operating system abstractions, and is based on such an implementation. The theoretical treatment is meant to establish a foundational system by providing the most difficult proofs (namely proofs of proper function and linearizability).

Part III addresses the issue of false dependence. It begins by describing methods for implementing data structures such as trees, sequences, and arrays in a manner such that they do not introduce false dependence. The subsequent chapters address more difficult and more original concepts. While structures whose meaning corresponds to their representation are easy to implement in this way, structures such as maps and sets are more difficult, as their meaning is *independent* of their representation. Finally, part III describes methods for eliminating false dependence in a situation where a structure is modified in–place, prior to another thread processing it. These methods have a close similarity to the techniques of lazy–evaluation, and give rise to a style of programming which computes a minimal set of values in a maximally–parallel fashion.

Finally, part IV concludes the thesis by discussing conclusions and implications. It also discusses the thought process by which this thesis came about, and finally, directions for future work.

Part II

Runtime Support for Concurrent Functional Programming Languages

Chapter 4

Overview

The effective use of fine–grained concurrency requires both the programming and language constructs to express it as well as a runtime system capable of providing what these constructs assume to be true. Concurrent functional assumes the ability to create many threads with short lifespans: an assumption which is not upheld by traditional program runtime systems. This portion of the thesis describes a runtime system built specifically to support programming in a concurrent functional style.

The motivation behind this work is to provide an execution environment which upholds the basic assumptions of the concurrent functional style of programming. Two key observations lead to the development of this runtime architecture. The first is that while paradigms like functional programming and higher–order concurrency are excellent at *expressing* concurrency, the actual implementation and execution falls short of expectations. In more direct language, while functional style might encourage us to create many threads very rapidly, "practical knowledge" discourages it. The second observation is that certain lightweight implementations of concurrent languages, Concurrent ML among them are able to achieve full-featured threading at a negligible overhead.

Proponents of functional programming have often touted the fact that functional programs are inherently parallelizable, while imperative programs are inherently sequential. Critics, however, point to the fact that this difficult to realize in a the "real world". Conventional wisdom in thread programming holds that it is a bad idea to create a large number of threads, to create threads rapidly, and to create short-lived threads, as the setup time for threads exceeds the benefit. However, the creation of lightweight user–threading implementations suggests that the concurrent functional way of doing things is not inherently unrealistic, but that current notion of the "real world" is simply unfriendly to it. Several attempts to build a lighter weight threading system on top of a traditional runtime[48, 107] lend further support to this argument. By designing a new runtime architecture whose explicit purpose is to support concurrent functional programming, it should be possible to realize the implicit parallelism expressed by functional and mostly–functional programs.

The first chapter, chapter 5 describes the conceptual process of designing such a runtime architecture and describes the actual architecture itself in detail. Interestingly, the architecture combines a number of design decisions any one of which in an isolated consideration is expensive, but when taken together serve to negate eachother's detriments, resulting in a runtime system with all the desired properties. Because of the properties of the components of the runtime architecture, the initial presentation takes of the form of reasoning from first–principles in the hopes that this will better illuminate the manner in which the components support eachother.

Chapter 6 describes in detail the process structure of the runtime system. This chapter is divided into both a conceptual description and a formal specification. Chapters 7 and 8 address in detail various components of the runtime system: namely its threading and memory management systems.

Program and algorithm examples in this part are presented in a pseudolanguage which approximates the C language, but in a more compact form, with certain type system features, and with certain liberties taken with regards to its semantics. It is meant to express the ideas more concisely and clearly, not to be a concrete language syntax. Its meaning should be clear enough to anyone familiar with C or similar languages.

Chapter 5

Design of the Runtime System

This chapter discusses the conceptual design of the runtime architecture. The creation of this architecture is motivated by the belief that concurrent functional programs actually can realize the implicit parallelism inherent in their structure if given the proper execution environment, and that the current model does not suit this task well. Therefore, I present the conceptual design process of the runtime architecture as though I were crafting it wholly from first principles with the explicit purpose of supporting concurrent functional programming.

Additionally, the runtime system includes several features which have a synergistic interaction wherein they each serve to eliminate negative properties of eachother. This is more evident when the runtime architecture is presented in a constructionistic manner, as this approach (hopefully) avoids calling up preconceived notions which would need to be refuted all at once otherwise.

The runtime architecture itself is a novel runtime architecture designed to support concurrent functional programming languages. It must support two main feature sets: those that pertain to functional languages, and those that pertain to concurrency. As it is, there does exist some overlap between the two feature sets. The following section states and justifies each feature requirement, and the remainder of the chapter describes the design process which gives rise to the architecture.

5.1 Required Features

As previously stated, the runtime system's required feature set stems from the two things it must support: functional programs and concurrency. The fact that this runtime system is built from the ground up for concurrency also mandates a number of implementation concerns: specifically the highest possible parallel performance.

At the minimum, functional languages must have some means of allocating memory. As they support both recursive data structures and functions, it is necessary to allocate arbitrarily–sized blocks of memory for this task. In a traditional runtime, this is done with two structures: a heap and a stack. The stack is "allocated" from by advancing a pointer. Stacks are generally used to allocate function contexts, or *frames*. The heap is a more complicated structure, and methods for heap allocation vary. The primary difference between stack and heap objects is that stack objects are deallocated in reverse of the order in which they are allocated, where heap objects are deallocated in an indefinite order.

The C language's standard library function `malloc` is perhaps the best-known example of an explicitly–managed dynamic allocator. Indeed, the name `malloc` is commonly used to refer to explicitly–managed dynamic allocation in general. Memory is allocated using the function `malloc`. It is valid until the same block is explicitly released using the `free` function. As previously mentioned, there is no requirement on the order in which blocks are freed. Implementations of this style of allocation rely on data structures to manage the available space, locate appropriately–sized blocks, and mark blocks as available when they are freed again.

Most functional languages support the notion of *garbage collection*[78]. Garbage-collected allocation also supports allocation of arbitrarily-sized blocks of memory. However, there is no notion of freeing memory. Instead, the runtime system is expected to automatically release unused memory. This eliminates the need for explicitly releasing memory and simplifies allocation at the cost of periodically detecting and releasing unused memory. In a garbage–collected environment, memory is typically allocated from a per-thread memory pool by simply advancing a pointer. The complex mechanisms used by a `malloc`-style allocator are not necessary, as memory is reclaimed automatically by the collector. It is worth noting that stack allocation is equivalently simple, except that the guarantee of LIFO ordering of allocation allows for simpler *explicit* deallocation.

Functional languages often support *continuations*[142], or the ability to snapshot an execution point and resume later. Continuations are also a commonly–used device within compiler internals[10], and are a common means for implementing exceptions[51]. Lastly, continuations have found uses within the context of operating system, both in the implementation of threading, and in more complicated manners[41, 33].

First–class continuations also give rise to threads, as they are equivalent to context switching[147]. I duly note, however, that there are differences between continuations and concurrency[138]. The execution of an unbounded number of threads on a finite number of execution resources is a well–studied problem. It is accomplished by periodically changing the thread being executed by a given resource by saving and resuming threads using continuations. Threading is only considered here in order to determine what runtime structures are required to implement it. The implementation of threading using lock–free scheduling and continuations is further discussed in chapter 7. The existence of threads has several implications for the runtime.

Both continuations and threads change the nature of frame allocation. In a single–threaded application without continuations, frames are guaranteed to be freed in reverse of the order in which they are allocated. In the presence of either threads or first class continuations[32], this assumption is no longer valid, as threads may share some part of their creator's context[1], and continuations permit return to a given context after execution has left it and entered another context.

With regard to heap allocation, the existence of threads, particularly the existence of many threads makes a strong argument in favor of garbage collection. Parallel garbage collection algorithms can drastically reduce the observable overhead of garbage collection. Though it is not apparent yet, the properties of this runtime architecture will all but eliminate any observable pause in execution. Furthermore, freeing explicitly–managed memory is problematic in highly–parallel programs, and has given rise to several

[1]This is only true in languages which support inner functions, as most functional languages do.

approaches[97, 96, 61, 35, 74, 95], all of which involve some overhead[55]. The existence of garbage collection vastly simplifies lock–free algorithms and transactional memory by eliminating the problems inherent with explicitly freeing a memory block. Practical experience- indeed the experience of building this runtime system further demonstrates the difficulty that can arise from using explicitly-managed memory in a lock–free algorithm.

The following list gives a summary of the required features of the runtime system:

- Save and restore an execution state (Continuations)
- Execute an unbounded number of threads on a fixed number of execution resources (Scheduling)
- Low cost to create and maintain threads (Lightweight Threads)
- Allocate memory for function contexts (Frame allocation)
- Allocate memory for objects (Heap allocation)
- Automatically release unused memory (Garbage collection)

The following section describes the features of, and the case for the design decisions of the runtime system built around these criteria.

5.2 Design Decisions and Rationale

As it happens, the existence of a garbage collector significantly changes the constraints of the problem. Most importantly, it allows frame allocation and heap allocation to be consolidated into a single requirement, which in turn permits efficient implementation of both threading and language features such as continuations. The methods for implementing these structures are presented in this order, so as to elaborate how one follows logically from the the others.

The most effective garbage collectors operate by traversing the program's data structures and identifying all reachable locations. This is done using type signatures generated by the compiler. Garbage collectors have been designed which are able to perform collection in parallel with execution[27, 25, 62, 39, 116, 26, 17], with little to no pause required to activate or deactivate collection. The Cheng–Blelloch[27, 26, 25] collector is the most advanced of these, and is fully scalable to many CPU's, often moreso than the application it hosts. As it is, this runtime system further improves on existing approaches by implementing a mostly lock–free garbage collector.

In the presence of such a garbage collector, dynamic memory allocation is done via pools. Each thread occasionally allocates a large block of memory, and allocates from it by simply incrementing a pointer. The implication of this is that dynamic memory allocation becomes essentially a no–cost operation, and freeing *is* a null operation. In fact, these costs are transferred to the collection threads. However, there is evidence that even in sequential programs, garbage collection can be faster than dynamic allocation[9, 69]. Considering the substantial benefit of garbage collection in the context of lock–free algorithms and transactional memory, the case is sufficiently made for garbage collection within the scope of this thesis.

The change in allocation and deallocation costs in turn affects the choice of how to implement frame and heap allocators. Stack–based allocators are used in lieu of general dynamic allocation for frames, as frames (for the most part) have the property that memory is freed in reverse of the order in which it is allocated. As such, a stack allocator for a given thread can be implemented by a simple pointer, which is incremented to allocate, and decremented to free. However, the design of this runtime is informed by additional concerns. First, allocating from the heap is now equivalent to allocating from a stack, and deallocation is no longer a concern. Furthermore, the ordering assumptions concerning frame allocation are no longer valid, as mentioned in section 5.1. Even the best strategies for implementing first–class continuations must copy part of the stack when stack–allocation is used[71]. In summary, allocating frames from a stack no longer confers any advantage, and in fact becomes something of a liability. Therefore, the runtime allocates both frames and heap objects using a single garbage–collected memory allocator.

This decision is no innovation, nor is it untried. The Continuation–Passing Style of Appel[11, 10] is a common means of implementing first–class continuations and full closures when compiling functional languages. This style, also called *linked closures* is used in existing functional language implementations[140, 12], as well as more general systems[141]. Intelligent compiler optimizations can reduce overhead of this scheme considerably[136].

As with previous systems, decision to allocate frames using the heap allocator rather than a stack has significant benefits for both the implementation of threads and continuations. With frames allocated in this manner, continuations can be implemented simply as a pair of pointers: one to the start point of execution and one to a frame. Simply saving this pointer is enough to preserve the entire state of execution; the garbage collector will preserve all objects reachable from this frame, including the other frames which comprise the continuation's state. With the traditional stack–based runtime, this can only be accomplished by copying the stack in full[2]. This benefit carries over to threads as well. Creation of a thread only requires the creation of a continuation, plus whatever overhead is involved in adding the thread to the scheduling system.

The space overhead of threads and continuations is also drastically reduced by using heap–allocation for frames. As the stack is not copied when a thread or a continuation is created, threads and continuations are able to share some or all of their state, depending on the exact semantics concerning mutable objects. With regard to threads, under CML–style or POSIX–style threading semantics, threads exit upon completion of their entry function, so the only overhead of the thread at its creation is the scope in which the entry function is declared. As such, the space overhead of threads and continuations when using the heap–allocator is reduced to nothing more than a pair of pointers at best, and copies of some portion of the creator's frames at worst. Some existing functional languages, most notably CML employ similar strategies for implementing threading, and are able to implement an almost negligible–overhead 1:N threading system.

The final goal of the runtime system is to implement a lightweight threading system. This threading system should achieve a low overhead of thread creation and maintenance and rapid context switches. With the benefits of allocating frames with the heap–allocator, threads can be rapidly created in user–space with little effort. However, the issue of thread–creation overhead from the operating system itself remains. For this reason and for others I shall discuss in the following paragraph, I opt for an M:N threading implementation.

There are several factors informing the choice of M:N threading. The CML runtime demonstrates that highly efficient 1:N threading can be implemented using the strategies I have chosen so far. Scheduler

[2]More work must be done to properly implement references to objects within a frame.

activations[8], when combined with asynchronous mailbox–style communication between kernel- and user–space[88] provide a groundwork upon which this sort of M:N threading system can be implemented. The FreeBSD Kernel Scheduling Entity (KSE)[42] system implements exactly this sort of interface. The KSE system was introduced in the FreeBSD 5 series[94]. A similar system, the Virtual Processor Interface (VPI)[21] has also been introduced to Linux. Lastly, this sort of an interface is markedly similar to the state of affairs on a bare–CPU environment, which makes the runtime easier to port to bare metal[3].

I would be remiss if I failed to mention the criticisms of M:N threading and activations and present counterarguments. This sort of activation system can be synthesized, though imperfectly using the traditional POSIX threads API[75] by creating a fixed number of threads. This approach typically has trouble with worker threads stalling. However, this runtime system is different from other implementations for several reasons. Most importantly, I do not make use of locks in the implementation of the scheduler. The only way for a thread to suspend is to be descheduled, or to make a blocking system call itself. However, the concurrent functional style, with its preference for futures, favors asynchronous system calls to the blocking variants. I am aware of the fact that many operating systems do not properly implement asynchronous I/O[118]. However, reported instances of dishonest API implementations hardly makes for a strong case for an otherwise detrimental design decision in a project such as this one! Furthermore, if blocking should become a problem, a solution involving I/O threads can be implemented to deal with it.

Assuming that worker threads will be kept as busy as there exists a demand for them to be, two issues remain to be resolved. The first is the scheduling of threads. As mentioned in the previous paragraph, implementing a lock–free scheduler effectively avoids blocking worker threads as a result of holding locks. The exact implementation of a lock–free scheduler will be discussed in chapter 7. The final issue is the handling of asynchronous signals, which is also related to preemptive scheduling. It should be noted that preemptive scheduling, or rather *involuntary context–switching* increases the overhead of context switches and thread creation/storage, as the entire processor–state must be stored as part of the thread's state.

The remaining issues arise from the need to interrupt a running thread, either to be replaced with another, or to execute a handler for some signal. This arguably should not be as difficult as it is, but for the fact that signal semantics have a history of poor specification and implementation. However, implementing a true interrupt mechanism in the presence of lock–free scheduling is quite challenging. Garbage–collection (particularly parallel garbage collection) is similarly inhospitable to asynchronous interrupts.

Safe–points are a mechanism for resolving the conflict between asynchrony and garbage collection[37]. Safe–points are periodic checks for statuses such as a pending signal, a timer, or another sort of asynchronous event. They are inserted at regular intervals in the program when it is compiled, ensuring a maximum execution time between any two safe–points. Asynchronous interrupts are not processed at the exact moment at which they occur, rather they cause a flag to be set, which will be detected the next time a safe–point is executed, at which time the interrupt will be processed. This has the effect of the program being a known state when it is interrupted. This is useful for garbage–collection, as it allows the garbage collector a consistent view of a thread–state. With regard to scheduling, it permits a very simple, elegant implementation of non–interruptible execution. Without safe–points, non–interruptible execution requires the setting and clearing of signal masks, as well as other bookkeeping. With safe–points, however, non–interruptible execution

[3]This fact is further discussed in the future work.

can be implemented simply by having the compiler omit all safe–points within a given block, without any management at runtime. Lastly, safe–points make all interrupts and context–switches voluntary. Because of this, there is no need to save and restore the entire CPU–state, and no need to store it in a thread. This approach effectively solves the issues arising from asynchrony.

In summary, the runtime system employs four central mechanisms: garbage–collection, heap–allocation of frames, M:N scheduling, and safe–points. These mechanisms have synergistic properties which serve to eliminate eachother's weaknesses. For instance, garbage collection eliminates the cost of heap–allocating frames, heap–allocation of frames in turn makes M:N scheduling more feasible, and safe–points help deal with issues that arise from asynchrony and scheduling. The result is a runtime system which can support the creation of a very large number of short–lived threads. By implementing the runtime system itself using lock–free algorithms, the parallel performance can be made to scale to a large number of physical processors.

The remaining chapters of this part describe the implementation of the runtime system in greater detail.

Chapter 6

Process Structure

The overall structure of a process in this runtime system differs enough from tradition to warrant an in–depth description. This chapter shows diagrams of the process structure, and presents a textual description of the major components of a process' memory image. Since one of the major challenges of implementing this runtime system is the almost exclusive use of lock–free data structures and algorithms, it is necessary to describe the format of data in considerable detail. Lock–free algorithms are dependent on the single–word compare–and–set operation for their functioning, which often relies on the trick of cleverly packing multiple elements of information into a single word. As such, it is necessary to describe shared structures down to the level of bits and bytes. This chapter describes the basic structures present in a process in the runtime system and covers the expected behavior of a process during execution.

6.1 Overall Structure

Similar to traditional runtimes, a process consists of several regions of memory space which are initialized when the process is created, as well as a regions which are allocated dynamically according to demand for memory. A program's initial state is implemented using the traditional mechanisms for the given operating system and architecture. For most systems, initial state is contained in a file which is memory–mapped into the new process' address space to create the initial memory–state. Dynamically allocated memory is organized into blocks, which are called *slices*. Slices are generally a fixed size, which is dependent upon the operating system and architecture; however, their exact size can vary.

Slices can be allocated for four kinds of data: static data, malloc–style allocation, garbage–collection, and custom use. Static data is used to initialize the entire runtime system, as many aspects of the system require knowledge of the exact number of executors, which is not known until startup. Slices for explicit–allocation and garbage–collection are allocated as needed, with a possible cap on the total size of memory committed to these functions. Lastly, slices for custom use are intended for implementing extensions to the runtime system.

To manage execution resources, the process creates a static number of system–level executable entities called *executors*[1]. Executors are usually implemented as either scheduler activations or as system–level

[1]This is a term which I created to more clearly describe what is usually called a "kernel thread". The anticipation of a bare–metal port of the runtime also motivated a change in terminology, as there is no kernel in such an environment.

```
type gc_thread = {
  queue : queue(gc_header),
  allocator : allocator,
  thread : thread*
  log : log_ent[LOG_SIZE]
}

type gc_header = <
  fwd_ptr : (gc_header*) word(1),
  list_ptr : (gc_header*) word(1),
  typeptr : (typedesc*) word(1),
  extra : (unsigned int) word(1)
> cacheline(1)

type objclass = NORMAL | ARRAY
```

```
type typedesc = {
  const : bool,
  class : objclass,
  nonptr : unsigned int,
  normptrs : unsigned int,
  weakptrs : unsigned int
}

type log_ent = {
  obj : void*,
  offset : unsigned int
}

gc_threads : gc_thread[execs];
```

Figure 6.1: Structures in the Garbage-Collected Memory Allocator

threads. The executor model also adapts well to CPUs for a bare–metal implementation of the runtime. The kernel–thread implementation of the runtime also creates a kernel thread for signal handling. The signal thread receives all signals, and delivers them to appropriate executors using mailbox structures. User–level threads are managed by a lock–free scheduler M:N scheduler system which multiplexes the executors to execute an unbounded, changing number of threads. The exact algorithm for lock–free scheduling is discussed further in chapter 7.

The remaining sections of this chapter discuss the exact structures present in the memory–management and threading systems.

6.2 Memory Management

Memory management in the runtime system is divided into two varieties: explicitly–managed and garbage–collected. Explicitly–managed memory allocation, or dynamically–allocated memory which must be explicitly released when it is no longer used is necessary for implementing the scheduling system, as well as for supporting calls to external functions which use this style of memory–management. Garbage–collected memory cannot be used for this purpose, as the collector itself depends on the scheduler to function.

The explicitly–managed allocator itself is an implementation of the lock–free allocator described by Michael[98]. Given a lock–free slice allocator[2], this allocator is also completely lock–free, and therefore reentrant. Further details on the structures and the algorithm can be found in Michael's journal article[98]. This runtime system makes no modifications to the algorithm. As such, the explicitly–managed allocation system is not discussed any further.

The garbage–collected memory allocator is a mostly lock–free algorithm. Its design is based on that of the Cheng–Blelloch[27] allocator, but without the use of spin–locks and room synchronization for work-sharing. The overall algorithm makes changes to the exact way in which objects are allocated and copied to the destination space in order to dispense with the need for spin–locks on objects. It also replaces the

[2]The slice–allocator is indeed lock–free, though its implementation is too simple to warrant a section to describe it.

room–synchronized stacks used for worksharing with a lock–free FIFO queue. Lastly, the algorithm uses atomically–modified bitmaps to allow arrays to be processed by multiple threads at a time. Figure 6.1 shows the structures present in the garbage collector.

The garbage collector creates a number of garbage collection threads equal to the number of executors. Each has its own `gc_thread` structure, which contains a local work queue (`queue`) and an allocator (`allocator`). The `queue` field is purely local and accessed entirely in a synchronous context, as is `allocator`. The `allocator` allocates space in the destination for objects to be copied. It is described abstractly to allow a more complicated system, such as a generational collector to be implemented. An `allocator` obtains a large block of free space and allocates from it simply by advancing a pointer. The same technique is used to allocate objects in normal execution. Each executor acquires a large block of memory in the current heap, then allocates from it by advancing a pointer. Because of this similarity, any leftover space at the end of a garbage collection cycle becomes the allocation pool for normal execution.

Each object is preceded by a `gc_header` structure. This structure acts as a description of the object's format, which allows the collector threads to interpret it. It also contains two fields which are used in the course of a collection cycle by the collector threads themselves. The `gc_header` is designed to occupy a cache line[3]. The `fwd_ptr` field is an atomically modified pointer to the copy of the object in the destination heap. The `list_ptr` field is used by the local queues. The `typeptr` field holds a pointer to the type descriptor for the object. Lastly, the `extra` field's exact meaning is dependent on the class of the object.

Arrays are processed differently than normal objects. Arrays which contain a sufficient number of elements and are of sufficient size are preceded by a bitmap. This bitmap is used to "claim" clusters of array elements, allowing an array to be simultaneously processed by multiple collector threads. Arrays are kept in a lock–free simple stack which is shared among all collector threads. Arrays which are *not* of sufficient size, or do not have enough elements to benefit from parallel processing are treated like normal objects.

Interpreting objects requires the ability to distinguish pointers and non–pointers. The runtime system implements this by requiring the compiler to generate a set of type–signature structures, which describe the format of a given object. The `typedesc` structure is a description of an object's type. The `const` field denotes whether or not the object is constant. The `nonptr` field contains the size in bytes of non–pointer information[4]. The `normptrs` and `weakptrs` fields contain the number of normal and weak pointers respectively. Weak pointers differ from normal pointers in that the collector may ignore them if memory–pressure is high. Normal objects consist of a header, immediately followed by non–pointer data, then normal, and finally weak pointers. For arrays, the `typedesc` structure describes the format of a single array element. The length of an array is held in the `extra` field of its object header.

As in the Cheng–Blelloch collector, program threads keep a write–log during a collection cycle, periodically entering the collector to process the log (usually when the log becomes full). The runtime keeps one log per executor, rather than per program–thread. The `log_entry` structure describes the structure of a single entry. Unlike in the Cheng–Blelloch collector, there is no need to keep the previous value of the field. The `obj` field contains a pointer to the modified object (actually a pointer to its `gc_header`), and the `offset` field contains the offset from the start of object data at which modification occurred. All modifications are

[3]This is beneficial, as it serves to prevent negative cache–effects resulting from the use of compare–and–set operations on the forwarding pointer field.

[4]This may contain pointers to non–garbage–collected data.

assumed to be word–sized[5].

The root–set for garbage collection is generated by traversing all global pointer–values and all active threads. The frames for all threads are accessible through their resume continuations. Traversing all global pointers, however, requires a description of the global data, just as traversing objects requires a description of their types. The `globals` variable is an array of pointers to all global pointers which is generated by the compiler, and allows the collectors to calculate the root–set.

As with the Cheng–Blelloch collector, the transition from collection–mode to normal execution must occur with a finite (and small) number of operations. As frames are now allocated as normal objects rather than a stack, they do not present a problem. However, the global pointers cannot be changed to point to objects in the destination space until the end of collection. In a generational collector, this problem is extended to all pointers. There are other issues with this transition, which are discussed in chapter 8. The solution to this problem is similar to the one used by the Cheng–Blelloch collector. Global (or all, in a generational collector) pointers are implemented as *pairs* of pointer values. During a collection cycle, collectors store pointers to the unused slot. When a collection cycle ends, the program threads switch the used and unused slots. This allows all pointers to be changed to their new values at once.

The garbage–collection algorithm is further discussed in chapter 8.

6.3 Executors and Threads

The overall design of the threading system for the runtime is an M:N threading system which uses a lock–free scheduler to multiplex execution resources called *executors* to execute user–level threads. Executors are patterned after scheduler activations. Inter–executor communication takes place via mailboxes: atomically–modified fields which are checked at periodic intervals by safe–points. Even in the scheduler, executors may only communicate with one another by setting flags in eachother's mailboxes, or via lock–free structures (to do otherwise would violate lock–freedom).

A static number of executor structures exist throughout the execution of the program. Each of these structures is created in reference to an actual execution resource. The executor structure consists of a signal mailbox, stack space[6], and a scheduler structure. The exact format of the scheduler structures varies depending upon the exact scheduling algorithm used. In all cases, however, the schedulers are strictly local to the executor, and are used to decide which of the threads currently assigned to the executor is executed in a given timeslice. In addition to the schedulers, a global lock–free queue exists for work–sharing purposes.

Threads are composed of an atomically–modified status and reference field, a mailbox structure, and scheduler information. The status/reference field combines the thread's scheduling status and a flag indicating that the thread is present in the scheduler's internal structures. Scheduling statuses have both an *request* and *acknowledged* state. For instance, the suspended state has both a "suspend" state, indicating that the thread will be suspended, and a "suspended" state, which indicates that it has been suspended. The reference flag indicates that the thread is present in scheduler data structures (either executing, in a local scheduler, or in the

[5]This ensures the consistency of pointer updates. Non–pointer data updates which span multiple words create multiple write–log entries.

[6]This stack space is used only for execution of scheduler code, garbage–collector code, and other runtime functions.

```
type executor = {
  id : unsigned int,
  os_exec : os_executor,
  sched : scheduler,
  current : thread*,
  c_stack : void*,
  mbox : exec_mbox,
  suspend_next : executor*
}

type thread = {
  stat_ref : status,
  mbox : thread_mbox,
  sched_info : sched_info,
  rlist_next : thread*,
  queue_next : thread*
}

type status = <
  ref : bool bits(1),
  state : state bits(3),
>
```

```
type state = RUNNING | RUNNABLE |
             SUSPENDED | SUSPEND |
             NONE

type exec_mbox = <
  run_scheduler : bool bits(1),
  run_gcthread : bool bits(1),
  os_signal : bool bits(1),
  os_signals : os_sigmask
>

type thread_mbox = <
  retaddr : void* word(1),
  frame : void* word(1),
  exec : (unsigned int) word(1),
  exec_mbox : (exec_mbox*) word(1)
>

type os_executor
type os_sigmask
type scheduler
type sched_info

executors : executor[execs]
workshare : thread* lf_queue
```

Figure 6.2: Structures in the Threading System

workshare queue). This is used to prevent a given thread from being inserted into scheduler structures more than once[7].

The thread mailbox structure is used for communication between user threads and the scheduler system. The mailbox contains a continuation–like structure, which is used to store the current frame and return address of the current thread when entering the runtime. When executing the user–thread, this structure contains the stack pointer to use when entering the runtime. Additionally, the mailbox contains two fields for use when executing the user–thread: the ID of the executor executing the thread, and a pointer to the executor's signal mailbox.

Details of the scheduling system and its algorithms are discussed in chapter 7.

6.4 Execution

Program execution begins with a single thread, which begins execution at the entry point of the program. Additional threads can be created or destroyed with a call to the runtime system. The process is also destroyed by a call to the runtime which terminates all threads and releases all memory.

When a thread is created, a pointer to its mailbox is stored to a pointer which is given to the thread creation function. Normally, this pointer will refer to a location in the thread's initial frame. Threads must keep this pointer in scope, as it provides the mechanism by which to make calls into the runtime. While in thread space,

[7]As it is, preventing this was the most difficult challenge I faced in the creation of the lock–free scheduler

the thread's mailbox contains the id of the executor currently running the thread in `exec`, and the address of the executor's stack in `frame`. Threads must save their own resume continuation into `ret_addr` and `frame`, then use the runtime's stack to call a runtime function as per the system's calling conventions[8].

All program executable code is expected to periodically execute safe–points. A safe–point must use the pointer to the executor mailbox to check for any raised signals, and must deal with them accordingly. The safe–point code *must* use an atomic compare–and–set to clear the executor mailbox. The `run_scheduler` flag indicates that the executor must call a scheduling function. The `run_gcthread` flag indicates that the executor must call a function which switches to its `gc_thread`.

Each executor maintains a per–executor memory pool for dynamic allocation. Threads allocate from this by advancing a pointer. When the memory pool is exhausted, threads replenish it by calling a `gc_alloc` function which allocates from shared storage. Frames for functions are also allocated in this fashion. Since the memory–pool information is accessible through an executor structure, it is necessary to pass a pointer to the thread mailbox into each function[9].

Each executor also maintains one garbage-collection thread. This structure also contains the write log for the executor. When garbage–collection is active, a thread must record all updates to any visible, mutable structure to this log. When the log is filled to capacity, the executor must switch to its garbage collection thread. If a structure is not visible to anyone except the current thread (such as a frame), it is permissible to avoid writing it to the log. Whenever the thread enters the scheduler or switches to the garbage collector, the current frame will be saved into the thread's resume continuation, which will make it visible. This has two implications. First, the thread must always reserve one slot in the write log for the write to the resume continuation. Second, and more subtly, if a frame (or any structure to which it points) does not escape the current closure, writes need only be logged when the thread enters the scheduler or garbage collector. If neither of these is the case, there is no reason to log any of the writes. Furthermore, only *updates* need be logged. Since frames tend to be updated frequently, they will likely generate the most entries (hence the strategy to reduce the write–log traffic they generate). However, when an object is allocated and initialized, there is no need to log writes, as it will not have been allocated in the destination space yet. If the object is reachable, it will be captured in an update either to some other structure which is visible either from a global pointer or in saving the thread's resume continuation. These two strategies serve to reduce the number of write–log entries which are generated.

Asynchronous signals from outside are handled using a mechanism similar to the one designed by Reppy for Standard ML[129], except that signal handlers do not interrupt any running thread; rather, they cause a new thread to be created for the signal handler[10]. Additionally, it is possible to design an asynchronous inter–thread signaling mechanism. This mechanism is discussed further in chapter 7.

This chapter has introduced the structure and execution of processes in the runtime system. It has also described the structures used by the various components of the runtime system in greater detail. Chapter 7 will discuss the actual implementation of threading, and chapter 8 will discuss the garbage collection algorithms.

[8]Most real implementations of the runtime functions require the ID of the executor to be passed in as an argument, which is why it is supplied in the thread mailbox. This is because many runtime functions rely on hazard pointers or a similar mechanism, and must know which executor is calling them.

[9]Optimizing compilers are, of course at liberty to optimize this as much as possible.

[10]This is analogous to the concept of "interrupt threads" in classical OS literature, and is significantly easier to achieve in this runtime architecture.

Chapter 7

Theading

This chapter describes the implementation of threading for the runtime system. As discussed in chapters 5 and 6, threading is implemented using an M:N approach based on a lock–free scheduling system. This chapter begins by discussing a formal model and requirements for the scheduler system, in preparation for discussing the implementation. This is followed by the lock–free scheduling algorithm and a proof of its correctness. Lastly, the chapter describes functionality which can be derived from the underlying formal model, including mutex- and CML–style synchronization, asynchronous thread termination, and asynchronous signaling.

Much of this chapter is dedicated to the development of the formal model, and the proofs of various theorems concerning this model. To the best of my knowledge, there has been no well–known precise semantic definition of a scheduling model prior to this thesis. Models of concurrency tend to take the form of an abstract semantics [133, 130, 120], wherein any number of threads may be created. The formal model I present in this chapter is meant to be relatively easy to implement in a real environment. The model describes a fixed set of execution resources, which are shared by a dynamic set of threads, the conditions under which threads execute, and the operations by which they may affect the system.

There are three main motivations in formally specifying the scheduling model. First, most “low–level” threading models are described in natural language. Natural language descriptions, no matter how long or in–depth [76, 49, 75] tend to give rise to ambiguities or problems. By contrast, definitions using formal notiation [124, 123] tend to be more brief, far more precise, and leave no ambiguity. To cite a specific case, the definition of the C language definition is far longer and less well–understood[76] than the definition of Standard ML[103], a much more complicated language. Additionally, attempts to formalize informally specified systems or languages such as C[119], or Java[151] have often turned up problems[126, 125].

Second, and more importantly, to prove the correctness of a lock–free scheduling algorithm requires a firmer foundation than informal semantics can provide. Given a formal semantic definition, the proofs of correctness follow fairly directly. The third issue is related to this. The models of concurrency I seek to support [130, 133, 120] are formally specified. Providing a formal specification of the runtime system on which they are implemented paves the way for both smoother implementations and verified systems.

The model provided here is meant to serve as a basis upon which other models of concurrency may be implemented. As such, its primitives are somewhat limited in scope, with little if any potential for interthread communication or synchronization. The implementation of traditional threading interfaces using the model

P	::=	$\langle E_a, E_s, T_a, T_s \rangle$	(process – state)	s	::=	ρ	(primitive)
E	::=	e	(executors)		|	`safepoint`	(safe – point)
e	::=	$\langle t, m \rangle$	(executor – state)		|	`spawn` t	(thread – create)
m	::=	$\top$	(message – on)		|	`raise` e	(raise – executor)
	|	$\perp$	(message – off)		|	`signal`	(signal – executor)
T	::=	t	(threads)		|	`sleep` t	(suspend – thread)
t	::=	$\langle S, m \rangle$	(thread – state)		|	`wake` t	(wake – thread)
S	::=	$s.S$	(statements)				
	|	`term`	(termination)				

Figure 7.1: Grammar for Processes, Executors, and Threads

is discussed at the end of the chapter.

7.1 A Formal Model of M:N Scheduling

Figure 7.1 shows the basic definitions of processes, executors, and threads. A process–state P is defined as two sets of executors: the active executors E_a and the suspended executors E_s, and two sets of threads: the active threads T_a and the suspended threads T_s.

An executor e is defined as a thread t and a message–state m. Message–states are either on $\top$ or off $\perp$. An executor's message–state is used to denote whether or not it should execute the scheduler at a `safepoint`.

A thread t is modeled as a sequence of statements S and a message–state m, which indicates whether the thread should be suspended at the next `safepoint`. For simplicity's sake, I treat threads as a generator for statements; the set of possible statements is not meant to describe a full computational model. A thread might be thought of as a sort of "ticker–tape" of the observable actions taken by the thread. This stream–type execution can be replaced with a monadic λ–calculus style evaluation to build systems atop this model.

Statements themselves represent atomic operations on the process–state. An important implication of this is that the implementations of `spawn`, `raise`, `sleep`, `safepoint`, `wake` and `signal` operations must be linearizable[67]. As previously mentioned, the ρ operation is an abstraction for some underlying model of computation. The `term` operation comes at the end of every thread's execution, and denotes thread termination and abandonment of the current state. Threads are created using the `spawn` operation. The `raise` operation sets the given executor's message state to $\top$. The `signal` operation wakes one sleeping executor. The perhaps deceptively–named `sleep` operation really only sets a flag on the current thread, requesting that it suspend at the next `safepoint`. The `wake` operation is similar to `signal`, except that it wakes a specific thread, whereas `signal` wakes some sleeping executor.

Figure 7.2 describes the evaluation rules for execution of the M:N threading model. The idle thread ω exists solely to suspend, then restart an executor. It is modeled as a `suspend` operation, followed by a sequence of some number of ρ operations, followed by `term`. The use of `term` here indicates that the state of the idle thread can be thrown away when a new thread is scheduled (meaning no resume continuation need be saved). Note that the since the idle thread is repeatedly used, the `term` statement does not imply the destruction of the thread structure itself, but rather the end of the current execution of it[1]. The program

[1]This tactic of abandoning the state of threads is used in several places to preclude the need for more than one traditional stack per executor.

Idle-Thread:

$$\begin{array}{rcl} \omega & = & \langle\ \texttt{suspend}.S_\omega, \top\rangle \\ S_\omega & = & \rho.S_\omega \\ S_\omega & = & \texttt{term} \end{array}$$

Program-Execution:

$$\frac{\exists e \in E_a\,(E_a - e, E_s, T_s, T_a, e \mapsto_S P')}{\langle E_a, E_s, T_a, T_s\rangle \mapsto_P P'}$$

Statement-Execution:

$$\begin{array}{rcl}
E_a, E_s, T_a, T_s, \langle\langle\texttt{term}, \ldots\rangle, \ldots\rangle & \mapsto_S & sched(E_a, E_s, T_a, T_s) \\
E_a, E_s, T_a, T_s, \langle\langle\rho.S, m_t\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, m_e\rangle\}, E_s, T_a, T_s\rangle \\
E_a, E_s, T_a, T_s, \langle\langle\texttt{safepoint}.S, m_t\rangle, \bot\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, \bot\rangle\}, E_s, T_a, T_s\rangle \\
E_a, E_s, T_a, T_s, \langle t = \langle\texttt{safepoint}.S, \top\rangle, \top\rangle & \mapsto_S & sched(E_a, E_s, T_a \cup \{\langle S, \top\rangle\}, T_s) \\
E_a, E_s, T_a, T_s, \langle t = \langle\texttt{safepoint}.S, \bot\rangle, \top\rangle & \mapsto_S & sched(E_a, E_s, T_a, T_s \cup \{\langle S, \bot\rangle\}) \\
E_a, E_s, T_a, T_s, \langle\texttt{spawn}\ (t = \langle\ldots, \top\rangle).S, m_t\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, m_e\rangle\}, E_s, T_a \cup \{t\}, T_s\rangle \\
E_a, E_s, T_a, T_s, \langle\texttt{spawn}\ (t = \langle\ldots, \bot\rangle).S, m_t\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, m_e\rangle\}, E_s, T_a, T_s \cup \{t\}\rangle \\
E_a, E_s, T_a, T_s, \langle\texttt{raise}\ e.S, m_t\rangle, \ldots\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, \top\rangle, \}, E_s, T_a, T_s\rangle \\
E_a, E_s, T_a, T_s, & & \\
e = \langle\texttt{raise}\ (e' = \langle t, \ldots\rangle \neq e \in E_a).S, m_t\rangle, \ldots\rangle & \mapsto_S & \langle(E_a - e') \cup \{\langle\langle S, m_t\rangle, \top\rangle, \langle t, \top\rangle\}, E_s, T_a, T_s\rangle \\
E_a, E_s, T_a, T_s, \langle\texttt{raise}\ (e' \in E_s).S, m_t\rangle, \ldots\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, \top\rangle\}, E_s, T_a, T_s\rangle \\
E_a, E_s, T_a, T_s, \langle\texttt{signal}\ .S, m_e\rangle & \mapsto_S & \\
\multicolumn{3}{l}{\exists e' = \langle t', \ldots\rangle \in E_s\ \langle E_a \cup \{\langle\langle S, m_t\rangle, m_e\rangle, \langle t', \top\rangle\}, E_s - e', T_a, T_s\rangle} \\
E_a, \emptyset, T_a, T_s, \langle\langle\texttt{signal}.S, m_t\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, m_e\rangle\}, \emptyset, T_a, T_s\rangle \\
E_a, E_s, \emptyset, T_s, \langle\langle\texttt{signal}.S, m_t\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, m_e\rangle\}, E_s, \emptyset, T_s\rangle \\
E_a, E_s, T_a, T_s, & & \\
\langle\langle\texttt{sleep}\ (t = \langle S', \ldots\rangle \in T_a).S, m_t\rangle, m_e\rangle & \mapsto_S & \langle\{E_a \cup \langle\langle S, m_t\rangle, m_e\rangle\}, E_s, (T_a - t) \cup \{\langle S', \bot\rangle\}, T_s\rangle \\
E_a, E_s, T_a, T_s, \langle t = \langle\texttt{sleep}\ (t' = t).S, \ldots\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, \bot\rangle, m_e\rangle\}, E_s, T_a, T_s\rangle \\
E_a, E_s, T_a, T_s, e = \langle\langle\texttt{sleep}\ (t = \langle S', \ldots\rangle\ | & & \\
\exists e' \neq e = \langle t, m'_e\rangle \in E_a).S, m_t\rangle, m_e\rangle & \mapsto_S & \\
\multicolumn{3}{l}{\langle\{(E_a - e') \cup \langle\langle S, m_t\rangle, m_e\rangle, \langle\langle S', \bot\rangle, m'_e\rangle\}, E_s, T_a, T_s\rangle} \\
E_a, E_s, T_a, T_s, \langle\langle\texttt{sleep}\ (t \in T_s).S, \ldots\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, \bot\rangle, m_e\rangle\}, E_s, T_a, T_s\rangle \\
E_a, E_s, T_a, T_s, & & \\
\langle\langle\texttt{wake}\ (t = \langle S', \ldots\rangle \in T_s).S, m_t\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, m_e\rangle\}, E_s, T_a \cup \{\langle S', \top\rangle\}, T_s - t\rangle \\
E_a, E_s, T_a, T_s, \langle\langle\texttt{wake}\ (t \notin T_s).S, m_t\rangle, m_e\rangle & \mapsto_S & \langle E_a \cup \{\langle\langle S, m_t\rangle, m_e\rangle\}, E_s, T_a, T_s\rangle
\end{array}$$

Scheduler Function:

$$\begin{array}{rcl} sched(E_a, E_s, T_a, T_s) & = & \exists t \in T_a \langle E_a \cup \{\langle t, \bot\rangle\}, E_s, T_a - t, T_s\rangle \\ sched(E_a, E_s, \emptyset, T_s) & = & \langle E_a, E_s \cup \{\langle\omega, \bot\rangle\}, \emptyset, T_s\rangle \end{array}$$

Figure 7.2: Execution Semantics

execution function $[P]$ specifies that execution steps from any active executor can be interleaved in any order.

The statement execution rules specify the meanings of each statement. It is important to note than in the statement execution rules, threads which are currently assigned to an executor are not present in either T_a or T_s. Also, in each rule, the executor e refers to the executor for which an execution step is being taken. The only rule for the `term` statement calls the *sched* scheduler function, discarding the current thread for e. The evaluation rule for ρ ignores the operation, assuming it to be processed by the underlying computational model. The ρ rule demonstrates the updating of the process–state, particularly E_a to reflect the changes to e.

There are three rules for the `safepoint` statement. In the first, the e's message–state is $\bot$, so no action is taken. The second and third reflect the current thread's status being $\top$ (executable), and $\bot$ (non–executable) respectively. The former calls *sched* with the previously–running thread inserted back into the set of active threads T_a. The later inserts the thread into the set of suspended threads T_s and calls *sched*.

The `spawn` statement has two rules, depending on the initial state of the thread when it is created. The only difference between these rules is that a thread starting as $\top$ (executable) is inserted into the set of active threads T_a, while a thread starting as $\bot$ (non-executable) is inserted into the suspended threads T_s.

The first two rules for `raise` statement are purely technicalities. The first rule covers an executor `raise`ing its own message–state, while the second covers `raise`ing another active executor. The third rule states that a `raise` performed on an inactive executor has no effect.

The `signal` statement has three rules. The first rule states that if both E_s and T_a are non-empty, then an executor from E_s is reactivated by removing it from E_s and inserting it into E_a. As will be shown later, the state of all executors in E_s is exactly $\langle \omega, \top \rangle$, meaning they are executing the idle thread. The last two rules for `signal` state that if there are no suspended executors E_s or active threads T_a, then the statement has no effect[2]

The `sleep` statement sets a thread's execution status to $\bot$ (non-executable). The rules for this statement are complicated, but the action is quite simple: it sets a thread's state to $\bot$. The four cases handle if the thread is the current executor's current thread, if it is some other executor's current thread, if it is active, or if it is already suspended. It is important to note that this does not set the executor's message state to $\top$, nor does it actually stop the execution of the thread. This detail will become very important in the implementation of blocking synchronization. To completely stop itself, a thread must execute the sequence `sleep.raise` e`.safepoint`, where e is the current executor. There is no way to block until another thread stops.

The `wake` statement has two rules. The first rule covers `wake`ing a suspended thread. The thread is removed from the suspended threads T_s, and inserted into the active threads T_a with its status set to $\top$ (executable). The second rule covers `wake`ing a thread which is not suspended, in which case the statement has no effect.

The scheduler function *sched* effectively describes what is expected of a real scheduler function. There are two cases for the function, and it is called in three rules. The first case covers normal scheduling, when there is an active thread available. The current executor chooses some thread from the active threads T_a and adopts it as its current thread. The second case covers the case where there are no active threads available. The current executor sets its current thread to be the idle thread ω and suspends, inserting itself into the suspended executors E_s.

[2]In the case of no suspended threads, an executor could be restarted, at which point it would execute *sched*, and be suspended again. This is case is explicitly stated for clarity's sake.

Definitions:

$\mathcal{P} = P_0, P_1, P_2, \ldots P_n \mid P_i \mapsto_P P_{i+1}$ (process – execution)
$T_e(\langle E_a, \ldots \rangle) = \{\forall t \mid \langle t \neq \omega, \ldots \rangle \in E_a\}$ (executing – threads)

Process-State Correctness Conditions:

$\forall P = \langle E_a, E_s, T_a, T_s \rangle$:

$\forall \langle t \ldots \rangle, \langle t', \ldots \rangle \in T_e(P)\ (t \neq t' \vee t = t' = \omega)$ (unique – execution)
$E_a \cap E_s = \emptyset$ (executor – non – duplication)
$T_s \cap T_a = \emptyset \wedge T_e(P) \cap T_a = \emptyset \wedge T_e(P) \cap T_s = \emptyset$ (thread – non – duplication)
$E_a = \emptyset \rightarrow T_a = \emptyset$ (no – deadsleep)

Figure 7.3: Correctness Conditions for Processes and Executions

Note that the three uses of *sched* introduce various cases. The rule for `term` discards the previously executing thread altogether. It is possible that T_a is empty, which may cause the executor to suspend. The second rule of `safepoint` adds the current thread to T_a prior to scheduling, meaning that the current thread may be chosen again, and that T_a cannot be empty, and thus, e cannot be suspended. Lastly, the third rule of `safepoint` suspends the current thread, but is otherwise identical to the rule for `term`[3]

Figure 7.3 introduces the correctness conditions for a valid process–state, as well as properties which the scheduler function must uphold. These, combined with the atomicity requirements of the execution semantics will form the conditions for proving the lock–free scheduler's correctness. These properties will also prove that this execution model can serve as a suitable foundation for implementing Concurrent ML.

A process–execution $\mathcal{P}$ is defined as a sequence of process–states, such that each successive process–state is the result of some application of the program–execution function to the current state. For convenience's sake, the executing–threads function T_e is defined as the currently–executing threads for all active executors, excluding the idle thread ω.

There are four conditions for a valid process–state. The first, the unique–execution rule states that no two executors may execute the same thread (unless they are both executing the idle thread). The second, the executor–non–duplication property states that a given executor must either be active or suspended, but cannot be both. Similarly, the third, the thread–non–duplication property states that a given thread must either be assigned to some executor, active, or suspended, but cannot be in more than one of these sets. The last states if there are any threads active, then some executor must also be active.

Figure 7.4 gives several properties necessary for describing an interactively–scheduled system, that is, one which is designed to obey some external timing constraint. Up to this point, the conditions given have been necessary for the runtime to function properly. The properties described from this point on represent the desired properties of an interactive scheduler, rather than required correctness conditions. It is certainly possible to implement a non–interactive scheduler which ignores these properties.

Two correctness conditions enable the implementation of the scheduling properties. For executors, the alarm–clock property simulates a timer interrupt. This property assures that for each executor, there is some

[3]Indeed, in the actual implementation, they are the same. Threads are not destroyed immediately upon termination.

Thread Properties:

$\forall t = \langle S, \ldots \rangle$:

$\exists \tau_s \mid \forall s_i \in S\ \exists s_{i+k}((s_{i+k} = \texttt{safepoint} \vee s_{i+k} = \texttt{term}\,) \wedge n < \tau_s)$ (safepoint – interval)

Scheduling Properties:

$\forall \mathcal{P}, P_i = \langle E_a, E_s, T_a, T_s \rangle \in \mathcal{P} \mid k > |T_e(P_i) \cup T_a \cup T_s|$:

$\exists \tau_a \mid \forall e_i \in E_a\ \exists P_{i+n} \mid (e_i = \langle \ldots, \top \rangle)$ (alarm – clock)
$\exists \tau_e \mid \forall t \in T_a\ \exists P_{i+n}\ (t \in T_e(P_{i+k}) \wedge n < \tau_e)$ (full – epoch)
$|E_a \cup E_s| = 1 \rightarrow \exists \tau_e \mid \forall t \in T_a\ \exists P_{i+k}\ (t \in T_e(P_{i+n}) \wedge n < \tau_e)$ (weak – epoch)

Figure 7.4: Properties of Interactive-Scheduler Systems

time interval in which its signal state will be raised. Additionally, there is a similar correctness condition for threads. The safepoint–interval property states that there is an upper bound τ_s on the amount of time before the execution of a safepoint for any thread.

In addition to the base conditions, two properties of scheduler functions are defined, both of which concern the maximum "pause" in a given thread's execution. These properties are not so much required preconditions as they are desired properties of the runtime. The full–epoch property states that there exists some global interval τ_e, such that any thread which is active (not executing) will be executed in fewer than τ_e steps. The weak–epoch property asserts the same property, but only for executions with a single executor. Obviously, both properties can only guaranteed for executions with a bounded number of threads[4]. As it turns out, the full–epoch property cannot be guaranteed at all under the current execution rules. In reality, this depends on an unlikely occurrence, that being that one executor stalls indefinitely. However, it still warrants consideration, as it is a consequence of the lock–free properties of the runtime system.

In order to fully provide a foundation, it is necessary to prove that the execution rules preserve the correctness properties of process–states. Also presented here is a proof that the full–epoch property cannot be guaranteed for all possible executions, and a corollary that states when it can be guaranteed.

Theorem 7.1.1 (Correctness Preservation)**.** *If some process–state P satisfies the correctness conditions for process–states, then P' also satisfies the conditions, where $P \mapsto_P P'$.*

The correctness conditions are:

(i) No two executors may execute the same thread, unless it is the idle thread. Formally:
$\forall \langle t \ldots \rangle, \langle t', \ldots \rangle \in T_e(P)\ (t \neq t' \vee t = t' = \omega)$

(ii) No executor may exist in more than one of E_a or E_s. Formally:
$E_a \cap E_s = \emptyset$

(iii) No thread may exist in more than one of $T_e(P)$, T_a or T_s. Formally:
$T_s \cap T_a = \emptyset \wedge T_e(P) \cap T_a = \emptyset \wedge T_e(P) \cap T_s = \emptyset$

[4]A fork–bomb demonstrates that this cannot be guaranteed when the number of threads increases without bounds

(iiii) If an active thread exists, then an active executor must also exist. Formally:
$E_a = \emptyset \rightarrow T_a = \emptyset$

Proof. The proof is straightforward from the definition of $\mapsto_S$, demonstrated below:

For condition (i), the only rule which affects an executor other than e is the second condition of the `raise` statement, but this rule only affects the message state m_e of e', not its current thread. Therefore, if (i) and (iii) hold for P, then (i) must hold for P'.

For condition (iii), only the rules for `spawn`, the second two rules for `safepoint`, and the rules for `wake` add elements to either T_a or T_s. Both rules for `spawn` add an element to only one set. As spawn creates a thread, it obviously cannot be currently executing, nor could it have been a member of T_a or T_s. The rule for `wake` removes an element t from T_s, and adds it to T_a. Since (iii) must hold for P, and the $t \in T_s$, then it cannot be in T_a, nor can it be executing. Finally, both the second and the third cases of `safepoint` add an the currently–executing thread t to T_a as an argument to *sched*. However, the *sched* function removes an element from T_a, and replaces the current thread with it. Since (iii) must hold for P, then t could not have been in T_a or T_s, and the element chosen from T_a is either t, or it also could not have been in T_s, nor could it have been executing.

For condition (iii), the only cases in which an executor is added to either E_a or E_s are the first rule for `signal` and the second rule for *sched*. In the first rule for signal, some executor e' is removed from E_s and placed in E_a. In the second rule of *sched*, the reverse occurs: some executor e is removed from E_a and placed in E_s. In both cases, since (iii) holds for P, then it also holds for $P' \mid P \mapsto_P P'$.

For condition (iiii), the only case in which an executor is removed from E_a is the second case of *sched*, when $T_a = \emptyset$. □

The following set of proofs explore the capacity of the execution semantics to implement the weak–epoch property, their inability to implement the full–epoch property.

First, it is necessary to prove that the number of executors remains constant, and that threads are not arbitrarily created or destroyed. These are also important correctness conditions.

Theorem 7.1.2 (Conservation of Executors)**.** *The number of total executors remains constant across the process–execution function $\mapsto_P$. Formally:*
$\forall P = \langle E_a, E_s, \ldots\rangle, [P] = \langle E'_a, E'_s, \ldots\rangle \; (|E_a \cup E_s| = |E'_a \cup E'_s|)$

Proof. The lemma is effectively proven by the proof for condition (iii) of theorem 7.1.1. In all cases in which an executor is added to E_a or E_s, it is first removed from the other set. □

Proving a similar conservation property on threads is slightly more intricate.

Theorem 7.1.3 (Conservation of Threads)**.** *The total number of threads behaves as follows across the process–execution function $\mapsto_P$:*

(i) If $e = \langle$ `spawn` $t.S, \ldots\rangle$, it increases by one.

(ii) If $e = \langle$ `term` $.S, \ldots\rangle$, it decreases by one.

(iii) Otherwise, it remains constant.

Proof. Proof is by definition.

Condition (i) is trivial. There is only one rule for `term`, which discards the current thread. Since the current thread is not in T_a or T_s, by the thread–non–duplication property, the condition holds.

Condition (ii) is similarly trivial. There are only two cases for `spawn`, both of which insert t into either T_a or T_s. Since t is freshly created, it cannot exist in T_a or T_s, nor can it be executing.

For condition (iii), the only rules which add or remove a thread from T_a or T_s are the first rule of `wake`, and the last two rules of `safepoint`, both of which call *sched*. The first rule for `wake` removes a thread from T_s and adds it to T_a. The last two rules of sched add the current thread to T_a or T_s, then immediately remove a thread from T_a and assign it to e. In both cases, the thread is not being added to a set to which it already belonged, by the thread–non–duplication property, so the condition holds. □

Armed with these, it is now possible to prove that the weak–epoch property can be guaranteed, and that the full–epoch property *cannot* be guaranteed.

Theorem 7.1.4 (Impossibility of Full Epoch)**.** *It is* not *possible to implement the full–epoch property, given the semantics of* $\mapsto_P$.

Proof. Proof is by contradiction.

There is no rule in $\mapsto_S$ which changes the current thread of any executor other than e. Therefore, if some thread is being executed by some executor e', it may only be executed if e' takes a step.

However, the $\mapsto_P$ function executes a step from an indeterminant executor. Therefore, it is possible to derive an execution wherein some executor does not take a step for an indefinite number of steps. Any given thread t can be assigned to an executor e, after which e' does not execute, which causes t to also not be executed, which violates the full–epoch property. □

The following definition simplifies the proofs to come.

Definition 7.1.5 (Epoch Bound)**.** If there exists some value b such that once any given thread t is scheduled b times, all other threads must be scheduled at least once before t is run again, then b is an *epoch–bound* on threads.

Likewise, if there exists some value b such that once any given executor e executes b steps, all other executors must execute at least one step before e executes again, then b is an *epoch–bound* on executors.

Remark 7.1.6. A scheduler function with an epoch–bound is called an “epoch–scheduler”. A scheduler with epoch–bound of 1 is called a “round–robin scheduler”.

Theorem 7.1.7 (Weak Epoch)**.** *It is possible to implement the weak–epoch property, given the semantics of* $\mapsto_P$.

Proof. Proof is by example.

Given theorem 7.1.2 and condition (iiii) of theorem 7.1.1, there will always be exactly one executor, and it will remain active so long as there is an active thread.

Assume the *sched* guarantees an epoch–bound b. If the maximum number of threads in the program is bounded by k, then in the worst case, all k threads execute b times. By the alarm–clock property, the executor

```
type lf_queue(a)

bool lf_enqueue(queue : lf_queue(a),
                item : a)
a lf_dequeue(queue : lf_queue(a))
```

```
type scheduler

thread* select(sched : scheduler)
thread* discard(sched : scheduler,
                thread : thread*)
void insert(sched : scheduler,
            thread : thread*)
thread* eschew(sched : scheduler)
```

Figure 7.5: Local Scheduler and Queue Functions

e is signaled at least once every τ_a steps, and by the safepoint–interval property, will acknowledge the signal at least once every $\lceil \frac{\tau_a}{\tau_s} \rceil$ steps. Therefore,

$\tau_e \leq bk\lceil \frac{\tau_a}{\tau_s} \rceil$ □

Theorem 7.1.8. *If the system guarantees an epoch–bound b_e for all executors, then the entire system has the full–epoch property if all executors have the weak–epoch property*

Proof. Proof is by example.

Trivially, this can be proven by having all the schedulers for executors other than e_0, always return the same thread, and have e_0's scheduler schedule all other threads as described in the proof of theorem 7.1.7. Then all threads on e_0 will be executed in at most $\lceil \frac{bk\lceil \frac{\tau_a}{\tau_s} \rceil}{b_e} \rceil$ "rounds" of execution by e_0, therefore

$\tau_e \leq b_e |E_a \cup E_s| \lceil \frac{bk\lceil \frac{\tau_a}{\tau_s} \rceil}{b_e} \rceil$ □

Armed with these theorems, it is now possible to prove the well–behavedness of an implementation of the runtime system. Such a proof must demonstrate that an implementation of each statement is both correct and linearizable, and that the initial state satisfies the correctness conditions.

Additionally, if the system is interactive, the correctness proofs should demonstrate that the *sched* function satisfies the weak–epoch property. The assumption of theorem 7.1.8 holds for most real systems, therefore, proving the weak–epoch property effectively proves the full–epoch property as well.

Now that the theoretical model has been established, section 7.2 demonstrates a lock–free implementation of this model. Following that, section 7.4 proves the correctness of this implementation.

7.2 Scheduling Algorithm

This section specifies the scheduling algorithm to the point where its correctness can be proven. The algorithm assumes the existence of a local scheduling algorithm and a lock–free queue. Figure 7.5 shows declarations of several functions which are expected to be defined by the implementation. The local scheduler is used in a purely synchronous context, and may make use of arbitrarily complex data structures. The lock–free queue interface is designed to support the queues described by section 7.3.

The `select` scheduler function cycles the scheduler, returning the thread to execute and returning the previously executing thread to its structures. The `discard` function removes the given thread from the scheduler. The `insert` function adds a thread to the scheduler. Lastly, the `eschew` function selects and

```
void safepoint(id : uint) =
  mbox = executors[id]
  do
    old = mbox.run_scheduler
  while !CAS(old, 0,
             executors[id].mbox)
  if old.run_scheduler
    sched(id)
  ...

void idle(id : uint) =
  forever
    safepoint(id)
    sigsuspend(IDLE_MASK);
```

```
void raise(id : uint) =
  mbox = executors[id]
  do
    old = mbox.run_scheduler
    new = old
    new.run_scheduler = true
  while !CAS(old, new,
             executors[id].mbox)

void signal() =
  for (i = 0; i < execs; i++)
    if(current(executors[i].sched) ==
       NULL)
    exec = executors[i].os_exec
    raise(id)
    kill(exec, WAKEUP_SIGNAL)
    break
```

Figure 7.6: The signal, safepoint and raise Statements, and the Idle Thread.

removes a thread from the scheduler to be sent to the workshare queue. The queue functions are typical lock–free queue functions, except that `lf_enqueue` may fail if no free nodes are available, in which case it returns `false`. The reasons for this are made clear in section 7.3.

Figure 7.6 shows some of the more basic components of the scheduling algorithm. The `safepoint` function implements a check of the safe–point, using a compare–and–set to atomically check and clear the mailbox. The `raise` function is similar, except it sets the `run_scheduler` portion of the mailbox (the only portion represented in the theoretical model) to true. The `signal` function uses a `kill` call, which is semantically identical to the POSIX function of the same name. Lastly, the `idle` thread repeatedly executes a safepoint, then suspends using `sigsuspend`. This is a slight divergence from the theoretical model, in which the idle thread would immediately yield back to the scheduler. However, this accounts for the fact that the idle thread may exit the `sigsuspend` for reasons other than being `signal`led. The `sigsuspend` call assumes the POSIX semantics of atomically setting the signal mask (which enables receipt of `WAKEUP_SIGNAL`) and suspending.

Figure 7.7 shows the implementation of the `spawn` statement, as well as the `update` function, which implements both the `wake` and `sleep` statements. Note that `spawn` takes several arguments: an initial state, an initial mailbox state, and a pointer. The initial state arguments are straightforward. The pointer is an address to which a pointer to the thread's mailbox should be stored. This is used to initialize the entry point of the thread. Since a thread could be executed prior to the return of `spawn`, and the address of the thread mailbox obviously is not known until the thread is created, `spawn` must store the thread's mailbox to this address after allocating the thread structure, but before activating the thread. Note that `spawn` uses `wake` to insert the new thread into the queues. As will be demonstrated, new threads are implicitly in the T_s set, so `wake` preserves correct semantics.

The `wake` function sets the reference flag to true, indicating that the thread is present in the scheduler's structures, and the status to `RUNNABLE` using atomic compare and set. If the reference flag was not previously set, then the thread is inserted into the lock–free workshare queue if a node is available, and into the scheduler

```
thread* spawn(initial_state : state, // RUNNABLE or SUSPEND
              initial_mbox : thread_mbox,
              mbox_addr_ptr : thread_mbox**) =
  thread : thread* = allocate()
  thread.stat_ref.ref = false
  thread.stat_ref.state =
    state == RUNNABLE ? NONE : SUSPEND
  thread.mbox = initial_mbox
  *mbox_ptr_addr = thread.mbox
  if(state == RUNNABLE)
    wake(thread)
  thread

void wake(thread : thread*, exec : executor*) =
  do
    oldstat = thread.stat_ref
    newstat.ref = true
    newstat.state = RUNNABLE
  while ((oldstat.state != RUNNING && oldstat.state != RUNNABLE) ||
        !CAS(oldstat, newstat, thread.stat_ref))
  if(oldstat.stat != RUNNABLE &&
     oldstat.stat != RUNNING)
    atomic_inc(active_threads)
  if (!oldstat.ref)
    if(!lf_enqueue(workshare, thread))
       insert(exec.scheduler, thread)

void sleep(thread : thread*) =
  do
    oldstat = thread.stat_ref
    newstat = oldstat
    newstat.state = SUSPEND
  while ((oldstat.state != SUSPENDED && oldstat.state != SUSPEND) ||
        !CAS(oldstat, newstat, thread.stat_ref))
  if(oldstat.stat == RUNNABLE && oldstat.stat == RUNNING)
    atomic_dec(active_threads)
```

Figure 7.7: The spawn, wake, and sleep Statements

```
bool setrunning(thread : thread*) =
  out = false
  cas = true
  while(cas)
    oldstat = thread.stat_ref
    switch(oldstat.stat)
    case RUNNING, RUNNABLE:
      newstat.state = RUNNING
      newstat.ref = oldstat.ref
      out = CAS(oldstat, newstat, thread.stat_ref)
      cont = !out
    case SUSPEND:
      newstat.state = SUSPENDED
      newstat.ref = false
      cont = !CAS(oldstat, newstat, thread.stat_ref)
  out

void try_workshare(exec : executor*) =
  sched = exec.sched
  while(NULL == current(sched))
    thread = lf_dequeue(workshare)
    if(setrunning(thread)
      exec.current = thread
      balance(sched)

void balance(sched : scheduler*) =
  if(count(sched) < lower_bound(active_threads))
    take_threads(sched)
  else if(count(sched) > upper_bound(active_threads))
    give_threads(sched)
  else
    swap_threads(sched)
```

Figure 7.8: Scheduler Support Functions

of the executor which created the thread otherwise. If the flag is set, then the thread is already present somewhere in the scheduler structures. If the state is changing from non–runnable to runnable, then the `active_threads` value is incremented.

The `sleep` function sets the thread's state to `SUSPEND` using a compare and set, the decrements the `active_threads` value if the status has gone from a runnable state to a non–runnable one.

Figures 7.8 and 7.9 show the main scheduler functions. The `cycle` and `replace` functions implement the two *invocations* of the *sched* function: considering and discarding the current thread respectively. The `term` statement is also implemented by `replace`. The only difference between either cases is the use of the `select` or `remove` functions, which either keep or discard the current thread. Both functions make use of the `try_workshare` function, which attempts to pull a thread from the workshare queue. The `balance` function is similar, and either pushes work to the workshare, or pulls work from it depending on the number of threads in the current scheduler. The `cycle_threads` branch is included to facilitate the epoch properties. The `give_threads`, `cycle_threads` and `take_threads` functions are not shown. These functions enqueue a

```
void cycle(exec : executor*) =
  sched = exec.sched
  do
    thread = select(sched)
    finished = false
    if(thread != NULL && setrunning(thread))
      finished = true
      setcurrent(sched, thread)
      balance(sched)
    else
      thread = discard(sched, thread)
  while(!finished && thread != NULL)
  try_workshare(exec)

void replace(exec : executor*) =
  sched = exec.scheduler
  discard(sched, exec.current)
  do
    thread = select(sched)
    finished = false
    if(thread != NULL && setrunning(thread))
      finished = true
      exec.current = thread
      balance(sched)
    else
      thread = discard(sched, thread)
  while(!finished && thread != NULL)
  try_workshare(exec)

void term(exec : executor*) =
  thread = exec
  replace(exec)
  // safely free thread
```

Figure 7.9: The Main Scheduler Functions

thread gotten from the local scheduler using `eschew`, or dequeue a thread from the workshare respectively[5].

Lastly, all of the functions make use of the important `setrunning` function. This function is actually quite important from the standpoint of proving linearizability. This function uses a compare and set to set the thread status to `RUNNING` if the thread is runnable. It also sets the status, and more importantly, unsets the reference flag if the thread is not runnable.

Now that the algorithm has been described, section 7.4 presents proofs of its correctness by the criteria defined in section 7.1. Once the core has been proven correct, section 7.5 demonstrates useful derived functionality based on the core statements. Prior to presenting the correctness proofs, however, section 7.3 discusses a modification to the core algorithm to significantly reduce the number of dynamic memory allocations.

[5]In the real implementation, they check if a thread is still runnable, and attempt to compare–and–set its state and discard it otherwise.

7.3 Lock-Free Worksharing Without Dynamic Allocation

Both the scheduling system and the garbage collector utilize a lock–free fifo queue as a workshare mechanism. In both systems, executors maintain a set of tasks using purely–local data structures, and share work by periodically moving tasks to or from a shared, lock–free data structure.

In both cases, the runtime implements this structure as a FIFO–queue. In the scheduler, doing so prevents thread–starvation. In the garbage–collector, the local work queue is implemented as a FIFO–queue in order to (hopefully) improve memory locality. Though the exact alignment of objects which are sent to the workshare is indeterminant, using shared FIFO–queue hopefully preserves as much of the original behavior as possible.

The lock–free FIFO algorithm of Michael and Scott[99] can be used as the basis for the workshare queues used by the runtime system. Michael and Scott's queue also requires one of several additional mechanisms to safely reclaim nodes and threads[97, 61, 35, 55, 95], and to prevent the ABA problem[6][96]. Unfortunately, these queues, as well as the safety mechanisms they employ require the use of dynamic allocation. While this is not a problem in theory, requiring dynamic allocation in a scheduler cycle or to process an object in a garbage collection cycle will result in high contention on the dynamic allocator and poor performance in the scheduler or garbage collector. A solution to these problems arises from the fact that, being a workshare queue, it is not *absolutely* necessary that every enqueue operation succeed. It is entirely acceptable for the queue to fill up and reject an attempt to enqueue another task. If the queue is full, then an executor simply keeps the task in its local structures, as work–starved executors clearly have enough tasks to keep them busy. This permits a static supply of nodes to be allocated during initialization and used during execution.

7.4 Correctness

This section presents proofs of the correctness of the lock–free scheduling algorithm presented in section. These proofs are presented separate from the algorithm itself, so as to facilitate better understanding of the algorithm, and because of the fact that the ability to prove correctness depends in part on properties of the entire algorithm, not just one component of it.

The following definitions formally state properties that were stated in section 7.2, as well as some new properties.

Definition 7.4.1 (Assignment to Executors)**.** A thread t is assigned to an executor e if the following conditions hold:

(i) It is inevitable the `current` field of e will be t at some point in the future, and no other thread satisfies this condition at a point farther in the future.

(ii) The `ref` field is `true`

Definition 7.4.2 (Idle Thread)**.** A `current` value of `NULL` denotes that an executor is executing the idle thread ω.

Definition 7.4.3 (Membership in E_a)**.** An executor is in E_a if any of the following conditions is met:

[6]ABA is a problem which arises from the use of compare–and–set operations wherein a new node that happens to have the same address as a node that was recently removed causes a compare–and–set operation to erroneously succeed.

(i) Its current thread is not ω

(ii) Its signal state is asserted and it will execute a `safepoint` instruction in a finite amount of time.

Definition 7.4.4 (Membership in E_s)**.** An executor is in E_s if and only if all the following conditions are met:

(i) It will acknowledge signals sent by the `kill` function in a finite amount of time.

(ii) It does not satisfy any condition of definition 7.4.3

Definition 7.4.5 (Membership in T_a)**.** A thread is in T_a if and only if the following conditions are met:

(i) The `stat_ref.state` field is either `RUNNING` or `RUNNABLE`

(ii) The `ref` field is `true`

(iii) The thread is observably present in either `workshare` or a scheduler.

(iiii) The thread is not assigned to any executor, as per definition 7.4.1

Definition 7.4.6 (Membership in T_s)**.** A thread is in T_s if and only if the following conditions are met:

(i) The thread is not in T_a, as per definition 7.4.5

(ii) The thread is not assigned to any executor, as per definition 7.4.1

(iii) The thread's status is not `NONE`.

The following lemma is trivial, but very important, as it proves that access to schedulers is wholly synchronous.

Lemma 7.4.7. *An executor's* `sched` *field is never accessed by another executor.*

Proof. Schedulers are only modified in `cycle`, `replace`, and `spawn`, and only the current executor's scheduler is modified. □

The following lemma establishes the legal transitions of `stat_ref`, which proves a powerful tool in the following proofs.

Lemma 7.4.8. *The only possible state transitions for* `stat_ref` *are:*

(i) `(NONE, false)` $\Rightarrow$ `(RUNNABLE,true)`

(ii) `(SUSPENDED, false)` $\Rightarrow$ `RUNNABLE,true`

(iii) `(SUSPEND, false)` $\Rightarrow$ `RUNNABLE,true`

(iiii) `(SUSPEND, true)` $\Rightarrow$ `RUNNABLE,true`

(v) `(SUSPEND, true)` $\Rightarrow$ `SUSPENDED,false`

(vi) `(RUNNABLE, true)` $\Rightarrow$ `SUSPEND,true`

(vii) `(RUNNABLE, true)` $\Rightarrow$ `RUNNING,true`

(viii) `(RUNNING, true)` ⇒ `SUSPEND,true`

Proof. The `stat_ref` variable is updated only by compare–and–set, therefore only those transitions which arise from possible values of the old or new values are possible.

First, observe that all transitions explicitly set the state. Therefore, the only transitions to a state are those that explicitly set that state.

Second, observe that no transition affects a thread regardless of its initial state.

The only place where the reference flag is explicitly set to `true` is in `wake`, but `state` is also set to `RUNNABLE`, it is impossible to transition from any state in which `ref` is false to any state in which `ref` is true, and `state` is anything other than runnable. Likewise, this is the only transition which sets `state` to `RUNNABLE`, so it is impossible to transition to any state in which `stat_ref` is `(RUNNABLE,false)`.

The only place where the `ref` flag is set to `false` is in `setrunning`, and also sets `state` to `SUSPENDED`. Note that this only occurs if the original `state` is `SUSPEND`. Likewise, this is the only transition to any state in which `state` is explicitly set to `SUSPENDED`, therefore, it is impossible to transition to any state in which `stat_ref` is `(SUSPENDED,true)`

A thread is assigned the `NONE` state only in the `spawn` function, when it is created (if the initial state is runnable), and `ref` is set to false. Note that it is immediately set to `(RUNNABLE,true)`, and that the thread is invisible to the outside world until `spawn` inserts it into the workshare queue, which occurs after this transition. Therefore, (i) represents the only transition in which `state` is `NONE`.

No transition other than the one in `wake` can affect a thread which is `SUSPENDED`. Given that it is impossible to transition to `(SUSPENDED,false)`, (ii) represents the only transition where `state` is `SUSPENDED`.

The only place where `state` is set to `SUSPEND` is in `sleep`. This transition preserves the `ref` flag, so it is possible that `(SUSPEND,true)` may arise. However, by previous observations, it is impossible to transition to `(RUNNABLE,false)` or `(SUSPENDED,true)`. Additionally, the only transition to `RUNNING` requires the original state to be `RUNNABLE`. Finally, observe that `(SUSPEND,false)` may arise if a thread is created in a suspended state. Therefore (iii), (iiii) and (v) represent the only transitions where `state` is `SUSPEND`.

The only transitions which affect a thread which is `RUNNABLE` are in `sleep` and `setrunning`. Both preserve `ref`, and set `state` to `SUSPEND` and `RUNNING` respectively. Since there is never any transition to `(RUNNABLE,false)`, (vi) and (vii) represent the only transitions where `state` is `RUNNABLE`.

Lastly, the only transition to `RUNNING` requires the initial state to be `RUNNABLE`, and preserves the `ref` flag, which, by earlier observations, must be `true`. The possible only transition from `RUNNING` is in `sleep`, and sets `state` to `SUSPEND`, preserving `ref`. Therefore, (vii) represents the only transition where `state` is `RUNNING`.

This covers all possible starting states. □

Corollary 7.4.9. *The following states of* `stat_ref` *cannot be reached:*

(i) `(NONE,true)`

(ii) `(RUNNING,false)`

(iii) `(RUNNABLE,false)`

(iiii) `(SUSPENDED,true)`

Proof. These are an immediate consequence of the legal state transitions. □

Now that the legal state transitions are known, it is also useful to prove the exact meaning of the `ref` field. The following lemmas set the stage for this.

Lemma 7.4.10. *A given thread is inserted into* `workshare` *or a scheduler only after a* `stat_ref` *transition in which* `ref` *goes from* `false` *to* `true`

Proof. The only point at which a thread is inserted either into `workshare` or into a scheduler immediately follows a transition which explicity sets `ref` to `true`, and the insertion is only done if `ref` was previously false. □

Corollary 7.4.11. *After any transition in which* `ref` *goes from* `false` *to* `true`*, a thread is inserted into* `workshare` *or a scheduler.*

Proof. The only transition in which `ref` is explicitly set to `true` precedes an insert into either `workshare` or a scheduler. □

Lemma 7.4.12. *A given thread undergoes a* `stat_ref` *transition in which* `ref` *goes from* `true` *to* `false` *only when it is not observably present in* `workshare` *or a scheduler structure, or is not assigned to a scheduler.*

Proof. The only transition which explicitly unsets `ref` is in `setrunning`, which causes the function to return `false`, causing the current thread to be `discard`ed in both `cycle` and `replace`, and not to be chosen in `try_workshare`. Therefore, all transitions to `ref` being `false` follow removing a thread from `workshare`, and result in it being discarded. Since access to all schedulers is purely thread–local by lemma 7.4.7, it is inevitable that it will be `discard`ed after its state has been set, and no other thread will observe it as being in the scheduler structure. Therefore, a `ref` is only set to `false` if the thread cannot be observed to be in any structure. □

Lemma 7.4.13. *Any thread assigned to a scheduler has a* `stat_ref.ref` *value of* `true`.

Proof. A thread is assigned to an executor only after the `setrunning` function successfully completes a transition to the `RUNNING` state. By corollary 7.4.9, the `stat_ref` value `(RUNNING, false)` is impossible. This, combined with lemma 7.4.12 proves the condition. □

This lemma proves that presence in the queues or schedulers implies `ref` is true, and also proves a very important correctness condition: the absence of observable duplicate entries.

Lemma 7.4.14. *If a given thread is observably present in any scheduler or* `workshare`*, its* `stat_ref.ref` *is* `true`.

Proof. Immediate from lemmas 7.4.10, 7.4.13, and 7.4.12 and corollary 7.4.11. □

Corollary 7.4.15. *The scheduler structures and* `workshare` *contain no observable duplicate entries.*

Proof. This is an immediate consequence of lemmas 7.4.14 and 7.4.10. □

The previous lemma was very important, as it establishes one of the key conditions for correct execution. The following proves the other direction of the relationship between `ref` and presence in the data structures.

Lemma 7.4.16. *If the* `stat_ref.ref` *field is set to* `true` *for any thread, then it is inevitable that the thread will be observably present in* `workshare` *or a scheduler structure, or will be assigned to an executor at some point in the future. Likewise, when a thread's* `stat_ref.ref` *field is set to* `false`*, it is no longer observable in* `workshare`*, in a scheduler or assigned to an executor.*

Proof. The first part is proven by lemma 7.4.11.

The only points at which a thread is removed from `workshare` are located in `try_workshare`, and `take_thread`. In both cases, the thread is only discarded after its `ref` transitions to `false` Otherwise, it is inserted into the scheduler.

In the scheduler functions `cycle` and `replace`, a thread is only discarded after its `ref` transitions to `false`.

The only other point at which a thread is removed from a scheduler is `give_thread`. It is only discarded after its `ref` transitions to `false`. Otherwise, it is inserted into `workshare`.

In all these cases, the transition occurs when the thread is present in a scheduler. Since scheduler access is purely local, by lemma 7.4.7, the thread is no longer observably present in the scheduler, since the transition marks the decision to remove it. Therefore, the lemma holds. □

This provides the final piece of a very important lemma, equating the `ref` field with presence in the scheduler structures. Several important corollaries follow from this.

Lemma 7.4.17. *A thread's* `stat_ref.ref` *is* `true` *if and only if the thread is observably present in scheduler structures, or inevitably will be at some point in the future.*

Proof. This is a direct consequence of lemmas 7.4.14 and 7.4.16, and corollary 7.4.15. □

Finally, in preparation for proving correctness and linearization of the statement implementations, the following lemmas prove correctness and linearization for the scheduler functions.

Lemma 7.4.18. *The* `replace` *function discards the current thread, removes a thread from T_a and assigns it to the scheduler, or suspends the executor if there are no threads in T_a, and is linearizable.*

Proof. The `replace` function always calls `discard` on the scheduler, and all paths replace `current` with some thread. The thread which is selected is observed to be in either `RUNNABLE` or `RUNNING` states transitions to `(RUNNING,true)`, at which point it becomes inevitable that the thread will be assigned to the current executor. Obviously this thread is present in the scheduler structures. Because the transition to `(RUNNING,true)` simultaneously causes condition (iiii) of membership in T_a to be violated and satisfies the definition of assigning a thread to an executor by 7.4.1. This action also causes the current thread to no longer be assigned to the current executor. Therefore, the action simultaneously satisfies all conditions, making it the linearization point when a new thread can be found.

If no thread is found which can be assigned to the executor, then assigning `NULL` to `current` simultaneously discards the current thread and sets the executor's thread to ω. Since there is a finite number of steps until the executor will execute `sigsuspend`, condition (i) of membership in E_s is fulfilled. Since there is no finite upper bound on the amount of time before a `safepoint` is executed, both options for membership in

E_a are violated, which in turn satisfies condition (ii) for membeship in E_s. Thus, the executor is simultaneously removed from E_a and placed in E_s. Thus, the linearization point comes when `NULL` is assigned to the `current`. □

Corollary 7.4.19. `replace` *returns the current thread to T_a or T_s as appropriate, removes a thread from T_a and assigns it to the scheduler, or suspends the executor if there are no threads in T_a, and is linearizable.*

Proof. The proof of `replace` proves the corollary, with the observation that `cycle` does not discard the current thread, but rather that the compare–and–set to `(RUNNING,true)` either retains the current thread, causes it to be inserted in T_a if its state is `RUNNING` or `RUNNABLE`, or causes it to be inserted into T_s if its status is `SUSPEND` or `SUSPENDED`. □

Now, with all supporting lemmas proven, it is possible to begin the main proofs of correctness. The primary correctness conditions come from proving that each implementation performs the required operations and is linearizable.

Theorem 7.4.20 (Correctness of wake)**.** *The* `wake` *function correctly implements all cases, and is linearizable.*

Proof. The first case of `wake` removes the thread from T_s and inserts it into T_a. This is accomplished when the thread is inserted either into the scheduler, or into `workshare`, as all conditions of definiton 7.4.5 are satisfied, and condition (ii) of definition 7.4.6 is violated. Since at the return of `lf_enqueue`, the thread is either in `workshare`, or will inevitably be inserted into the scheduler, which, by lemma 7.4.7 means it is observably in the scheduler at that point, the return of `lf_enqueue` is the linearization point for this case.

In the second case, wake observes the thread to be in T_a and does nothing. The linearization point is when it fetches the thread's `stat_ref.state` as either `RUNNABLE` or `RUNNING`. □

Corollary 7.4.21. *If the thread's* `stat_ref.state` *is* `NONE`*, the* `wake` *function inserts it into T_a.*

Proof. The proof is subsumed by the proof of the first case of the model's behavior. □

Theorem 7.4.22 (Correctness of term)**.** *The* `term` *function correctly implements all cases, and is linearizable.*

Proof. There is one case for `term`, which has exactly the behavior of `replace` by lemma 7.4.18. Therefore, the implementation is correct, and the linearization point is the call to `replace`. □

Theorem 7.4.23. *The* `safepoint` *function correctly implements all cases, and is linearizable.*

Proof. The first case of `safepoint` is trivial. The linearization point is the compare–and–set of the executor's `mbox`.

The behavior of the second case is exactly the behavior of the `cycle` function, so the linearization point is at the call to `cycle`.

Likewise, the behavior of the third case is exactly the behavior of the `replace` function, making the linearization point the call to `replace`. □

Theorem 7.4.24 (Correctness of spawn)**.** *The* `spawn` *function correctly implements all cases, and is linearizable.*

Proof. There are two cases for `spawn`, one in which the first thread is `RUNNABLE`, and one in which it is not.

In the case of the thread being `RUNNABLE`, `wake` implements the proper behavior by inserting the thread into T_a, making it the linearization point for this case.

In the case where the thread begins in the `SUSPEND` state, it is purely local until returned, so the linearization point is the return of the function. □

Theorem 7.4.25 (Correctness of raise)**.** *The* `raise` *function correctly implements all cases, and is linearizable.*

Proof. Both cases of `raise` are implemented the same way. The linearization point is a successful compare–and–set of the `mbox` for the executor. □

Theorem 7.4.26 (Correctness of signal)**.** *The* `signal` *function correctly implements all cases, and is linearizable.*

Proof. Signal has two cases, one in which a suspended executor exists, and one in which it doesn't.

In the first case, the `signal` function locates a suspended executor, does a compare–and–set to assert `run_scheduler` in its mailbox, then sends an OS signal to the thread using `kill`. Since the only time a flag in the mailbox is unasserted is when the executor itself executes a `safepoint`, and the executor is `sigsuspended`, or will be in a finite number of steps, this field can be treated as though it were local. Once the call to `kill` completes, the target executor's `run_scheduler` is asserted, and it will execute a `safepoint` upon resuming execution, thus it now satisfies condition (ii) of membership in E_a, and ceases to satisfy condition (ii) of membership in E_s. Thus, the linearization point is the call to `kill`.

In the second case, the linearization point comes when the `signal` function completes a snapshot in which there are no suspended executors. □

Theorem 7.4.27 (Correctness of sleep)**.** *The* `sleep` *function correctly implements all cases, and is linearizable.*

Proof. All cases of `sleep` are implemented in the same way, despite the statement's complexity in the model. A successful compare–and–set to `SUSPEND` implements correct behavior in the first three cases. In the first case, the compare–and–set simultaneously violates condition (i) of definition 7.4.5 and satisfies condition (i) of definition 7.4.6. In second and third cases, it sets the state. Therefore, in the first three cases, the successful compare–and–set is the linearization point.

In the fourth case, the linearization point comes when the value of `stat_ref.state` is fetched as `SUSPEND` or `SUSPENDED`, as it becomes inevitable that the function will do nothing. □

This completes the proofs of the various statement implementations. Next, the initial state must be proven to satisfy the correctness requirements.

Definition 7.4.28 (Initial State)**.** In the initial state, a single thread is assigned to one of the executors, and all other executors are idle. The resume continuation of the initial thread is the starting state of the program.

Theorem 7.4.29 (Correctness of Initial State)**.** *The initial state satisfies the criteria for validity.*

Proof. The executor- and thread–non–duplication properties are inherently satifsied by definitions 7.4.3, 7.4.4, 7.4.1, 7.4.5, 7.4.6.

The unique–execution property is satisfied by the fact that only one thread exists. The no–deadsleep property is satisfied by the fact that one executor is active. □

With this, the theorems proven in section 7.1 now apply fully to the runtime system. Finally, it is useful to prove some properties of interactive scheduling. The following remarks set the stage for proving the weak–epoch property.

Remark 7.4.30. The native execution environment is assumed to implement the process–execution function $\mapsto_P$ and the alarm–clock property.

Remark 7.4.31. The compiler is assumed to implement the safepoint–interval property.

Theorem 7.4.32 (Weak Epoch)**.** *The algorithm can implement the weak–epoch property under the following assumptions:*

(i) Schedulers implement the weak–epoch property

(ii) The `eschew` *function retains additional information in each thread about when it was last executed.*

(iii) Once a thread is inserted into a scheduler, `eschew` *will not remove it until it has been executed at least once.*

Proof. First, observe that `balance` is called at the end of every scheduler function. Second observe that the only case in which `balance` does not take a thread from `workshare` is when it has more than its upper bound, which implies that there is at least one executor which has fewer threads than the upper bound, assuming the upper bound is well–chosen. Therefore, at least one thread is removed from `workshare` per scheduling cycle, which implies that there is a finite number of scheduler cycles for which a thread inserted into `workshare` will remain in it until dequeued.

Given this observation and the assumptions of the theorem, it follows that the weak–epoch property holds for the entire system. □

7.5 Derived Forms

Previous sections defined a formal model of M:N scheduling, gave a lock–free implementation of it, and proved the correctness of the implementation. However, the primitives provided by this model are not very useful by themselves for concurrent programming. Indeed, these primitives are meant to be used as a basis for implementing more complex synchronization mechanisms, not to provide the entire model by themselves. This section explores several derived constructs built on the base model.

Figure 7.10 shows a basic non–spinning mutex implementation. The example here is meant to demonstrate the concept, not to provide an efficient implementation. This implementation makes use of the fact that `sleep` does not actually suspend a given thread immediately but rather sets a flag, causing it to be suspended on the next safe–point. The ability to perform operations atomically within the scope of a single thread simply by omitting safe–points is another common technique, which is also used here. These techniques are

```
type mutex = < holder : thread*,
               queue : thread* >

void unlock(m : mutex) =
  do
    oldm = m
    (holder, queue) = oldm
    if(NULL == queue)
      newm.holder = NULL
      newm.queue = NULL
    else
      newm.queue = queue.next
      newm.holder = queue
  while(!CAS64(oldm, newm, m))
  wake(queue)
```

```
void lock(m : mutex) =
  sleep(self)
  do
    oldm = m
    (holder, queue) = oldm
    if(NULL == holder)
      newm.holder = self
      newm.queue = queue
    else
      self.queue_next = queue
      newm.queue = self
      newm.holder = holder
  while(!CAS64(oldm, newm, m))
  if(NULL == holder)
    wake(self)
  else
    raise(self.executor)
    safepoint();
```

Figure 7.10: A Basic Mutex Implementation

used together to set a thread's status to SUSPEND then declare itself as waiting on a structure in order to allow another thread to wake it without losing the wakeup. This technique also integrates well with more efficient implementations of mutexes using efficient spin–locks[7], delayed blocking [43], and other techniques.

The mutex implementation shown above is necessary for implementing complex synchronization, such as the choose combinator in CML. Additionally, the mutex can be modified easily to implement synchronous channels, asynchronous channels, futures, and other constructs. Most of these modifications involve simply changing the structure of the mutex object to facilitate the desired behavior. For instance, in the case of synchronous channels, there are two queues instead of a "holder" and a queue. Each operation checks the other queue, dequeuing an operation if one exists, and enqueuing itself otherwise.

Implementing the choose combinator is significantly harder, requiring the ability to simultaneously solve multiple consensus problems. There is no efficient implementation of choose with synchronous 2–way channels using only single compare–and–set[90]. The choose combinator can be easily implemented using the monadic style of software transactional memory[53], and is slightly more difficult to implement using mutexes. As I have already demonstrated an implementation of mutexes, which can serve as a mechanism for implementing monadic STM or for implementing choose directly, there is no need to discuss it further.

It is also possible to provide the ability to terminate a thread asynchronously, and to retain dead threads (sometimes known as "zombies") until they are marked for destruction. Indeed, the second change is necessary in order to safely reclaim memory held by threads when threads are explicitly destroyed. Threads cannot be simply destroyed, as there is no way of removing destroyed threads from the scheduler structures. Instead, they must be marked for destruction, but only truly destroyed when removed from scheduler structures. Additionally, a thread cannot even be marked for destruction until there are no live references to it in the program as well. In a garbage–collected program, the same technique proposed by Reppy for garbage–collecting suspended threads[130] can be put to use. However, as the runtime is intended to also support the

```
bool setrunning(thread : thread*) =
  out = false
  cas = true
  while(cas)
    oldstat = thread.stat_ref
    switch(oldstat.stat)
    case RUNNING, RUNNABLE:
      newstat.state = RUNNING
      out = CAS(oldstat, newstat, thread.stat_ref)
      cont = !out
    case SUSPEND:
      newstat.state = SUSPENDED
      newstat.ref = false
      cont = !CAS(oldstat, newstat, thread.stat_ref)
    case TERM:
      newstat.state = DEAD
      newstat.ref = false
      cont = !CAS(oldstat, newstat, thread.stat_ref)
    case DESTROY:
      cont = false;
      destroy(thread)
  out
```

Figure 7.11: Modifications for Thread Termination and Destruction

execution of program segments written in non garbage–collected languages, it is also necessary to support the safe explicit destruction of threads.

Figure 7.11 shows the `setrunning` function modified to support three new thread statuses: `TERM`, `DEAD`, and `DESTROY`. The `TERM` and `DEAD` states are identical to the `SUSPEND` and `SUSPENDED` states, except that once a thread enters either state, it can no longer be set to `RUNNABLE` again. Additionally, the garbage–collector will not preserve the frame in the resume continuation for these states (unless a thread marked `TERM` is still executing, which is possible with asynchronous termination). The `DESTROY` state is similar to `TERM` or `SUSPEND` in that it is unacknowledged. Threads which are maked `DESTROY` will be destroyed when a transition is made to a `stat_ref` state of `(DESTROY,false)`.

Lastly, the ability to send and receive asynchronous signals is present in many operating systems. This ability can be provided to threads in the runtime with minor modification to the runtime in one of two ways. In the first, when signals are received, they are recorded in an atomically modified bitmap for the given executor, and flag is set in the executor's mailbox, indicating that OS signals have been received. At the next safe–point, the executor executes a signal handler for the given signal.

The second method takes advantage of the fact that the scheduler and memory allocator are both completely lock–free, and therefore, reentrant. The signal handler for a given signal simply creates a thread from a given function. The simplicity with which interrupt threads can be implemented is a significant advantage of lock–freedom. Non lock–free systems must work much harder to implement the same functionality. It bears mention that the signal mask and os signal fields in an executor mailbox may still be necessary, depending on exactly how executors and signals are implemented.

Asynchronous signaling between threads can also be implemented by adding a signal mask to a thread mailbox. When a thread executes a safe–point, it checks its own mailbox for signals and acts accordingly. The actual sending of signals can be implemented as a modification to `wake`. The signaling function sets the thread's signal mailbox atomically, then performs the usual actions of `wake`. Since `wake` does nothing to an active thread, and because `wake`ing the target in between setting the mailbox and calling `wake` has the same effect, the linearization point is when the mailbox is set.

This chapter has discussed the threading system for the runtime in great detail, presenting a formal semantic model, a concrete algorithm, proofs of correctness, and derived forms. The threading system represents what is arguably the most intricate portion of the runtime system. The other major component, the garbage collector is discussed in chapter 8.

Chapter 8

Garbage Collection

This chapter describes the mechanisms for performing dynamic allocation and garbage collection in the runtime system. The garbage collection algorithms are derived chiefly from the Cheng–Blelloch collector[27], but also from other collectors. Unlike previous collectors, however, this particular garbage collector is mostly lock–free, requiring only barrier synchronizations when switching modes.

The mode–switching is a fundamental barrier to a lock–free garbage collector. Any algorithm which requires mode–switches to be acknowledged by all threads is inherently not lock–free. All collectors of which I am aware at the time of writing observe this structure. Though I propose ideas for a wholly lock–free "always–on" collector as future work (chapter 15), such a collector is beyond the scope of this thesis. The overhead of two or three barriers (especially if done right) is a perfectly reasonable and almost negligible cost for an otherwise lock–free collector.

Chapter 7 described a formal model for the threading semantics. This was necessary in order to properly describe the threading model and prove its correctness, primarily due to the sheer complexity of precise threading semantics and the difficulties inherent in reasoning about concurrent programs. This chapter does *not* develop such a model for garbage collection for several reasons. First, the correctness conditions of a garbage–collector are easier to state, and the "interface" for a garbage collector is markedly simpler, and can be stated much more succinctly. Additionally, there is already a body of work dealing with the formal modeling and correctness of garbage collectors[77, 92, 148, 146]. The intended contribution of this thesis with regard to garbage collection is a mostly lock–free garbage collector, not a formal model of garbage–collection. The primary intent in forming proofs of correctness for these algorithms is to guarantee that the lock–free alorithms used herein do not misbehave, rather than to construct a formally–correct garbage–collector.

Section 8.1 presents an overview of the collector, while section 8.2 presents the exact algorithms. Proofs of the correctness of these algorithms are in section 8.3.

8.1 Background and Overview

Garbage collection is a programming model wherein objects are allocated dynamically, and released when they will no longer be used by the program. This is in contrast to `malloc`–style explicitly–freed memory, which must be explicitly released using a `delete` or `free` function. Garbage collection is arguably beneficial for performance even in a single–threaded context[69, 9]. However, in the context of concurrent programming, particularly with lock–free data structures, transactional memory, and other such mechanisms, garbage collection easily solves a serious problem. The lock–free reclamation problem arises from the fact that once an object or node is "deleted" from a lock–free structure, there may still exist threads with references to it. In a non garbage–collected environment, this requires one of several solutions [97, 96, 61, 35, 74, 95] Additionally, as shown in chapter 5, garbage collection has a synergistic effect with other constructs in the runtime.

Effective garbage–collection uses a mark–sweep–copy approach, in which the collector begins with a "root set" of references, and recursively traces the objects, marking them as reachable until it has exhausted the entire memory graph. Objects which are marked reachable are copied into a new memory space, which becomes the active space. The previous active space is then treated as free space. The mark–sweep–copy approach has the advantage of improving memory locality of objects and avoiding fragmentation altogether. As previously mentioned, allocation is done simply by advancing a pointer.

Generational collection is a further enhancement to the mark–sweep–copy approach. In a generational collector, objects are grouped into "age–classes", and most collections deal only with the youngest age–class. This is based on the notion that most objects are short–lived, while objects which survive long enough are likely to survive for an even longer time. Generational collection reduces the overhead of most collections by confining them to the youngest age–class. It also improves locality further by clustering objects together based on their age (and likely usage patterns).

Semi–space collection is a technique used by non–parallel collectors to reduce the overhead of a single collection. In a semi–space collector, the collection is divided into several segments. This reduces the pause in execution incurred by the collector. Even better than the semi–space approach, however, are parallel collectors. These collectors allow the program to execute while other threads perform the collection. Parallel collectors have the benefit of fully exploiting the parallelism of the system, and of nearly eliminating the pause needed to start or stop a collection. Parallel collectors need only execute a barrier synchronization to change "modes", which incurs a negligible pause.

The collector presented in this chapter is a mostly lock–free parallel collector. It is compatible both with the generational and non–generational approach, and can trace through data structures which are allocated using other methods as well (known as "external" objects). The collector maintains a single "collector" thread for each executor. Unlike normal threads, collectors are bound to their executors and cannot be moved between them. Normal program threads are called "mutators", and may execute alongside any number of collectors. This collector is designed to handle large arrays in parallel, allowing multiple collectors to work on them at once. The visible pause in execution is no more than a barrier synchronization, which is necessary for switching the "mode" of the program and having all executors acknowledge the change in mode. The collector provides a straightforward implementation of weak pointers, which are pointers which are not followed by the collector. Lastly, the collector is entirely lock–free with the exception of the mode–change barriers.

This collector requires several things from the runtime and compiler. First, the compiler must generate type signatures for all datatypes, and must ensure that all garbage–collected objects have a properly–formatted header. Type signatures provide a tractable means by which to trace the memory graph, with none of the difficulties that arise from tags[143]. Type signatures are generated and used by several real–world compilers, including the MLton compiler[105]. Second, the compiler must implement two execution modes, which may be changed at any safe–point. In the first mode, execution is normal. In the second mode, the compiler must log all updates to mutable objects to a special write–log. Lastly, the compiler must implement pairs of pointers, deciding which of the pair to use depending on a bit in the execution mode. Global pointers must be implemented in this way for non–generational collection; for generational collection, *all* pointers require this treatment.

The root set for a collection is calculated by starting from all active threads and all global pointers. To facilitate this, the compiler must generate a structure which lists all global pointer values and their types. The active threads can be derived by traversing scheduler structures. When a collection is active, the scheduler itself must keep track of the changes to the status and the resume continuations of threads which have already been visited by the collector.

A collector thread "claims" a given object by setting the forwarding pointer in its header using a compare–and–set operation. For objects which are being copied, the new value is the address of the object's replica in the destination space. Claimed objects are inserted into a local queue for processing. The queue is a FIFO queue, and all "neighbors" of a given object are claimed at once, before attempting to process another object. This attempts to cluster objects so as to improve their locality. The tactic of clustering objects is known to be beneficial[139], though the lock–free and concurrent nature of this algorithm makes it difficult to do perfectly.

Collectors periodically push objects from their queues into a shared lock–free structure, or pull work from this structure into their queues, depending on whether they have too much or too little work. Additionally, arrays of sufficient size are divided amongst all collectors using an atomically–modified bitmap structure. Arrays which match the size criteria are pushed into a lock–free stack or queue as they are discovered, and are processed concurrently by all collectors.

While a collection is running, mutators must keep a write–log. When this log becomes full, the executor stops executing the mutator and begins its collector thread. Upon entry to the collector, the write–log for the executor is processed, then normal processing begins.

When all work is exhausted, a collector executes a barrier and signals all executors to switch to their collectors. This will result in the processing of all write–logs, which should not incur a significant overhead, as there can be at most one new object per entry. Once this is done, the mode is switched, and normal execution resumes.

8.2 Garbage Collection Algorithms

This section presents the core algorithms of the garbage collector. The garbage collector's organization is markedly different from that of the scheduler, in that it is more a "protocol" than an singular algorithm. The garbage collector requires the cooperation of the compiler, the program, and the garbage collection library itself. As such, there are many distinct procedures which lack the central organization present in the scheduler.

The program–side interface to the garbage collector is too simple to warrant discussion. Each executor maintains an `allocator` structure, from which it allocates memory. In a non–generational collector, this consists of nothing more than a pointer and a size. Since the allocator is a per–executor structure, memory is allocated simply by advancing the pointer. When an allocator runs out of memory, the executor requests

```
type allocator

void* prealloc(gen : uint, size : uint)
void* postalloc(gen : uint, size : uint)
void* allocate(gen : uint, size : uint) =
  out = prealloc(gen, size)
  postalloc(gen, size
  out
```

Figure 8.1: The Garbage Collection and Thread-Local Allocator Interface

another block from the garbage collector. At this point, a voluntary collection may begin if available memory passes a threshold. A forced collection may begin if memory begins to approach dangerously low levels. If all free memory is exhausted (except for the free memory necessary to guarantee that the collection finishes), then all requests will immediately transfer control to the garbage collector thread for that executor. Additionally, there is a function by which a program can voluntarily start a collection cycle.

Figure 8.1 shows the interface for an allocator. The `prealloc` and `postalloc` functions are used later in the process of "claiming" an object. All functions take a `gen` argument, indicating the generation from which to allocate, and a `size` argument. The `prealloc` function ensures that sufficient space exists, acquiring more free space if it is not. The `postalloc` function actually completes the allocation, advancing the executor's pointer. The `allocate` function demonstrates the equivalence to subsequent `prealloc` and `postalloc` calls.

Since there is a constant number of generations, an allocator can be implemented simply as an array of pointer pairs. During execution, a program need only increment the first pointer, and check if it exceeds the second, invoking the garbage collector to allocate another memory pool if it does.

Upon encountering a block, a collector attempts to "claim" it by setting its forwarding pointer using a compare–and–set. Figure 8.2 shows the `claim` function. If the object qualifies for collection (as determined by the `collected` function), and has not already been `claimed`, the function attempts to allocate space for the object, then set the forwarding pointer. In all cases, the function returns the address of the destination object. If the claim succeeds, the collector finalizes allocation, and adds the object to its queues, or to the array queue if appropriate, and returns the address in the destination space. If the object has already been claimed, or the compare–and–set operation failed, the function returns the forwarding pointer value. If the object is not being collected at all, the function attempts to set the forwarding pointer to the value `claimed`, and returns the object's address in all cases.

The `generation` and `share_array` functions are not shown. The first decides which generation from which to allocate, while the second decides if an array is large enough to be shared, or if it should be processed as a normal object.

Another important detail is the manner in which the destination object's header is initialized. The `fwd_ptr` is set to `claimed`, and the bitmap (if it exists) is also initialized such that all clusters are claimed. This is important, since at the end of collection, the `claimed` and `unclaimed` values are switched, making the destination object become unclaimed. This also causes all objects which were not collected to revert to an unclaimed status.

```
void add_obj(thread : gc_thread*, obj : gc_header*) =
  if(type.objclass == NORMAL || !share_array(obj))
    enqueue(thread.queue, obj)
  else
    lf_enqueue(arrays, obj)

gc_header* claim(thread : gc_thread*, obj : gc_header*) =
  type = obj.typeptr
  fptr = obj.fwd_ptr
  gen = generation(obj)
  if(collected(gen))
    if(unclaimed == fptr)
      newobj = prealloc(gen, size(type))
      if(CAS(unclaimed, newobj, obj.fwd_ptr))
        out = newobj
        postalloc(gen, size(type))
        // initialize header
        add_obj(thread, obj)
      else
        out = obj.fwd_ptr
    else
      out = fptr
  else
    out = obj
    if(CAS(unclaimed, claimed, obj.fwd_ptr))
      add_obj(thread, obj)
  out
```

Figure 8.2: Claiming Objects

After claiming objects, a thread proceeds to process objects in its queue, which involves actually copying the data in the objects to the destination space, and likely involves further `claim` attempts. Figure 8.4 shows the `process` function, and figure 8.3 shows the functions on which it depends. The `copy` function copies all non–pointer data, and attempts to `claim` all pointers to get their proper values in the destination space. Note that all pointers are double–pointers, and the function accesses the `used` value in the source space, and writes the `usused` value in the destination space.

The processing of constant objects and those which are mutable differs. In the case of constant objects, a `copy` is enough to guarantee consistency. In the case of mutable objects, however, a write–after–write hazard may occur. It is important to note that since objects in the destination space are not actually used until the end of collection (at which point a barrier will be executed), it is not necessary to guarantee a *consistent* view of the data, merely that all write–after–write hazards will be dealt with before the barrier is executed. This is done by the `check` function, which scans both objects, checking for equivalence of all fields. If the objects match, then there was no possible write–after–write. Otherwise, the function copies from the source to the destination until both are observed to be identical. A more rigorous proof of this function is presented in section 8.3.

In the case where an object is not being collected, the `update` function claims all of the `used` pointers, storing the resulting addresses to the `unused` side. The `update_check` function is analogous to the `check`,

```
void copy(thread : thread*, dst : void*, src : void*,
          nonptr_size : uint, normptrs : uint, weakptrs : uint) =
  memcpy(dst, src, nonptr_size)
  dstptrs = dst + nonptr_size + sizeof(gc_header)
  srcptrs = src + nonptr_size + sizeof(gc_header)
  for(i = 0; i < normptrs; i++)
    dstptrs[i][unused] = claim(thread, srcptrs[i][used])
  if(keep_weak)
    for(i = normptrs; i < weakptrs + normptrs; i++)
    dstptrs[i][unused] = claim(thread, srcptrs[i][used])

bool check_ptr(dst : gc_header*[2], src : gc_header*[2]) =
  do
    succeed = true
    if(srcptrs[i][used] != NULL)
      if(srcptrs[i][used].fwd_ptr != claimed)
        if(dstptrs[i][unused] != srcptrs[i][used].fwd_ptr)
          succeed = false
          dstptrs[i][unused] = srcptrs[i][used].fwd_ptr
      else
        if(dstptrs[i][unused] != srcptrs[i][used])
          succeed = false
          dstptrs[i][unused] = srcptrs[i][used]
    else if(dstptrs[i][unused] != NULL)
      succeed = false
      dstptrs[i] = NULL
  while(!succeed)

void check(dst : void*, src : void*, nonptr_size : uint,
           normptrs : uint, weakptrs : uint) =
  for(i = 0; i < nonptr_size; i++)
    while(dst[i] != src[i])
      dst[i] = src[i]
  dstptrs = dst + nonptr_size + sizeof(gc_header)
  srcptrs = src + nonptr_size + sizeof(gc_header)
  for(i = 0; i < normptrs; i++)
    check_ptr(dstptrs + i, srcptrs + i)
  for(i = 0; i < normptrs + weakptrs; i++)
    if(NULL != dstptrs[i][unused])
      check_ptr(dstptrs + i, srcptrs + i)
  while(!succeed)
```

Figure 8.3: The copy and check Functions

```
void update(thread : thread*, obj : gc_header*, normptrs : uint,
            weakptrs : uint) =
  dstptrs = dst + nonptr_size + sizeof(gc_header)
  srcptrs = src + nonptr_size + sizeof(gc_header)
  for(i = 0; i < normptrs; i++)
    srcptrs[i][unused] = claim(thread, srcptrs[i][used])
  if(keep_weak)
    for(i = normptrs; i < weakptrs + normptrs; i++)
    srcptrs[i][unused] = claim(thread, srcptrs[i][used])

void update_check(obj : gc_header*, nonptr_size : uint,
                  normptrs : uint, weakptrs : uint) =
  do
    succeed = true
    dstptrs = dst + nonptr_size + sizeof(gc_header)
    srcptrs = src + nonptr_size + sizeof(gc_header)
    for(i = 0; i < normptrs; i++)
      succeed &= check_ptr(srcptrs + i, srcptrs + i)
    for(i = 0; i < normptrs + weakptrs; i++)
      if(NULL != dstptrs[i][unused])
        succeed &= check_ptr(srcptrs + i, srcptrs + i)
  while(!succeed)

void process(thread : thread*, obj : void*) =
  type = obj.typeptr
  nonptr_size = type.nonptr_size
  normptrs = type.normptrs
  weakptrs = type.weakptrs
  constant = type.constant
  if(collected(obj))
    copy(thread, dst, src, nonptr_size, normptrs, weakptrs)
    if(!constant)
      check(dst, src, nonptr_size, normptrs, weakptrs)
  else
    update(thread, obj, normptrs, weakptrs)
    if(!constant)
      update_check(thread, obj, normptrs, weakptrs)
```

Figure 8.4: The process Function

except that it only checks and propagates modifications involving the pointers.

The exact algorithms for claiming and processing array element clusters are not shown here, primarily because of their complicated nature and their similarity to the algorithms for processing normal objects. Arrays of sufficient size are preceded by an atomically–modified bitmap. As with the `claimed` and `unclaimed` values, and the `used` and `unused` indexes of double–pointers, the bitmaps' representation alternates between set and clear bits indicating available clusters each collection cycle.

In the case of mutable objects, it is necessary to perform additional actions when an object is updated after it has been copied to the destination space. As with the Cheng–Blelloch collector, this collector maintains a per–executor write log which records each write. When the log is full (or if the executor begins executing its collector for some other reason), the executor switches to its collector thread to process the log. The write log must also be updated each time a context–switch occurs, as the resume continuation for the old thread is altered, likely changing the current frame and the pointers it contains.

Figure 8.5 demonstrates the functions that process a write–log. Log entries include a pointer to the the object and the offset at which a modification occurred. For each entry, the value in the source object is repeatedly copied to the destination object until the two fields are observed to be identical. This repeated copy is necessary to avoid a write–after–write hazard which can occur if a simple copy is performed. If the offset refers to a pointer, the collector uses `claim` to acquire the new address. Since write–log entries only record that a write occurred, there is no need to process multiple entries for the same location. If the object is in a generation which is not being collected, the `unused` side of a pointer needs to be updated, but non–pointer data does not. The top–level `process_log` function uses the `already_processed` function to ensure this[1].

Unlike the Cheng–Blelloch collector, there is no need to keep track of the old value of the field. Additionally, the initialization of an object does not cause write log entries to be generated, so long as these writes take place before the object becomes reachable. The double–allocation phase which is necessary in the Cheng–Blelloch collector is also not necessary, as all new, reachable objects will be saved to the thread's frame, or to some existing object, thereby generating write–log entries prior to switching to the collector. This does imply, however, that the compiler must ensure that there are enough free write–log slots available to store all updates to the current frame, as well as one to store the update to the resume continuation for the thread.

The actual work of garbage–collection is performed by a number of garbage–collection threads. Garbage–collection threads keep their state in a `gc_thread` structure. Each switch into a garbage–collection thread represents an attempt at completing collection. The thread will execute a function which executes the thread's write log, then attempts to process the entire remaining collection of objects. Garbage collector threads are designed such that they do not need to save a resume continuation. If a collector is interrupted, it can safely resume execution at the beginning of the `gc_thread` function, rather than saving its current state. The `final_barrier` can also be implemented in this fashion. Once one collector reaches the `final_barrier`, all attempts to allocate memory will cause the executor to switch into its collector. The barrier itself blocks the underlying executor[2], until the last one passes the barrier, in which case it performs the switch out of

[1]This can be easily implemented with a hash table

[2]The only other instance in which the runtime blocks an executor is when it runs out of threads to execute.

```
void process_ptr(thread : gc_thread*, src : gc_header**, dst : gc_header **) =
  do
    cont = false
    srcptr = *src
    dstptr = *dst
    if(NULL != srcptr)
      if(dstptr != srcptr.fwd_ptr)
        *dstptr = claim(thread, srcptr)
        cont = true
    else if(NULL != dstptr)
      *dstptr = NULL
      cont = true
  while(cont)

void process_entry(thread : gc_thread*, entry : log_ent) =
  src = entry.obj
  offset = entry.offset
  type = src.typeptr
  if(unclaimed != dst)
    realoffset = offset + sizeof(gc_header)
    if(collected(src))
      dst = src.fwd_ptr
      if(offset < type.nonptr_size)
        while(dst[realoffset] != src[realoffset])
          dst[realoffset] = src[realoffset]
      else
          check_ptr(thread, src + realoffset, dst + realoffset)
    else
      if(offset > type.nonptr_size)
        check_ptr(thread, src + realoffset, src + realoffset)

void process_log(thread : gc_thread*) =
  log = thread.log
  for(i = 0; i < LOG_SIZE; i++)
    if(!already_processed(log[i])
      process_entry(thread, log[i])
```

Figure 8.5: Executing Write Logs

garbage–collection mode and releases all other executors, which immediately invoke their schedulers.

The `initial_barrier` sets a global variable indicating that garbage–collection mode is active. The `final_barrier` sets the same variable, indicating that garbage–collection mode is inactive, swaps the `used` and `unused` indexes for double–pointers, `claimed` and `unclaimed` for forwarding pointers, and reverses the interpretation of array bitmaps, and frees the entire source space.

There are several requirements of the program threads. First, double pointers must be dealt with properly, which requires the program to know the value of `used` at all times. Additionally, the write–barrier requires the program to know whether a collection is underway or not. Finally, the behavior of memory allocation and safe–points changes when an executor has passed the `final_barrier`. In practice, all these values can be kept in a single word, which is updated atomically. While pointer–dereferencing and writes do require a

```
void gc_thread(thread : gc_thread) =
  process_log(thread)
  while(claim_array_cluster() || claim_root())
    while(NULL != (obj = dequeue(thread.queue)))
      for(i = 0; i < work_size; i++)
        process(thread, obj)
      balance(thread);
      safepoint_without_context_save()
  final_barrier()

void start_collection() =
  initial_barrier()
```

Figure 8.6: Garbage Collection Top-Level Functions

conditional block, aggressive compiler optimizations and speculative architectures should be able to reduce the cost of there conditionals significantly. Lastly, when the program allocates and initializes an object, it must properly initialize its `gc_header` (and bitmap if it is an array of sufficient size). The header's `fwd_ptr` must always be initialized to `unclaimed`, and its bitmap must also be initialized such that all clusters are unclaimed.

Now that the algorithm has been presented, section 8.3 proves several correctness conditions.

8.3 Correctness

As I previously discussed, I do not develop a full model of garbage collection as I did for the scheduler. The primary purpose of these proofs is to establish the well–behavedness of the lock–free algorithms, rather than to exhaustively proof the garbage collector's preservation of liveness and typing, the correctness of the root set, *et cetera*. For this purpose, I define simple correctness conditions and present proofs that the collector preserves them.

First, it is necessary to define the meaning of reachability and liveness:

Definition 8.3.1 (Reachability)**.** An object o is *reachable* with respect to another object o' if and only if o' contains a normal pointer to o.

An object o is reachable with respect to set of objects O if and only if o is reachable from some element of O.

Definition 8.3.2 (Root Set)**.** The *root set* R is a set of objects which are always reachable, due to the properties of the program being executed.

Definition 8.3.3 (Live Set)**.** The *live set* L is the largest possible superset of R such that all objects in L are either in R or are reachable from some other object in L. If an object is in L, it is said to be *live*.

Definition 8.3.4 (Visitation)**.** An object is *visit*ed in a collection cycle if both `claim` and `process` are called on it.

Definition 8.3.5 (End of Collection Cycles)**.** A collection cycle ends when the last collector executes the `final_barrier`

Definition 8.3.6 (Object Equivalence)**.** An object o is equivalent to o' if and only if the following conditions are met:

i o and o' have the same type.

ii All non–pointer fields of o and o' are equal

iii For each normal pointer field in o and o', both fields point to equivalent objects.

iiii For each weak pointer field in o and o', both fields point to equivalent objects, or else the destination object's pointer is null.

Definition 8.3.7 (Correctness Conditions)**.** The following conditions define correctness of the garbage collector:

(i) There is a bijective mapping between reachable objects in the source space and objects in the destination space.

(ii) When the final barrier is passed at the end of collection, all objects in the destination space are equivalent to their corresponding objects in the active space.

(iii) The `final_barrier` is the linearization point for ending collection, adopting the destination space as the active space, and adding the old active space to the pool of free memory.

Condition (i) implies that all reachable objects are copied, and that a given object has only one duplicate in the destination space. Condition (ii) implies that the destination space represents a "snapshot" of the source space at the time when the barrier is passed (at which time the source space becomes the active space). Lastly, condition (iii) establishes the point at which collection ends and the new active space is adopted.

An important precondition for proving condition (i) is that the reachable objects are in an unclaimed state at the beginning of every collection. The following lemma sets up for a mutually inductive proof that a collection cycle finds all reachable objects.

Lemma 8.3.8. *Assuming a collection cycle visits all live objects, all live objects at the beginning of the next collection cycle are unclaimed.*

Proof. Proof is by induction.

Leading up to the initial collection, all objects are initialized as unclaimed.

In any given collection cycle, all object's `fwd_ptr`s are either set to `claimed` if the object is not collected in this cycle, or to a pointer to an object in the destination space if it is. The destination object is initialized with its `fwd_ptr` set to `claimed`. Additionally, all arrays' bitmaps are either completely claimed if the array is not collected. If the array is collected, then its replica in the destination space has its bitmap initialized to be completely claimed. The only way an object could be unclaimed at the end of a collection cycle is if the collection cycle never visited it. By the base assumption of the lemma, the collection cycle visits all reachable objects, so only unreachable objects are unclaimed at the end of a collection.

When the `final_barrier` is passed, the values for `claimed` and `unclaimed`, and the interpretation of bitmaps are reversed, meaning all objects in the destination space, or all objects which were reachable but were not collected in this cycle become unclaimed. As previously demonstrated, all objects are initialized to an unclaimed state during normal execution. □

Now, it is necessary to prove that an object is visited if it is `claim`ed.

Lemma 8.3.9. *If* `claim` *is called on an object,* `process` *is called on it before the end of the collection cycle, assuming all objects are unclaimed at the start of the collection cycle.*

Proof. Observe that if `claim` is called on an object at least once, the `CAS` operation must succeed for one of those calls, since it tests for the value `unclaimed`. When the `CAS` succeeds on an object , it is either inserted into `shared_arrays`, or into the collector's local queue. Additionally, observe that a collector only executes the `final_barrier` when the list of shared arrays and its local queue are empty. Finally, observe that whenever an object is removed from the local queue, it is `process`ed, and an array is only removed from `shared_arrays` when all of its clusters have been claimed and processed. □

The second half of the mutual induction involves proving that a collection visits all reachable objects.

Lemma 8.3.10. *Assuming all live objects are unclaimed at the beginning of the beginning of a collection cycle, the cycle visits all live objects.*

Proof. Proof is by induction.

With an empty root set, the lemma is vacuously true.

When the live set exactly equals the root set, observe that a collector only executes `final_barrier` after all objects in the root set are `claim`ed.

Now, add an new object o to live set L, but not to the root set. By definition 8.3.3, there is some pointer to it in an object in L. Observe that all pointers in any given object are `claim`ed when the object is `process`ed. Lastly, observe that whenever a write takes place to a pointer field, the new pointer value is also `claim`ed. Hence, since all objects in L must be `claim`ed and all `claim`ed objects are `process`ed by lemma 8.3.9, and all pointers of an object which is `process`ed are `claim`ed, and there must be some pointer to o in some object in L, then o must be `claim`ed and `process`ed. □

With these lemmas proven, it is now possible to prove that all collections visit all live objects.

Lemma 8.3.11. *All collection cycles visit all live objects.*

Proof. Proof is by mutual induction using lemmas 8.3.8 and 8.3.10.

Prior to the first induction, all objects are unclaimed, as demonstrated in the proof of lemma 8.3.8, hence, by lemma 8.3.10, the first collection visits all objects.

Given that all previous collection cycle visited all live objects, then all objects will be unclaimed at the beginning of the next collection cycle, by lemma 8.3.8, which in turn implies that that collection cycle will visit all live objects, by lemma 8.3.10. □

Corollary 8.3.12. *If an object is* `claim`*ed in a collection cycle, it is visited by that collection cycle.*

Proof. Immediate consequence of lemmas 8.3.9, 8.3.11, and 8.3.8. □

Corollary 8.3.13. *For each object in the source space, there is at least one object in the destination space corresponding to it at the end of a collection cycle.*

Proof. Observe that when an object is `claimed`, a copy of it is created in the destination space. By lemma 8.3.11, all objects are visited in every collection cycle. □

This proves one half of condition (i). Now it is necessary to prove that there is at most one copy of any object in the destination space.

Lemma 8.3.14. *An object's replica is allocated in the destination space, its* `fwd_ptr` *is set, and it is* `process`*ed at most once in a given collection cycle.*

Proof. Observe that an object is processed only if it is in a local queue, or it is in `shared_arrays`. An object is inserted into the local queues or `shared_arrays`, and `postalloc` is called to complete an allocation if and only if the `CAS` operation in `claim` succeeds. The `CAS` tests for `fwd_ptr` to be `unclaimed` and sets it so something other than `unclaimed`, and `fwd_ptr` is not modified anywhere else, so the operation only succeeds at most once per collection cycle. □

Corollary 8.3.15. *For each object in the source space, there is at most one object in the destination space corresponding to it at the end of a collection cycle.*

Proof. Immediate consequence of lemma 8.3.14 □

Condition (i) is now trivial:

Theorem 8.3.16 (Bijective Copy)**.** *There is a bijective mapping between reachable objects in the source space and objects in the destination space.*

Proof. Immediate consequence of corollaries 8.3.13 and 8.3.15. □

Some definitions and lemmas are necessary to set up the proof of condition (ii).

Definition 8.3.17. All writes are logged to the executor's write–log while a collection cycle is taking place.

Definition 8.3.18. If an object is not copied in a collection cycle, the destination space for it is defined as the same object, but considering the `unused` pointer values rather than the `used` values.

Lemma 8.3.19. *An executor's write–log is completely executed prior to execution of* `final_barrier`.

Proof. Straightforward from the definition of `gc_thread`. □

Lemma 8.3.20. *Write–log execution guarantees that if a field in an object is modified in the source space, that field will be atomically copied to the destination space at some point before the end of the collection cycle. If the field is a pointer, the destination will be atomically set to the return value of* `claim` *on the source.*

Proof. The `process_entry` function repeatedly performs a copy if it observes that the source and destination fields differ. If the fields are not observably different, then either a) no write to the source occurred, or b) a write, followed by an atomic copy occurred. Either case preserves the property.

In the case of pointers, if the object is not being collected, then `claim` will return the source pointer, and the equivalence check is a simple equality. If the object is being collected, then `claim` will set the `fwd_ptr` if it is not already set, and return its value regardless. By lemma 8.3.14, `fwd_ptr` is set at most once, so all subsequent `claim`s will return its value. Therefore, the equivalence test can safely check the destination against the source's `fwd_ptr`.

Lemma 8.3.19 guarantees all write log entries will be executed. The `process_log` function executes only one atomic copy for any number of log entries. However, all executions of `process_entry` after the first will observe the two objects' fields to be identical, unless another write occurs by some other executor. However, if another write occurs, then that executor will generate a write log entry and execute it at some point in the future. □

Lemma 8.3.21. *The* `process` *function maintains the property that if a field in an object is modified in the source space, that field will be atomic–copied to the destination space at some point before the end of the collection cycle. If the field is a pointer, the destination will be atomically set to the return value of* `claim` *on the source.*

Proof. If the object is constant, then no writes occur and the lemma is trivially correct.

When the object is mutable, `process` performs the same atomic copy performed by a write–log execution to all fields. By lemma 8.3.20, the property is preserved. □

Lemma 8.3.22. *Assuming that the* `claim` *function returns a pointer to an object that will be equivalent to its argument at some point in the future, if all writes to fields in a source object are followed by atomic copies to the destination at some point in the future, then the source and destination object will be equivalent at some point before the end of the collection cycle.*

Proof. When the last thread enters the `gc_thread` function in which it will execute `final_barrier`, no more writes will occur. Lemma 8.3.20 implies there is some point in the future when all fields which have been written will be atomic–copied to the destination, and all destination pointers will be atomically set to the value of `claim` on the source. Since the destination object is created using the same type signature as the source, both objects have the same type. Therefore, by the assumptions of the lemma, both objects will be equivalent. □

Lemma 8.3.23. *The* `claim` *function returns a pointer to an object that will be equivalent to its argument at some point before the end of the collection cycle.*

Proof. Proof is by induction.

Observe that for a `claim` on an object with no pointer data, lemma 8.3.22 without assuming this lemma. By lemma 8.3.9, `process` will be called on the object at some point in the future after being `claim`ed. Since lemma 8.3.22 can be used in this case, it follows that the result of `claim` will be equivalent to its argument at some point in the future.

Observe that pointers which are `NULL` are not valid arguments to `claim`, and `process` copies the `NULL` values to the corresponding object.

Now assume that there are two objects, both with one pointer, which are mutually recursive. Observe that for both, `claim` returns an object in the destination space, whose non–pointer data is equivalent by previous assumptions. Observe that the destination objects are also mutually–recursive, which satisfies definition of equivalence.

Now assume that the lemma is true for all pointers in some object. By lemma 8.3.9 the object will be `processed` at some point in the future, and since the lemma holds for all pointers in the object, lemma 8.3.22 can be safely used to guarantee that this object will be consistent at some point in the future. □

Now it is possible to prove condition (ii).

Theorem 8.3.24 (Source–Destination Equivalence)**.** *When the* `final_barrier` *is passed at the end of collection, all objects in the destination space are equivalent to their corresponding objects in the active space.*

Proof. Lemmas 8.3.11 and 8.3.14 imply that each live object will be `claim`ed exactly once. Lemma 8.3.23 implies that `claim` returns a pointer to an object which is equivalent to its argument at some point before the end of the collection cycle. Since all pointers in objects which are not copied have their `unused` index set to the return value of `claim` on their `used` side, and all objects which are copied have their `fwd_ptr` set to the result of `claim`, then the theorem holds. □

The proof of condition (iii) is trivial, given the existing theorems.

Theorem 8.3.25 (Linearization)**.** *The* `final_barrier` *is the linearization point for ending collection, adopting the destination space as the active space, and adding the old active space to the pool of free memory.*

Proof. Definition 8.3.5 establishes that `final_barrier` ends a cycle. By theorem 8.3.24 and definition 8.3.18, all non–copied objects (which includes the root set) contain pointers to equivalent objects in the destination space in the `unused` index of their pointers. Since `final_barrier` swaps `used` and `unused`, the second condition is met. Finally, the third condition follows directly from the definition of `final_barrier`. □

This completes the proofs of the garbage collector, which in turn completes the runtime system itself. This part has stated the design rationale for a runtime system for concurrent functional programming, given the details of its implementation, and proven the correctness of that implementation. This lays the foundation for using concurrent functional programming to exploit fine–grained parallelism. Part III will describe techniques in concurrent functional programming which further expose parallelism.

Part III

Reducing False Dependence in Concurrent Functional Programming

Chapter 9

Overview

Concurrent functional programming is able to express fine–grained concurrency by separating a program's linear control flow into a tree–like structure of control flows with non–interfering branches. The success of this strategy depends both on the ability to create and maintain a large number of short–lived threads, and on the ability to shorten the critical path as much as possible. Part II dealt with implementing a runtime system which makes creating large numbers of short–lived threads feasible. As the overhead of thread creation and maintenance drops, the shortest tolerable thread lifespan and the maximum tolerable number of threads increases.

This part of the thesis concerns itself with the second of the two concerns: shortening the critical path. Chapter 3 discussed functional programming, higher–order concurrency[132], and the critical concepts of concurrent functional programming. Even with a runtime which provides infinite parallelism at no cost, a program's minimum execution time is still bound by the longest chain of data dependencies. Data dependence is the prime enemy of parallelism; no chain of dependent operations can be parallelized[1]

Imperative programming is inherently sequential; each operation is dependent upon the state of the world at the end of the last one. In effect, the entire program becomes one long dependency chain. Automatically locating branches in this chain is quite difficult and has resisted considerable research efforts. Functional programming, on the other hand is inherently parallel as there is a direct correspondence between function calls and opportunities to create threads. However, the critical path can be further shortened by eliminating false data–dependence between functions which produce a structure and the rest of the program which consumes it.

Futures are the primary means of inter–thread communication in concurrent functional programming. Futures permit a "placeholder" for a value to be substituted, while a separate thread calculates the actual value. This enables execution to immediately progress the thread which calls a function, while the actual return value is calculated in another thread. Hopefully, by the time the future's value is requested, it has already been calculated by the other thread.

The näive use of futures for complex data structures leads to a large amount of artificial dependency. In a case where a first function produces a structure, which is consumed by a second function to produce a

[1]For an example of a pathological program, consider the program which calculates the first n results of a pseudorandom generator function which feeds the previous result into the next.

second structure, any element of the second structure is dependent upon the entire first structure. The *real* dependencies may be much less, however. False dependencies in effect prevent functions which "consume" a complex data structure to begin executing before the function which "produces" them completes.

The ability to process multiple stages of execution concurrently, of course, benefits multi–stage computations by shortening the critical path and increasing the number of threads which can be live at a given time. This is analogous to pipelining, which is a cornerstone of modern computer architecture[57]. For this strategy to succeed, complex structures must be implemented using futures to break the structure into pieces which can be passed down the pipeline when they are completed. While this approach is effective for basic data structures, more complex structures must be specifically designed to support this style of programming.

The chapters in this part of the thesis discuss how to effectively implement futures for various kinds of complex structures. Chapter 10 covers the basics of futures, as well as how to effectively represent futures of tuple and record types, recursive types, and lists[2]. As far as I am aware, this chapter introduces concepts which are either generally known or fairly intuitive, yet unpublished. I present them here for the sake of completeness, and in order to serve as a starting point for the following chapters. Chapter 11 discusses the challenges with futures for associative data structures. It presents an effective implementation of such futures, and discusses various methods for further improving concurrency when one such structure is used to create another. Chapter 12 discusses futures for the results of a stateful computation. This chapter's methods are geared towards encapsulating a modify–in–place pass over some data structure, and allowing subsequent phases to continue before it completes.

Program code examples in this part are written in Standard ML[103], unlike those in part II. I choose this language for several reasons. First, the Concurrent ML[130] language, upon which much of the work in this section is initially based is derived from Standard ML. Additionally, the Standard ML language provides exactly the sort of language needed to express the ideas of this section: a language built for functional programming, but which supplies imperative constructs. Imperative techniques are necessary for the implementation of the futures described by chapter 11, and the stateful computations described by chapter 12 obviously depend on state.

[2]It is a somewhat common practice to represent sets as lists. Futures on sets are a completely different issue.

Chapter 10

Direct Structures

Concurrent functional programming exploits the implicit parallelism in functional programs by expressing separable control–flows. Calls to sufficiently large functions are performed asynchronously, returning a placeholder value called a future. At some point, (hopefully after the function has completed), the future is evaluated to produce a concrete value. If the value of the future is already available, it is simply used; otherwise, the evaluator must wait for the value to be produced. This style not only extracts the natural concurrency of functional languages, but also allows partial overlapping of dependency among computations.

Futures work well with functions that produce a single scalar value. However, if a function produces more complicated data, their performance begins to suffer. For instance, in the case of a tuple or a record type[1]. A future of a record must be evaluated to get at any one of its fields, which forces the evaluator to wait for *all* fields, not just the one it wants. The more complicated the structure, the worse the performance detriment.

All chapters in this part of the thesis describe methods for overcoming this impediment. The methods described by these chapters create structures which can be operated on while they are semi–complete. This chapter deals with the most basic case: structures whose meaning matches their implementation, which I term *direct* structures. Section 10.1 begins by introducing methods for tuples, vectors[2], and recursive structures. Section 10.2 describes methods suitable for lists. The section also discusses the real meaning of lists, and when they are actually necessary, rather than merely convenient. This is an important consideration, and is further explored in chapter 11. For the most part, the concepts described in this chapter are not meant to represent innovative concepts, but are included for the sake of introduction and completeness.

An important note to make is that none of the structures described in this chapter, or in chapter 11 are allowed to be mutable. Both of these chapters are oriented toward a situation wherein a *producer* function creates a structure, which is then processed by a *consumer*. The producer can modify the structure, while the consumer can only read it. Additionally, once a producer creates part of the structure, that action is considered final. Chapter 12 discusses more general cases, where a structure can be modified in–place.

Types:

τ	::=	τ `future`	(future)
		τ `[]`	(vector)
		$\tau \times \tau$	(product)
		$\tau + \tau$	(sum)
		$\mu X.\tau$	(recursive)
		$\forall X.\tau$	(universal)
		X	(type – variable)

Future Translation:

$[\![\tau$ `future future`$]\!]$	=	$[\![\tau$ `future` $]\!]$
$[\![\tau$ `[] future` $]\!]$	=	$[\![\tau$ `future` $]\!]$ `[]`
$[\![(\tau \times \tau)$ `future` $]\!]$	=	$[\![\tau$ `future` $]\!] \times [\![\tau$ `future` $]\!]$
$[\![(\tau + \tau)$ `future` $]\!]$	=	$([\![\tau$ `future` $]\!] + [\![\tau$ `future` $]\!])$ `future`
$[\![(\mu X.\tau)$ `future` $]\!]$	=	$\mu X.[\![\tau$ `future` $]\!]$
$[\![(\forall X.\tau)$ `future` $]\!]$	=	$\forall X.[\![\tau$ `future` $]\!]$
$[\![X$ `future` $]\!]$	=	X `future`

Figure 10.1: A Type–Translation Function for Futures for Basic Types

10.1 Straightforward Futures

For some structures, there exists a relatively straightforward implementation of futures. This is the case for compound datatypes such as tuples and records. Similar methods also work for vectors. Indeed, this section can be thought of as describing a translation from a single future on some data-type to a structure which can be produced and consumed concurrently, or *pipelined.* Lazily–evaluated languages also provide a straightforward method for using futures in a structure or computation. Lazy evaluation makes use of *thunks*, which are nothing more than futures without any notion of concurrency. The exact relationship between lazy evaluation and concurrency will be explored later.

Figure 10.1 shows a type translation function, which translates a future on some compound type into a more effective future type by "plunging" the futures deeper into the data type. The gist of this function is that a future on a compound data type is implemented by proceeding down the "tree" of types to the leaves, and implementing these leaves as futures. It is important to note that despite the formality of the presentation of this function, it does *not* represent a transformation which can be blindly applied to a program's types without also modifying the program itself. Clearly, plunging futures into a datatype requires modification to accesses to that type, as some values are changed from concrete values to futures and vice versa.

The first rule addresses double–futures, which serve no beneficial purpose, and are implemented instead as a single future. In the case of both vector types and product types (and by derivation, tuples and records), the futures are distributed to the inner types, and the transformation continues. Both recursive and universal quantification also propagate futures inward and continue the transformation. On any bound type variable, the transformation halts.

[1]In C or Java parlance, a struct or an object

[2]The meaning of "vector" is taken in the context of ML, Haskell and other languages: a non–modifiable array.

Sum (and by derivation, variant types) deserve special treatment. As the inner types of a variant may be arbitrarily complex, it is necessary to distribute futures and recur. However, unlike products types, which are mere containers, sum types do carry information in the form of which side of the sum has actual meaning (or in terms of variants, which variant the value is). In the worst case, a program spends a considerable amount of time deciding *which* variant to generate, and then spends a considerable amount of time actually generating the structure. In this case, *both* the variant *and* its data must be implemented as futures.

Obviously, some judgement must go into the application of these rules. Sum types, as discussed above give rise to one pathological case which requires both the variant itself and its inner types to be futures. However, in the case of a simple enumeration, there is no need to implement the inner types as futures. Likewise, care must be taken when distributing futures as with product and vector types. It makes no sense, for instance to translate a simple two–dimensional point implemented using products into a pair of futures, as both values are likely to be produced at the same time. In short these transformations are meant to be used where appropriate, with careful judgement, not blindly applied.

10.2 List Futures

Section 10.1 discussed futures for basic compound datatypes. Among these types were recursive types. At a glance, this treatment applies to lists, as lists are a special case of recursive types. However, the manner in which lists are used, and the exact data they represent benefits most from a specific treatment.

Lists represent an ordered sequence of items. In functional languages, lists are very often constructed in reverse–order, then either reversed, or evaluated in such a manner as to take this into account. This is because elements can be added to the front of a purely–functional list very cheaply, while adding them to the end requires traversing the entire list.

However, reversing a list requires the presence of all the items. Furthermore, if items are added to the front, then the "first" item in the list will be the last one actually produced, which presents the same problem.

If a list represents an ordered sequence of data, implementing a list future as a FIFO–queue is preferable. This requires conscious thought on the part of the programmer, but typically involves *reduction* of actual complexity, as lists do not need to be reversed, or items can be processed in the natural order. List futures can be easily implemented using the CML `mailbox` type, which implements a buffered asynchronous channel. As with the types discussed in section 10.1, the decision to distribute futures inward to the elements of the list requires judgement of the costs and benefits of doing so.

It is often the case, however, that lists are used for purposes *other* than ordered sequences. It is common, for instance, to implement sets or even maps using a list. In these cases, a FIFO–queue will not perform as well. The reason for this is that the list's *structure* no longer exactly expresses the *meaning* of the data it organizes. Data structures of this kind require considerably more thought in the implementation of their futures. These sorts of structures are discussed in chapter 11.

Chapter 11

Indirect Structures

Chapter 10 discusses the use of futures to implement structures which can be consumed by one set of threads even as they are being produced by another. A common assumption in these designs was the similarity between a complex structure's implementation and its meaning, or more precisely, a correspondence between *syntax* and *semantics*. For instance, in the case of syntax trees, each node represents some element of syntax in a structure. Likewise, when a sequence of elements is implemented as a list, the "next" node in the list directly corresponds to the "next" node in the sequence.

However, the techniques described in chapter 10 are unsuitable for data structures where this is not the case. In many structures, the implementation of the structure is a matter of convenience or efficiency, while the actual *meaning* of the structure is dependent upon global, not local properties. For instance, an unordered set which happens to be implemented as a list is completely apathetic to the actual ordering of the list; all that matters is the question "does the list contain a given element". It is important to note that for the most part, structures addressed in chapter 10 are equivalent only if they are *structurally equivalent*, whereas structures addressed in this chapter may be equivalent, yet have very different internal structures.

Applying the straightforward strategies described in chapter 10 naïvely may result in widely varying performance. In the case of the list–set, the producer of the set may happen to generate the elements of the set in the exact order in which the consumer accesses them, in which case the list–set will perform quite well. However, if a consumer happens to attempt to access the element which the producer will generate *last*, then the list–set performs no better (and probably worse) than it would if consumers blocked until the entire set was complete.

This chapter addresses structures where the implementation and meaning diverge, which I term *indirect* structures. It discusses methods for implementing futures of these structures which are affected only by the semantic meaning of the structure, not its syntactic structure. The chapter opens with a discussion of general strategy, then applies the strategy to sets and maps[1]. Finally, it discusses methods for further improving the concurrency of functions which translate one set- or map–futures into another.

It is important to note that this chapter considers *only* cases where a structure is produced by one thread, and consumed by another, in which the consumer performs no modification to the structure. Cases in which a structure is modified in–place are considered in chapter 12

[1]The most common instance of a map is a hash–table.

```
signature SET =
sig

  type 'a set

  add : 'a * 'a set -> 'a set
  member : 'a * 'a set -> bool
  remove : 'a * 'a set -> 'a set
  ...

end
```

```
signature MAP =
sig

  type ('k, 'a) map

  val insert : 'k * 'a * ('k, 'a) map ->
               ('k, 'a) map
  val lookup : 'k * ('k, 'a) map -> 'a
  val exists : 'k * ('k, 'a) map -> bool
  val remove : 'k * ('k, 'a) map ->
               ('k, 'a) map
  ...

end
```

Figure 11.1: Partial Signatures for Sets and Maps

11.1 Futures for Sets and Maps

A *set* is an unordered collection of unique elements. A *map* is a set containing associations, or key–value pairs. In terms of implementation, however, it is often more convenient to characterize sets as maps from some type to `unit` (a non–existent value) for the purposes of an actual implementation. Figure 11.1 shows a simplified signature for maps and sets. In this and other examples in the chapter, some simplifications are made from the Standard ML utility library for the sake of brevity. In the actual Standard ML utility library, these signatures require the elements of a set or the keys of a map to be defined in a structure of their own, which also supplies an order function.

Unfortunately, the concept of an "unordered collection of elements" does not have a straightforward implementation on traditional computing hardware. Implementations must create some other data structure, such as a list, a table, or a tree which is *interpreted* by some set of functions. In the case of maps, a hash–table is an example of such a structure, designed to implement an imperative map, while a tree–map is a common implementation of a purely–functional map[115]. In both cases, the internal representation is a matter of efficiency, while the actual meaning depends on global properties of the structure.

Application of the techniques of chapter 10 to a hash–table would yield an array of list–futures. This would perform reasonably well for "hits". However, for "misses", the algorithm must wait until the entire table is complete. Structures which make use of lists, balanced trees, or heaps will suffer worse performance problems, as the final internal structure may not be determined until late in the computation.

It would be better if the *operations* on the data structure, rather than the structure itself were treated as synchronous operations. This approach better captures the true nature of indirect structures, in essence creating a future on the *result*, while leaving the actual internal representation up to implementation. The remainder of this section describes a simple implementation of this for maps, and thereby, sets. Subsequent sections explore the case of ordered indirect data structures, and finally describe methods for efficiently moving data from one indirect–structure's future to another.

Figure 11.2 shows a simple implementation of a map future structure in Standard ML. This structure also gives a straightforward definition for a set, which is achieved simply by using an element as the first type parameter, and `unit` as the second. The structure uses lock–free hash–tables, particularly the `insertIfAbsent`

```
type 'a future
structure LFHashTab

signature MAP_FUTURE =
sig

  type ('k, 'a) map_future

  val insert : ('k, 'a) map_future -> 'k * 'a -> unit
  val find : ('k, 'a) map_future -> 'k -> 'a option future
  val finish : ('k, 'a) map_future -> unit

end

structure MapFuture : MAP_FUTURE =
struct

  type ('k, 'a) map_future =
    ('k, 'a future) LFHashTab.hash_table

  fun insert table (key, value) =
    let
       val f = future ()
       val ent = LFHashTab.insertIfAbsent table (key, f)
    in
      putIfAbsent (ent, SOME value)
    end

  fun find table key =
    let
       val f = future ()
       val ent = LFHashTab.insertIfAbsent table (key, f)
    in
      ent
    end

  val finish =
    let
       fun putNone future =
         putIfAbsent (future, NONE)
    in
      LFHashTab.app putNone
    end

end
```

Figure 11.2: An Implementation of a Basic Map–Future

```
fun pathology () =
  let
     val set = Set.empty
     val set1 = addTenRandomNums set
     val set2 = addTenRandomNums set
  in
    if Set.member (set1, 42) andalso
       Set.member (set2, 13) then
      set1
    else
      set2
  end
```

Figure 11.3: A Pathological Case for Futures on Purely-Functional Maps

operation to inject a future into the table. This future is then returned from this and all subsequent lookups. The `lookup` and `insert` functions are almost identical. Note that the `lookup` function returns a future, not an actual value. This will become important in later discussions.

The `future` type here is an alias for the CML `SyncVar.iVar` type; however, in this example, some modification to the type is necessary. Both the `insert` and `finish` functions run the risk of raising an exception if the `future` type follows the same semantics as the CML `ivar` type. Under CML, there exists no way to atomically check if an `ivar` is empty and `put` a value if it is. Hence, both functions may `put` a value into an already–full future for some pathological executions. This is averted by assuming the `future` type supports a `putIfAbsent` function, which atomically checks if a future is empty, and performs a `put` if it is.

This particular implementation has several limitations. It cannot be used in the typical style of a purely–functional structure, wherein modifications do not destroy the original copy. Furthermore, once all entries are present, there may exist futures which have been inserted in anticipation of a value which will never be present. These must be signaled as having no value using the `finish` function.

Lastly, implementations of the functions `app`, `map`, and others must block until the table is `finish`ed, which is impossible with the exact structure presented above. I will address the issue of a functionalizing the structure first, then describe methods for overcoming the need to `get` a future when translating one table to another, as well as how to properly implement the `map`–like functions.

Unfortunately, there exists no way to preserve purely–functional semantics on a map, and efficiently implement a map future. Figure 11.3 shows a pathological case, wherein two sets are created in a manner which relies on the functional properties of the original set, then a different one is returned depending on the contents of both sets. The problem arises from the fact that it is impossible to know when a given element has been *committed* to the final set of elements that will be returned.

A compromise exists based on the concept of "committing" an element to the final set. The functional properties of the data structure can be preserved by keeping a temporary set of changes using a purely–functional map or set. When these changes become inevitable, a `commit` function is called to "send" them to the consumer. Once committed, these changes cannot be undone. In compiler terminology, a `commit` should be performed at any point which dominates the exit of the producer function, and before which some

```
structure Map : MAP

signature FUNC_MAP_FUTURE =
sig

  type ('k, 'a) map_future

  val insert : 'k * 'a * ('k, 'a) map_future -> ('k, 'a) map_future
  val find : 'k * ('k, 'a) map_future -> 'a option future
  val commit : ('k, 'a) map_future -> ('k, 'a) map_future
  val finish : ('k, 'a) map_future -> unit

end

structure FuncMapFuture : FUNC_MAP_FUTURE =
struct

  type ('k, 'a) map_future =
    (('k, 'a) MapFuture.map_future,
     ('k, 'a) Map.map)

  fun insert (key, value, (mapfuture, changes)) =
    (mapfuture, Map.insert (key, value, changes))

  fun commit (mapfuture, changes) =
    (mapfuture, Map.empty) before
     Map.app (MapFuture.insert mapfuture) changes

  fun find (key, (mapfuture, _)) =
    MapFuture.lookup mapfuture key

  fun finish (mapfuture, changes) =
    (commit (mapfuture, changes);
     MapFuture.finish mapfuture)

end
```

Figure 11.4: A Map Future with Functional Features

number of `inserts` may have been performed. Note that in figure 11.3, the only such point occurs just prior to the exit of the function `pathology`, which will result in poor performance. Better performance will result if functions are structured such that commit–points are very common. Also note that if *every* insert is a commit–point, then the basic implementation in figure 11.2 suffices.

Figure 11.4 shows a functionalized map future, built using the basic structure from figure 11.2. This map future implementation uses a functional map to store changes. This change–log is used to insert all the accumulated operations. In order to avoid accumulating a large change–log, a fresh functional map is created with each `commit`.

This implementation takes advantage of the permissive semantics of the `insert` function in `MapFuture`. It is possible that two threads, both of which have a functional map future with a common "ancestor" may `commit`, and in doing so, `insert` a value which already exists (in this case, the intersection of their change–logs). A real implementation should provide both an `insert` and an `insertIfAbsent` operation.

The `FuncMapFuture` provides some degree of the benefits of a purely–functional data structure, though `commit` operations must still be inserted in order to "ferry" over data once its presence in the final structure is guaranteed. However, the `map`-style functions must still block until the structure is `finish`ed in most cases. Section 11.2 will discuss a method for dealing with this problem.

11.2 Pipelining Map and Set Futures

One of the major benefits of the methods discussed in chapter 10 was the ability to not only process a single structure as it was being created, but to be able to "pipeline" an entire *series* of such structures. In a multi–stage computation many phases can operate concurrently, each consuming and producing a structure. So far, the methods described in this chapter allow a single producer and consumer to overlap; however, because the structures require a `finish` operation before some futures can be evaluated, there is no way to achieve the sort of pipelining possible with other structures.

This section describes techniques for achieving this sort of pipelining. In summary, these techniques revolve around the ability to extract a value from one map future and insert it into another without ever actually requiring its value. At a glance, this is relatively simple. However, with more work, it becomes possible not only to blindly transport values from one map to another, but to perform transformations on them in the style of `map` or `filter` without needing to evaluate the future until it becomes available.

The techniques in this section make greater use of synchronous events[131]. The `event` type in CML and CML–like languages provides an elegant abstraction, which allows the result of a `get` operation on a future to be treated as a first–class value. Particularly useful is the `wrap` function, which permits a post–synchronization action to be attached to an event. This becomes vital for implementations of `map`, or for less direct translations between various futures, as it allows a synchronous function to be delayed until the future becomes available.

Figure 11.5 shows a map future suitable for pipelining. Note that the signature has changed, with the future now returning the `event` type from `find`, rather than the `future` type. Additionally, this structure adds an `insertEvent` function, which permits an event to be inserted into the table. This enables the pipelining of the table by permitting a value to be transferred from one table to another without ever performing a `get`

```
signature MAP_FUTURE =
sig

  type ('k, 'a) map_future

  val insert : ('k, 'a) map_future -> 'k * 'a -> unit
  val insertEvent : ('k, 'a) map_uture -> 'k * 'a option event -> unit
  val find : ('k, 'a) map_future -> 'k -> 'a option event
  val finish : ('k, 'a) map_future -> unit

end

structure MapFuture : MAP_FUTURE =
struct

  datatype 'a entry = Future of 'a option future * 'a option event
                    | Event of 'a option event

  type ('k, 'a) map_future = ('k, 'a entry) LFHashTab.hash_table

  fun insertEvent table (key, event) =
    case LFHashTab.insertIfAbsent table (key, Event event) of
      Future (f, _) => putIfAbsent (f, SOME value)
    | _ => ()

  fun insert table (key, value) =
    insertEvent table (key, always (SOME value))

  fun find table key =
    let
       val f = future ()
    in
      case LFHashTab.insertIfAbsent (key, Future (f, getEvt f), table) of
        Future (_, evt) => evt
      | Event evt => evt
    end

  val finish =
    let
       fun putNone (Future (f, _)) = putIfAbsent (f, NONE)
         | putNone _ = ()
    in
      LFHashTab.app putNone
    end

end
```

Figure 11.5: A Map–Future Implementation for Pipelining

operation.

This structure lays most of the groundwork for pipelinable map futures. However fully implementing the `map`, `mapPartial`, and `filter` family of functions requires further modifications. Efficiently implementing the `fold` operation requires an extension to CML event combinators to support an efficient, unordered fold operation over a list of `event`s.

Implementing the `map` family of functions is not as simple as moving all entries in the source table to the destination. It will often be the case that entries will be inserted into the source after the point in time at which the `map` takes place, which will result in an incorrect result. Therefore, when a table is `map`ped (or `filter`ed or `mapPartial`ed), it must possibly defer lookups to the source table. This seemingly has a high potential cost. However, the cost is one–time, as once an event has been synchronized upon, further synchronizations yield the value immediately (this is similar to the amortization of costs in the union–find data structure).

Figure 11.6 shows a map–future which supports the functions `map`, `mapPartial`, and `filter`. In order to facilitate this, the structure of a map is expanded to include a function from the key type to an event. For freshly created tables, a constant function which always returns `always NONE` is used. For all other tables, the `find` function on the source table is used.

The most intricate portion of the structure's implementation comes from "joining" two events: the event derived from the previous structure and the event representing the entry being inserted into the current structure. Since neither event's result is apparent until it is synchronized upon and both events do indeed yield a value, either `SOME` value, indicating that the structure does indeed contain the requested value, or `NONE` indicating that it does not. This precludes the use of the `choose` combinator to "select" which event actually yields a value[2]. Concurrent ML does not provide either a join or sequential composition for events, and implementing a system which provides such a composition has proven to be quite difficult[40]. My work on this topic[91] has demonstrated that the combination of both sequential–composition (or join) and choice–composition is equivalent to solving existentially quantified boolean formulae (SAT).

A similar issue prevents the use of events for the `finish` function. At first glance, it seems possible to have a separate event which is synchronized upon by `finish`, and to use `choose` to decide between an event representing the actual value (representing the `get` function) and an event representing no value (the event synchronized upon by `finish`). However, there is no way to "join" the `finish` events of the source and the destination tables. An tempting, but flawed alternative solution is to have the `finish` function of the destination table synchronize on the `finish` event of the source before triggering its own event. However, this approach may fail if *both* the future event and the `finish` event have been activated, as `choose` may select the "wrong" one. Additionally, there is no way to force the event from the destination table to take precedence over the event from the source.

The solution takes advantage of the fact that the events returned by the `find` and `insert` will always become active, as a value will always be put into the future. This makes it pointless to include these events in a `choose`. Because of this, it is possible to synthesize the sequential composition of synchronization on both futures without the usual consequence of causing `choose` to misbehave. This is also beneficial, as it allows inserts to the destination table to take precedence over values inherited from the source.

The `app` and `fold` functions must still wait until the table is completely `finish`ed, as entries may be

[2]So-called "conditional synchronization" is impossible in either CML, or its underlying model, π–calculus.

```
structure PipelineMapFuture =
struct

  datatype 'a entry = Future of 'a option future * 'a option event
                    | Event of 'a option event

  type ('k, 'a) map_future =
    (('k, 'a) LFHashTab.hash_tab, 'k -> 'a option event)

  fun localmap (m, _) = m

  fun combine _ (SOME value) = SOME value
    | combine evt NONE = sync evt

  fun insertEvent (table, func) (key, event) =
    let
       val evt = wrap (event, combine (func key))
    in
      case LFHashTab.insertIfAbsent table (key, Event (combine event)) of
        Future (f, _) => putIfAbsent (f, SOME value)
      | _ => ()
    end

  fun insert fut (key, value) =
    insertEvent fut (key, always (SOME value))

  fun find (table, func) key =
    let
       val f = future ()
       val evt = wrap (getEvt f, combine (func key))
    in
      case LFHashTab.insertIfAbsent (key, Future (f, evt), table) of
        Future (_, evt) => evt
      | Event evt => evt
    end

  fun map func fut =
    (LFHashTab.create (), wrap (LFHashTab.find fut, Option.map func))

  fun mapPartial func fut =
    (LFHashTable.create (), wrap (LFHashTab.find fut, Option.mapPartial func))

  fun filter func fut =
    (LFHashTable.create (), wrap (LFHashTab.find fut, Option.filter func))

end
```

Figure 11.6: Modifications to Support Map-style Functions

inserted after the call completes, which will be missed by the two functions. As discussions above mention an explicit `finish` event, I have omitted such an event, as well as the `app` and `fold` functions from figure 11.6 to avoid confusion. The actual `finish` event is implemented as a `unit future`, the first future representing the source–table's, and the second representing the current table's. The same technique of having `finish` wait on the source table's `finish` event before triggering its own can be used here. Also, both functions must recursively union source tables with the current table, always giving the destination table's entries precedence. This is actually achieved fairly easy with higher–order functional programming, by composing the applied or folded function with another, which checks the destination–level table for a value with the given key, substituting that instead.

Set–futures can be implemented as a map–future with the element type as a key, and `unit` as the value. Map–futures will play an integral role in the implementation of futures for stateful computations, which are discussed in chapter 12.

Chapter 12

Stateful Computations

Chapters 10 and 11 have dealt with futures for data structures in a strictly consumer/producer model. This model is the norm in functional programming, wherein a structure is produced by one function. However, there are cases where in–place modification of a complex structure can be simpler and more efficient than walking and reconstructing it entirely. An example of such a case comes from the analysis phases of a compiler, which set various attributes for components of a program.

Modification in–place presents a problem for several reasons. The function modifying the structure may not follow any predictable pattern. In a structure representing a program, for instance, most attributes depend upon the values of other attributes (which, in turn depend on others). Since these structures can be complex or tree–like, and traversal of them often involves recursively walking down into them, the problem of the structure changing cannot be solved by simple synchronization.

The solution presented in this chapter makes use of the techniques described in chapter 11 for implementing map–futures. The essence of the technique presented in this chapter is the treatment of a stateful computation as an entity itself, and the creation of a future of its *results*. In short, the stateful computation is viewed as a function which produces a new structure, even though it does not. Section 12.1 describes the technique in detail, while section 12.2 explores the similarity between this technique and lazy–evaluation, and the benefits thereof.

12.1 Path-Keyed Maps

Techniques in the previous chapters have focused on a “pipelining” approach to concurrency, using futures to allow various phases of a program to overlap. Phases which modify a structure in–place, however, resist both pipelining and more basic methods of parallelization. The problem for pipelining stems from the fact that the assumptions of the producer/consumer model, which have been essential for the methods described in the previous chapters do not hold here. Additionally, in the worst case, the in–place modification follows a structure defined by a set dependencies, which may not be at all apparent from the structure itself.

The technique presented in this chapter creates a future for the *results* of such a computation. This future is treated exactly like a map–future, which is keyed by *paths* into the data structure to desired attributes. This method treats the map–future as an object representing the final results of the computation, and the paths as

```
type attribute = string * string

datatype tree =
  Node of attribute list ref *
          (string * tree) list

val modifyTree : tree -> unit
val subtree : tree * string -> tree
val attr : tree * string -> string

fun process tree =
  let
     val () = modifyTree tree
     val sub =
       subtree (tree, "students")
     val subsub = subtree (sub, "Bob")
  in
    attr (subsub, "rank")
  end
```

```
datatype tree =
  Node of (string * tree) list

type path = string list

val modifyTree : tree -> (path, string)
                          map_future

fun process tree =
  let
     val attrs = modifyTree tree
     val path = [ "Students", "Bob",
                  "rank" ]
  in
    wrap (lookup attrs path,
          Option.valOf)
  end
```

Figure 12.1: A Simple Example of a Tree–Like Data Structure

queries against the final result.

It bears mention that the techniques described in this chapter are intended for *modification* of existing values, not for values which are calculated and added to an existing data structure where there was no value before. The latter can be implemented quite simply using basic futures.

To create a future on modification in–place, all mutable fields are removed from a data structure. These will instead be accessed through paths, which act as keys into a map–future, which will be produced by the in–place modification function and consumed by subsequent phases of computation. The paths themselves can be derived from field names for records, indexes for arrays, keys for maps, and in other similar ways for other data types.

Figure 12.1 shows a simple example of the transformation described above. The `modifyTree` function walks the tree, adding or removing attributes, but not changing the tree structure itself. The `process` function runs `modifyTree`, then extracts one of the attributes of a node several levels down. The left side shows conventional methods, while the right shows the same program modified using the technique described in this chapter.

In the program on the right, `modifyTree` is assumed to be a function which creates a map–future, then starts a thread to modify the tree, returning the map–future as a result. Also note that the `process` function will return an event, rather than a concrete result. Since the table will hold a value for the given path, it is safe to assume the result of synchronizing on the event will always yield `SOME` value, hence `wrap`ping the event in the `valOf` function.

This technique works well for modification phases which make small changes, such as to attributes. If a phase makes major *structural* changes, however, it becomes significantly harder to traverse the tree structure, as traversals must query the result–future, then `wrap` further traversal functions around the result of the query. Keen readers may note that this method of traversal is a more–expensive, less–direct version of simply rewriting the structure using the techniques in previous chapters. When this is the case, it is better to revert to the producer/consumer model, and generate a whole new structure.

12.2 Lazy Concurrent Evaluation

Section 12.1 described the technique of using a map–future to represent the results of in–place modification of a data structure. This technique works by treating aspects of the result of in–place modification as queries, effectively reducing the problem to that of futures on associative data structures. This section explores how this technique can be adapted in a manner similar to lazy evaluation in order to compute the minimum set of values necessary for a result in the most concurrent fashion possible.

The method of using a map–future to represent the results of a in–place modification also serves to expose parallelism *within* the stateful computation itself. The final state of some given variable (or attribute, in the context of complex structures) can be thought of as a function of the final states of other variables, and so on. This gives rise to a dependency graph, whose structure may be very different from the actual structure being modified. For instance, in program analysis, a function is pure if it calls no impure functions, *or* if it calls impure functions in a manner such that they access only local data. Depending on the complexity of analysis, a function's purity may depend on many other aspects of the program.

In cases where there are no "loops" in the dependencies for the final values of attributes, the result–future technique provides a convenient way to parallelize the modification phase. If a given attribute is a function of others, then the result–future itself can be used to gather the necessary futures. Rather than start one thread which sequentially calculates the attributes for the entire tree, many threads are created, which calculate attributes on-demand.

Figure 12.2 shows a trivial example in which several functions which compute attributes of a data structure. These attributes depend on other attributes, which are requested from the `results` future, causing threads to be created to calculate them. This technique is very similar to *lazy–evaluation* in languages such as Haskell. A benefit of lazy–evaluation is that a value is only computed if it is necessary. This technique goes a step further by computing only the necessary values in a fashion which maximizes their concurrency. Previous work has examined parallel lazy execution of purely–functional languages[19, 54]. This technique executes in–place modification in a similar manner.

It should be noted, however, that some attributes require a pass over the entire structure, or a significant portion of it. For instance, deciding where a given value can be treated as a constant in a program requires a pass over the entire program. In these cases, applying the concurrent lazy evaluation technique will *reduce* performance, as a pass over the entire program will be made for every single value. In these cases, it is better to simply start a thread which makes a single pass over the entire structure, calculating the values of all attributes which require it[1].

As a final note, the case in which there is a cycle in the dependencies among attributes cannot be addressed by this method. Such a cycle must be addressed by global reasoning. Often this involves the detection of such cycles and their elimination based on some global reduction, iteration to a fixed–point, or finding a solution to some set of constraints[2]. While the algorithms for dealing with such cases can be parallelized

[1]This thread can still be *started* lazily.

[2]All three of these techniques are solutions to such a case in the context of compilers

```
fun gpa results name =
  let
     fun doGpa name = ...
  in
     spawnc doGpa name;
     wrap (lookup results [ "students", name, "gpa" ], Option.valOf)
  end

fun rank results name =
  let
     val path = [ "students", name, "rank" ]
     fun doRank "Bob" =
       let
          val mine = gpa results "Bob"
          val hers = gpa results "Alice"
          val res = if sync mine > sync hers then 1 else 2
       in
         insert results (path, res)
       end
  in
    spawnc doRank name;
    wrap(lookup results path, Option.valOf)
  end

fun process tree =
  let
     val results = modifyTree tree
     val path = [ "Students", "Bob",
                  "rank" ]
  in
    rank "Bob"
  end
```

Figure 12.2: Calculation of Attributes using Lazy Concurrent Evaluation

using techniques from this part, the cyclic dependencies prevent knowing any one attribute without knowing the others. Similarly, if dependencies form a long, singly–linked chain, Then there there exists no way to parallelize the computation. Dependencies are the chief limiting factor to parallelism, and if there is no fan–out in the dependency graph, then *no* technique can find parallelism in the algorithm, because it simply does not exist.

Aside from these two cases, which represent the limiting factors on the parallelism of a program, this chapter has described a method for implementing futures on the results of an in–place modification of a data structure, and extended the method to expose parallelism within the modification itself.

This concludes the discussion of false dependence, which began by introducing basic techniques for direct structures, and continued to describe techniques for implementing futures on indirect structures, namely sets and maps. These techniques serve to improve the performance of concurrent functional programs by shortening their critical paths, as well as allowing multiple phases of a program to operate at once, thereby allowing more threads to exist at once. The final section concludes this part by discussing the relationship between these ideas and concepts in computer architecture.

12.3 Relationship to Computer Architecture

As a final note, I will mention the relationship between the ideas presented herein and similar concepts in computer architecture. Because of the omnipresence of parallelism in architecture, many of its concepts and ideas appear in other domains as they relate to concurrency and parallelism. Because parallelism is readily available in hardware, methodologies used therein make excellent examples for other domains.

The techniques described in this part relate to architecture concepts in several ways. First, and most obviously, the notion of pipelining is common throughout all three chapters. Permitting stages to overlap benefits concurrent functional programming by shortening the critical path and producing more threads at any given time. Pipelining also has the potential to reduce memory usage, as many structures may never need to exist in a fully–complete state.

The concepts introduced in chapter 11 bear a resemblance to explicit speculation in VLIW architectures. Though the intent is different, the concepts are quite similar. Both feature the ability to perform speculative operations which yield possibly–nonexistent values, and to "chain" such operations, validating them at a later time.

Lastly, the techniques described in this chapter regarding stateful computation, particularly the lazy concurrent evaluation technique bear a strong resemblance to dataflow architectures. Dataflow machines compute values "back–to–front", using a graph structure (which can be implemented as an associative map in the manner I describe) in a very similar way to what this chapter proposes.

This similarity to hardware has been explored by other efforts. Past efforts, namely the π–RED^{+}[45] engine have examined the use of speculative execution in software. Additionally, the Bluespec language and semantics are inspired by hardware design, but bear a striking similarity to π–calculus. The ideas in these chapters, by contrast, were designed within the context of higher–order concurrency, and came to share many ideas with computer architecture. This apparent relationship may prove helpful in the future. As concurrency becomes increasingly important, concepts from hardware may inspire solutions in other domains.

Part IV

Conclusions, Retrospect, and Future Work

Chapter 13

Conclusions

This thesis has presented concurrent functional programming as a workable means of parallelizing programs in which there is very little coarse–grained concurrency. It has presented arguments from existing work on the parallelism in functional programming, as well as concrete examples of how functional programs readily admit parallelism. The thesis has discussed the two main factors which limit the viability of concurrent functional programming, and has presented work which aims to address these factors.

The thesis began by discussing concurrent functional programming as compared to the traditional model of concurrency. Traditional concurrency is dependent on the ability of the programmer to split workloads among several large threads. While this works for some types of programs, it is not always the case that coarse–grained parallelism can be found, or that enough exists to be worth the cost to create threads. Traditional threads generally carry a high cost to create and maintain, which means that for the most part, the number of threads must be small, and threads cannot be created or destroyed often.

Some programs have a structure which is inhospitable towards traditional parallelism. When a program's data structures are represented as tree- or graph–structures as opposed to arrays, parallel traversal via traditional parallelism becomes significantly harder. Furthermore, when the traversal order is not regular, but occurs in an order determined by the actual content of the structure, parallelizing the traversal is far more difficult still. Finally, multiple stages in a program introduce false data dependencies, which further limit parallelism.

Concurrent functional programming is able to deal with both of these cases. Because functional programming produces separable control flows in the form of function calls, the technique effectively exposes fine–grained parallelism, as exists in a tree–traversal. Furthermore, by constructing data structures intelligently, multi–stage computation can be pipelined. Unfortunately, the traditional runtime does not support this style of programming very well. Traditional threads are heavyweight structures with considerable setup and maintenance costs. The rapid creation of short–lived threads does not perform well if implemented directly on top of a traditional threading mechanism. Additionally, the multiple stages of computation are still a problem if they are not dealt with. Futures, which are one of the core mechanisms of concurrent functional programming can limit the parallelism of a program if they are used naïvely.

In summary, there are two limiting factors on the performance of concurrent functional programming. The first is the lack of runtime support. The second is the issue of false data dependence.

The second part of the thesis devoted itself to the issue of runtime support. Systems tend to be built as various components or layers, which provides the advantage that one layer can be changed independently of the others. However, the disadvantage of this is that layers which are infrequently changed (such as runtime systems) can become a liability if the basic assumptions under which they were created are no longer valid. I have argued in this thesis that this is the case with regard to concurrent functional programming and traditional program runtime systems. The traditional runtime system was built to execute single–threaded programs written in a C–like language. I have argued that a runtime for concurrent functional programming operates on fundamentally different assumptions and therefore should be built accordingly.

The design proposed in part II makes use of several design decisions which are seemingly expensive by themselves, but have a mutually beneficial interaction when combined. Heap–allocating frames is no more (and perhaps less) expensive than stack–allocation of frames in the presence of garbage–collection. This heap–allocation also gives a very simple, straightforward implementation of both threads and continuations. Saving a frame pointer is all that is needed to create a continuation, and threads require only the overhead of inserting a structure into the scheduler. Garbage–collection also has significant benefits in a highly–parallel programming environment, eliminating the lock–free memory reclamation problem and in many cases, the ABA problem. This lightweight implementation of threading, along with a lock–free scheduler implementation makes M:N threading feasible by eliminating some of the major problems with that approach. Lastly, safe–points, which are a mechanism commonly used for garbage collection also permit a very lightweight context switching and asynchronous messaging framework.

The thesis presented a lock–free scheduler implementation and a mostly lock–free garbage collector. The scheduler also makes use of existing work on lock–free memory allocation. The presentation of the scheduler developed a formal semantic model of M:N scheduling and proved the lock–free algorithms to be a correct implementation of these semantics. The garbage collector did not warrant a full formal model, as such work already exists. However, as most of the garbage–collector is lock–free, the thesis does present several strong guarantees and prove that the algorithms preserve them. These guarantees are strong enough to demonstrate correctness in conjunction with other work on provably correct garbage collectors.

The third part of the thesis developed the idea of futures for complex data structures. This part addressed the issue of false data dependence in a multi–stage program. Futures are used in concurrent functional programming to represent values which are being calculated by another thread. While this works for simple values, naïve use on complex structures introduces false data dependence, making subsequent values dependent on the entire structure, rather than just what is necessary. The third part introduced methods for implementing futures for complex data structures, beginning with structures whose meaning is derived directly from their representation. Following this, the thesis addressed structures which have a meaning which differs from their internal structure. Finally, the thesis moved away from the producer–consumer model. The final chapter dealt with representing futures on the results of computations which modify a structure in–place. The techniques in this chapter not only allow subsequent steps to proceed, but also provide a mechanism to effectively parallelize such in–place modification steps.

In addition to developing these ideas, I have developed and am continuing to develop a real implementation of the runtime system I propose herein. Unfortunately, as concurrent functional programming has suffered from lack of runtime support, there is not a large existing body of programs which can act as a benchmark. Therefore, exhaustively testing the runtime system's performance is extremely difficult. Moreover, there are no alternative approaches to test it *against* at the time of writing. Most concurrent functional language runtimes are either not parallel, or are proofs–of–concept, and building support for a concurrent functional language directly on top of a traditional runtime is a considerable (and fruitless) effort in its own right.

Considering these issues, this thesis has presented the runtime system more as a foundational system, presenting formal semantics and proofs of correctness. Since the runtime I have proposed is not an incremental improvement to existing systems, but an entirely new direction, it is impossible to present the sort of performance comparisons which are common for incremental improvements. However, if the runtime is to act as a foundation for concurrent functional programming and a starting point for future work on runtime support for it, then the design must be well–founded in all of its design decisions, its semantics, and its implementation. As such, this thesis has presented detailed arguments and proofs covering each of these areas in an effort to provide a sound foundation for future work.

Chapter 14

Retrospect

This chapter exists because I believe that as new models for concurrent programming come into existence, it will become necessary to undertake an effort similar to the one undertaken by this thesis. Because runtime systems are infrequently redesigned or changed, it is likely that at some point in the future, some new model of concurrent computation will suffer poor performance due to an outdated runtime system, just as I have argued that concurrent functional programming does now. Therefore, I have written this chapter, which outlines the thought process of this thesis in a more narrative style for the sake of future research efforts.

This thesis began as an investigation into the parallelization and distribution of a compiler. I chose this project, believing that compilers represent a pathological case for parallelism, as they combine several very difficult problems from the standpoint of classical pthread–style parallelism. My investigations were to center around whole–program compilation, as this style of compilation can require significant resources for even medium–sized programs.

I was also originally considering techniques for distributed compilation; however, I abandoned distribution in favor of parallelism as my approaches changed. Nonetheless, the techniques I describe for implementing futures and pipelining stages of computation can be adapted to a distributed setting. The end product of my thoughts concerning distribution was to split a program's call graph into sections, which would be compiled by separate machines. Since many of the optimizations performed by compilers[109, 80] have time–complexity greater than n^2, the costs of distribution are paid for in any sufficiently large compilation task. However, this technique relies on being able to split a call–graph into non–interacting subgraphs. This generally requires removing several functions from interprocedural optimization, or else duplicating or inlining commonly–called functions. Most unfortunately, this approach *does* compromise the quality of optimization, as it does not actually perform whole–program compilation, but rather whole–*subprogram* compilation for some (hopefully) loosely coupled set of subprograms.

My original approach was still very centered around the split–workload, shared–state approach to parallelism, as found in scientific computing, graphics, and other commonly–parallelized areas. Surprisingly, many compilation steps *can* be divided into split workloads, though without pipelining, the benefits are considerably less. However some troublesome cases remain. Most troubling are the analysis phases and register allocation. The analysis phases do not consume one structure in order to produce another, rather they "decorate" or modify an existing structure in–place.

Register allocation is even more troublesome. Register allocation is done by generating an interference graph and solving for a coloring. This is a complex, multistage, and infinitely worse, iterative fixed–point based algorithm. In short, it combines all the worst possible cases. My original approach to the problem was to modify Lal George and Andrew Appel's algorithm to eliminate the "give up and try again" methodology, replacing it with an approach which modifies the existing data structures and continues. As it turns out, part of the actual graph coloring algorithm itself (the part which derives an elimination ordering) can be parallelized by degrees with a worksharing approach similar to that used by the garbage–collector. However, using the ordering to color the graph is strictly non–parallel.

This effort demonstrated several things. First, forward–progress, that is not "giving up and starting over" is an important quality which should be preserved in order to maximize parallelism. The second observation was that any algorithm or technique which relies on strictly–ordered iteration, like dynamic programming or iteration to a fixed–point presents a very strong barrier, if not *the strongest* barrier to parallelism I have observed in the course of this research. Lastly, even in such cases, there often exists opportunities for parallelism *within* each iterative cycle, even if the iterative cycles themselves can neither be parallelized nor pipelined effectively.

In addition to register allocation[1], I was also investigating how to deal with the analysis phases found in the compiler. While register allocation presents a single monolithic parallelism challenge, the analysis phases are difficult in that they compute properties of functions and variables based on properties of others, and they modify the program structure in–place. Parallelizing the computation of properties is nontrivial, as a naïve approach may recompute some value several times or yield incorrect results (I was still working under the shared–state and split–workload way of thinking at this time). Also, compilation cannot continue until the entire analysis is complete.

I had already begun to explore the techniques described in chapters 10 and 11 at this time. Working on an unrelated and seemingly fruitless compiler phase: the final production of a program in its binary format nonetheless yielded several important lessons. The first was the ease and effectiveness of the concurrent functional style of programming at parallelizing even a phase which is seemingly inherently sequential. The second was the deleterious effects of stream–based I/O on parallelism. Rewriting portions of this phase using an indexed I/O scheme resulted in much greater capacity for parallelism. Further investigations into the idea of futures, as well as the concepts of monads and lazy–evaluation yielded the results in chapter 12, which provided a way to parallelize and pipeline the in–place–modifying analysis phases.

Embracing concurrent functional programming, however, raised new issues. While it seemed to almost effortlessly expose parallelism where the split–workload, shared–state way of thinking could only chip away, the actual support for concurrent functional programming is lacking. While many such languages have very lightweight, elegant 1:N threading models, those few which actually support real parallelism tend to directly map threads to kernel threads (executors, in my terminology), which does not perform as well as one would hope. As I mention in part II, traditional pthread–style threading assumes small, relatively fixed numbers of long–lived threads.

After reviewing literature, I suspected that concurrent functional programming may be able to achieve

[1]As it turns out, I independently discovered a result which others had also discovered about six months prior: that SSA–form programs generate chordal interference graphs, which can be colored in polynomial time with significantly more parallelism. This put an end to my efforts to improve register allocation. However, the lessons learned from this effort apply to other areas as well.

the same level of performance as more traditional styles of concurrent programming, and that the reason this performance is not realized is that the traditional notion of runtime environments strongly favors single–threaded imperative programming. With this in mind, I set out to design a runtime system from the ground up to fully support concurrent functional programming, the results of which are found in chapter 5. In the end, I arrived at a runtime architecture which combines elements of existing functional language runtimes, but completely dispenses with the traditional runtime system which usually sits underneath such systems. As I discuss in chapter 5, these components function well together. Of particular importance in my design was the absence of locks in the scheduler. This permits the executors to run without stalling: a problem encountered by other M:N threading systems.

Though scheduling was undoubtedly the central concern for the runtime system, I also sought to build a garbage collection system free from locks. I began with the Cheng–Blelloch garbage collector, which is highly parallel in its own right, seeking to remove all blocking synchronization. Unfortunately, this is impossible given the collector's "on/off" modus operandi. Mode switching which must be acknowledged by all threads is inimical to lock–freedom. However, my efforts to design a mostly–lock–free garbage collector did yield an algorithm which differs substantially from the one of Cheng and Blelloch, and requires only barrier synchronization at each mode–switch. This algorithm is very hospitable to real–time guarantees, requiring no mode–switch overhead other than the barrier, supporting modification of the heap during a collection, and operating within the confines of the normal scheduler system.

At this point the thesis had grown from parallelizing a compiler to something broader: a view of parallelism from the standpoint of concurrent functional programming, rather than the traditional coarse–grained approach. This view necessarily incorporates improvements to the critical communication mechanism (futures) as well as a new model of the underlying runtime system. The beauty of concurrent functional programming itself is that it follows almost effortlessly from ordinary functional programming. In a higher–order, polymorphically–typed functional language, it is possible to write a function which transforms an arbitrary function call into a thread, and returns the function call's result as a future. Of course, the preferred approach is to use the techniques in part III to allow pipelining of subsequent stages of computation. Regrettably, there does not exist a wide body of existing work which uses concurrent functional programming to acheive parallelism, *precisely because* the problems I aim to solve by the work of this thesis are not solved. This precludes exhaustive testing of the performance of the system and techniques I advocate[2]

This change in focus became the defining point of this thesis. The thesis' topic came to be the question of "how does one effectively make use of fine–grained parallelism", rather than simply "how does one parallelize a compiler". The answer, I concluded, is concurrent functional programming, and from here, I set out to solve the two major limiting factors on the effectiveness of this style of programming: false data dependence and the lack of runtime support. If these problems can be effectively solved, then programs can be parallelized by executing branches in their control flows as separate threads, and by minimizing data dependence in the program. In short, concurrent functional programming frames the issue of finding parallelism in a program as the much more tractable question of "how many threads can be kept alive at once", rather than the bleak picture painted by Ahmdahl's Law which rules the world of coarse–grained parallelism.

[2]This is a problem faced by both the transactional memory and functional programming research communities.

Chapter 15

Future Work

The first and most obvious opportunity for continuing my work in this area is to implement a working compiler using the techniques from this thesis. The only published effort to parallelize a compiler at the time of writing of this thesis is the implementation of the Glasglow Haskell Compiler[46] using parallel lazy evaluation techniques[19, 54]. I am aware from personal conversations[4, 106] that Intel, as well as others have attempted and abandoned such projects in the past. The successful implementation of a parallel compiler would demonstrate the viability of fine–grained concurrency.

In a broader sense, the use of parallelism in programs where there is no coarse–grained parallelism remains relatively unexplored from the standpoint of practical experience. However, if parallelism is to become the primary mechanism for improving performance of programs, this area must be not only explored, but mastered to the fine degree to which conventional (coarse–grained) parallelism has been developed. Furthermore, if work is to continue is this direction, it will be necessary to build a set of implementations which make use of fine–grained concurrency to act as benchmarks and test suites for various developments.

In addition to developing implementations of programs which exploit fine–grained concurrency using the techniques described in this thesis, there is additional work to be done in the area of programming language and library support for concurrent functional programming. The techniques described in part III require careful use to be beneficial to performance. Moreover, the constructs I have described are not organized into any taxonomy, but rather are presented in an order that makes sense from the standpoint of developing the concepts. Future work on these structures should focus on developing a more comprehensive collection of structures, similar to what exits for conventional (non–concurrent) data structures.

As far as programming languages are concerned, the primary existing languages (C++, Java) are simply inadequate for concurrent functional programming. The common functional languages (Haskell, OCaml, Standard ML, Lisp) suffer from their own pathologies, being generally unsuited for full–scale production environments. There are several notable production languages which are built with concurrency in mind, or are able to support the concurrent functional programming style. Two noteworthy examples are the Erlang and Bluespec languages. Erlang provides a Haskell–like purely–functional language with a explicit threading and asynchronous messaging. The hardware–centric Bluespec language provides what amounts to polyadic π–calculus: a concurrent execution model built on synchronous message–passing. Development of functional languages suitable for production use, as well as languages based on a concurrent programming model are necessary if use of fine–grained concurrency is to become commonplace.

It remains to be seen whether a truly lock–free garbage collector would indeed be beneficial. As previously mentioned, the need to switch the collector "on" and "off" prevents a lock–free implementation under the current scheme. A lock–free collector would most likely take an "always–on" approach, continuously reclaiming memory. While this may seem to have a considerable overhead, it is possible that a collector could be designed which is able to do this efficiently.

The runtime system developed in part II is based on the activation model, and relies only on the allocation of large memory blocks (slices). As such, it is naturally friendly to a port to bare–metal (meaning running directly on hardware, with no operating system). A bare–metal port of the runtime could serve as a foundation for both operating systems kernels written in higher–level languages, and for high–parallelism operating systems. The proofs of correctness provided in this thesis also establish a foundation for verified operating systems.

Lastly, the runtime system proposed in part II is only a starting point. Future work on runtime systems should aim to incrementally improve the performance of runtime systems built for concurrent functional programming, or else to propose new models by starting from the basics and making design decisions, as I did in chapter 5. The runtime architecture advocated by this thesis is a start; however, it is built for concurrent functional programming in languages like Haskell or ML, which are based on λ–calculus. More progressive concurrent languages which are based on a model of concurrency like π–calculus may suffer poor performance in the same way that concurrent functional programming suffers from a runtime built for imperative languages with coarse–grained concurrency. As a simple example, an inadequacy of the runtime system I propose in this thesis when used as a runtime for a π–calculus based language is that it assumes threads are *created*, rather than simply *existing* as they do in π–calculus[1]. Though concurrent functional programming is quite powerful when it comes to exploiting fine–grained concurrency, one cannot expect that it is the terminal end of progress in this area. As newer models are developed, it will again be necessary to provide the runtime systems to support these models and the programming methods to make the most of them.

[1]This is intended purely as an example to demonstrate a possible line of thought, not to imply that future research should adopt a strategy based this example.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Brian R. Murphy, Bratin Shah, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *27th ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2006.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4), 1993.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, Massachusetts Institution of Technology, 1986.

[4] Conversations with Ali–Reza Adl–Tabatabai.

[5] James Anderson. Multi–writer composite registers. *Distributed Computing*, 7(4), 1994.

[6] James Anderson, Ambuj Singh, and Mohamed Gouda. The elusive atomic register. Technical report, University of Texas, Austin, 1986.

[7] Thomas Anderson. The performance of spin-lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1), 1990.

[8] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler activations. In *ACM Transactions on Computer Systems*, 1992.

[9] Andrew Appel. Garbage collection can be faster than stack allocation. In *Information Processing Letters*, 1987.

[10] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[11] Andrew Appel and Trevor Jim. Continuation–passing, closure passing style. In *16th ACM SIGPLAN Principles of Programming Languages*, 1989.

[12] Andrew Appel and David MacQueen. A standard ML compiler. In *Conference on Functional Programming Languages and Computer Architecture*, 1987.

[13] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Sweden, 2003.

[14] Conversations with Arvind.

[15] Arvind and David Culler. Dataflow architectures. *Annual Review of Computer Science*, 1(1), 1986.

[16] Arvind, Rishiyur Nikhil, and Keshav Pingali. I–structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.

[17] David Bacon, Perry Cheng, and V Rajan. A real–time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices*, 38(1), 2003.

[18] Peter Bailey and Malcolm Newey. Implementing ML on distributed memory multiprocessors. In *Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, 1992.

[19] Clement Baker-Finch, David King, and Philip Trinder. An operational semantics for parallel lazy evaluation. In *5th ACM SIGPLAN International Conference on Functional Programming*, 2000.

[20] Paul Barth, Rishiyur Nikhil, and Arvind. M–structures: Extending a parallel, non–strict language with state. In *5th ACM Conference on Functional Programming Languages and Computer Architectures*, 1991.

[21] Gregory Benson, Matthew Butner, Shaun Padden, and Alex Fedosov. The virtual processor interface: Linux kernel support for user-level thread systems, 2007.

[22] Hans Boehm. Threads cannot be implemented as a library. In *26th ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2005.

[23] James Burns and Gary Peterson. Constructing multi–reader atomic values from non–atomic values. In *16th ACM Symposium on Principles of Distributed Computing*, 1987.

[24] Nicholas Carriero and David Gelernter. S/NET's Linda kernel. *ACM Transactions on Computing Systems*, 4(2), 1986.

[25] Perry Cheng. *Scalable Real-Time Parallel Garbage Collection for Symmetric Multiprocessors*. PhD thesis, Carnagie Mellon University, Pittsburgh, PA, 2001.

[26] Perry Cheng and Guy Blelloch. On bounding time and space for multiprocessor garbage collection. In *20th ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 1999.

[27] Perry Cheng and Guy Blelloch. A parallel, real–time garbage collector. In *22nd ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2001.

[28] William Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institution of Technology, 1981.

[29] Richard Cole and Ofer Zajicek. The APRAM: Incoprorating asynchrony into the PRAM model. In *1st ACM Symposium on Parallel Algorithms and Architectures*, 1989.

[30] Eric Cooper and Richard Draves. C Threads. Technical report, Carnegie Mellon University, 1988.

[31] Peter Damron, Alexandra Federova, Yossi Lev, Victor Luchango, and Daniel Nussbaum. Hybrid transactional memory. *ACM SIGPLAN Notices*, 41(11), 2006.

[32] Olivier Danvy and Julia Lawall. Back to direct style II: First–class continuations. In *ACM Symposium on LISP and Functional Programming*, 1992.

[33] Randall Dean. Using continuations to build a user–level threads library. In *3rd USENIX Mach Conference*, 1993.

[34] K. Debattista, K. Vella, and J. Cordina. Wait–free cache–affinity thread scheduling. *Software, IEE Proceedings*, 150(2), 2003.

[35] David Detlefs, Paul Martin, Mark Moir, and Jr. Guy Steele. Lock–free reference counting. In *21st ACM SIGPLAN Symposium on Principles of Distributed Computing*, 2002.

[36] Edsger Dijkstra. Cooperating sequential processes. *Programming Languages*, 1968.

[37] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in statically typed languages. In *13th ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 1992.

[38] Thomas Doeppener. Threads: A system for the support of concurrent programming. Technical report, Brown University, 1987.

[39] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *20th ACM SIGPLAN Symposium on Principles of Programming Languages*, 1993.

[40] Kevin Donnelly and Matthew Fluet. Transactional events. In *11th ACM SIGPLAN International Conference on Functional Programming*, 2006.

[41] Richard Draves, Brian Bershad, Richard Rashid, and Randall Dean. Using continuations to implement thread management and communication in operating systems. *ACM SIGOPS Operating Systems Review*, 25(5), 1991.

[42] Julian Elischer, Jonathan Mini, Daniel Eischen, and David Xu. KSE (kernel scheduling entity) library. Component of the FreeBSD Operating System.

[43] Joseph R. Eykholt, Steve R. Kleiman, Steve Barton, Roger Faulkner, Anil Shivalingiah, Mark Smith, Dan Stein, Jim Voll, Mary Weeks, and Dock Williams. Beyond multiprocessing: Multithreading the sunOS kernel. In *Summer USENIX Technical Conference and Exhibition*, 1992.

[44] Stephen Fortune and John Wyllie. Parallelism in random access machines. In *10th Annual Symposium on Theory of Computing*, 1978.

[45] Dieter Gartner and Werner Kludge. π–red+: A compiling graph–reduction system for a full–fledged λ–calculus. In *ARCS*, 1993.

[46] The Glasglow Haskell Compiler. http://www.haskell.org.

[47] P. Gibbons. A more practical PRAM model. In *ACM Symposium on Parallel Algorithms and Architectures*, 1989.

[48] Seth Goldstein. *Lazy Threads: Compiler and Runtime Structures for Fine–Grained Parallel Programming*. PhD thesis, University of California, Berkeley, 1997.

[49] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[50] Irene Greif. *Semantics of Communicating Parallel Processes*. PhD thesis, Massachusetts Institution of Technology, 1975.

[51] Carl Gunter, Didier Remy, and Jon Riecke. A generalization of exceptions and control in ML–like languages. In *Functional Programming Languages and Computer Architecture*, 1995.

[52] Kevin Hammond and Greg Michaelson. *Research Directions in Parallel Functional Programming*. Springer–Verlag, 1999.

[53] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *10th ACM SIGPLAN Conference on Principles and Practices of Parallel Programming*, 2005.

[54] Tim Harris, Simon Marlow, and Simon Peyton-Jones. Haskell on a shared–memory multiprocessor. In *ACM SIGPLAN Workshop on Haskell*, 2005.

[55] Thomas Hart, Paul McKenny, and Angela Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *20th International Parallel and Distributed Processing Symposium*, 2006.

[56] Danny Hendler, Nir Shavit, and Lena Yerushlami. A scalable lock–free stack algorithm. In *16th ACM Symposium on Parallelism in Algorithms and Architectures*, 2004.

[57] John Hennessy and David Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 2003.

[58] Maurice Herlihy. Impossibility and universality results for wait–free synchronization. In *7th ACM SIGPLAN Symposium on Principles of Distributed Computing*, 1988.

[59] Maurice Herlihy. Wait–free synchronization. In *ACM Transactions on Programming Languages and Systems*, 1991.

[60] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5), 1993.

[61] Maurice Herlihy, Victor Luchango, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures. In *16th International Conference on Distributed Computing*, 2002.

[62] Maurice Herlihy and Eliot Moss. Lock–free garbage collection for multiprocessors. In *3rd ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[63] Maurice Herlihy and Eliot Moss. Transactional memory: Architectural support for lock free data structures. In *International Symposium on Computer Architecture*, 1993.

[64] Maurice Herlihy and Sergio Rajasbaum. Algebraic topology and distributed computing- a primer. In *Lecture Notes in Computer Science*, 1996.

[65] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *The Journal of the ACM*, 46(6), 1999.

[66] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2007.

[67] Maurice Herlihy and Jeanette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[68] Maurice Herlihy and Jeannette Wing. Axioms for concurrent objects. In *14th ACM SIGPLAN Symposium on Principles of Programming Languages*, 1987.

[69] Matthew Hertz and Emery Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *20th ACM SIGPLAN Symposium on Object–Oriented Programming, Languages, Systems, and Applications*, 2005.

[70] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. Technical report, Massachusetts Institute of Technology, 1973.

[71] Robert Hieb, Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first–class continuations. In *11th ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 1990.

[72] Tony Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), 1974.

[73] Tony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[74] Richard Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin Hertzberg. McRT–malloc: A scalable transactional memory allocator. In *5th International Symposium on Memory Management*, 2006.

[75] IEEE and The Open Group. *IEEE Standard 1003.1-2001*. IEEE, 2001.

[76] ISO/IEC. *ISO 9899-1999*. ISO/IEC, 1999.

[77] Paul Jackson. Verifying a garbage collection algorithm. In *11th International Conference on Theorem Proving in Higher Order Logics*, 1998.

[78] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[79] Jorg Keller, Christoph Kessler, and Jesper Traff. *Practical PRAM Programming*. John Wiley and Sons, 2001.

[80] Ken Kennedy and John Allen. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 1997.

[81] Leslie Lamport. The mutual exclusion problem, part I- a theory of interprocess communication. *Journal of the ACM*, 33(2), 1986.

[82] Leslie Lamport. The mutual exclusion problem, part II- statements and solutions. *Journal of the ACM*, 33(2), 1986.

[83] Leslie Lamport. On interprocess communication, part I. *Distributed Computing*, 1(2), 1986.

[84] Leslie Lamport. On interprocess communication, part II. *Distributed Computing*, 1(2), 1986.

[85] Jochen Liedtke. Improving IPC through kernel design. In *14th ACM Symposium on Operating System Principles*, 1993.

[86] Jochen Liedtke. On microkernel construction. In *15th ACM Symposium on Operating System Principles*, 1995.

[87] Virendra Marathe, Michael Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William Scherer III, and Michael Scott. Lowering the cost of nonblocking software transactional memory. In *1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[88] Brian Marsh, Michael Scott, Thomas LeBlanc, and Evangelos Markatos. First–class user–level threads. In *13th ACM Symposium on Operating System Principles*, 1991.

[89] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1991.

[90] Eric McCorkle. Simultaneous–consensus and the complexity of concurrent protocols. Submitted to 19th International Conference on Concurrency Theory.

[91] Eric McCorkle. Synchronous channels, choice, and transactions: Analysis, implementation and feasibility. Submitted to 19th International Conference on Concurrency Theory.

[92] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. *ACM SIGPLAN Notices*, 42(6), 2007.

[93] Austen McDonald, Jae Woong Chung, Brian Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *33rd International Symposium on Computer Architecture*, 2006.

[94] Marshall McKusick and George Neville-Neil. *Design and Implementation of the FreeBSD Operating System*. Addison Weseley, 2004.

[95] Maged Michael. Safe memory reclamation for dynamic lock–free objects using atomic reads and writes. In *25th ACM SIGPLAN Symposium on Principles of Distributed Computing*, 2002.

[96] Maged Michael. ABA prevention using single–word instructions. Technical report, IBM Research Division, 2004.

[97] Maged Michael. Hazard pointers: Safe memory reclamation for lock–free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6), 2004.

[98] Maged Michael. Scalable lock–free dynamic memory allocation. In *25th ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2004.

[99] Maged Michael and Michael Scott. Simple, fast, and practical non–blocking and blocking queue algorithms. In *15th ACM SIGPLAN Symposium on Principles of Distributed Computing*, 1996.

[100] Sun Microsystems. Multithreading in the Solaris operating system. Technical report, Sun Microsystems, 2002.

[101] Robin Milner. *A Calculus of Communicating Systems*. Springer–Verlag, 1980.

[102] Robin Milner. *Communicating and Mobile Systems*. Cambridge University Press, 1999.

[103] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[104] Jayadev Misra. Axioms for memory access in asynchronous hardware systems. *Transactions on Programming Languages and Systems*, 8(1), 1986.

[105] The MLton SML compiler. http://mlton.org.

[106] Correspondance on the MLton compiler developer's mailing list. http://mlton.org.

[107] Eric Mohr, David Kranz, and Robert Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. Technical report, MIT, 1990.

[108] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock–free FIFO queues. In *17th ACM Symposium on Parallelism in Algorithms and Architectures*, 2005.

[109] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[110] Alexandar Nanevski, Paul Govereau, and Greg Morrisett. Type theoretic semantics for transactional concurrency. Technical report, Harvard University, 2007.

[111] Rishiyur Nikhil. Id language reference manual. Technical report, Massachusetts Institute of Technology, 1991.

[112] Rishiyur Nikhil, Gregory Papadopoulos, and Arvind. T: A multithreaded massively parallel architecture. In *19th International Symposium on Computer Architecture*, 1992.

[113] Risiyur Nikhil, Arvind, J Hicks, S Aditya, L Augustsson, J Maessen, and Y Zhou. pH language reference manual. Technical report, Massachusetts Institute of Technology, 1991.

[114] Naomi Nishimura. Asynchronous shared memory parallel computing. In *2nd ACM Symposium on Parallel Algorithms and Architectures*, 1990.

[115] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1996.

[116] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Eliot Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental, and concurrent gc for servers. *ACM SIGPLAN Notices*, 37(5), 2002.

[117] Susan Owiki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5), 1976.

[118] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Annual USENIX Technical Conference*, 1999.

[119] Nikolaus Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.

[120] Simon Peyton-Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, 1996.

[121] Benjamin Pierce and David Turner. PICT: A programming language based on the π–calculus. Technical report, Indiana University, 1997.

[122] Benjamin R. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2002.

[123] Benjamin R. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[124] G. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, 1981.

[125] William Pugh. Fixing the Java memory model. In *Java Grande Conference*, 1999.

[126] William Pugh. The Java memory model is fatally flawed. *Concurrency- Practice and Experience*, 12(6), 2000.

[127] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *The 32nd International Symposium on Computer Architecture*, 2005.

[128] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, and Michael Jones. Mach: A system software kernel. In *34th Computer Society International Conference*, 1989.

[129] John Reppy. Asynchronous signals in standard ML. Technical report, Cornell University, 1990.

[130] John Reppy. Concurrent ML: Design, application, and semantics. In *Functional Programming, Concurrency, Simulation, and Automated Reasoning*, 1993.

[131] John Reppy. First–class synchronous operations. In *Theory and Practice of Parallel Programming*, 1994.

[132] John Hamilton Reppy. *Higher–Order Concurrency*. PhD thesis, Cornell University, Ithaca, NY, 1992.

[133] Davide Sangiorgi and David Walker. *The π–Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[134] Bratin Shah, Ali-Reza Adl-Tabatabai, Richard Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT–STM: A high performance software transactional memory system for a multi–core runtime. In *11th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2006.

[135] Ori Shalev and Nir Shavit. Split–ordered lists: Lock–free extensible hash tables. In *22nd ACM SIGPLAN Symposium on Principles of Distributed Computing*, 2003.

[136] Zhong Shao and Andrew Appel. Space–efficient closure representations. In *ACM Symposium on LISP and Functional Programming*, 1994.

[137] Nir Shavit and Dan Touitou. Software transactional memory. In *The 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995.

[138] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *2nd ACM SIGPLAN Workshop on Continuations*, 1997.

[139] David Siegwart and Martin Hirzel. Improving locality with parallel heirarchical copying gc. In *Proceedings of the 5th International Symposium on Memory Management*, 2006.

[140] Standard ML of New Jersey. http://smlnj.org.

[141] Stackless python. http://www.stackless.com.

[142] Christopher Strachey and Christopher Wadsworth. Continuations: A mathematical semantics for handling full jumps. In *Higher–Order and Symbolic Computation*, 2000.

[143] Andrew Tolmach. Tag–free garbage collection using explicit type parameters. In *ACM Symposium on LISP and Functional Programming*, 1994.

[144] Victor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving the correctness of highly–concurrent linearizable objects. In *11th ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, 2007.

[145] John Valois. Lock–free linked lists using compare–and–swap. In *14th Annual Symposium on Principles of Distributed Computing*, 1995.

[146] Joseph Vanderwaart and Karl Crary. A typed interface for garbage collection. In *ACM SIGPLAN International Workshop on Types in Language Design*, 2003.

[147] Mitchell Wand. Continuation–based multiprocessing. In *ACM Symposium on LISP and Functional Programming*, 1980.

[148] Daniel Wang and Andrew Appel. Type–preserving garbage collectors. *ACM SIGPLAN Notices*, 36(3), 2001.

[149] Peter Wegner. *Programming Languages: Information Structures and Machine Organization*. McGraw Hill, 1971.

[150] Luke Yen, Jayaram Bobba, Michael Marty, Kevin Moore, Haris Volos, Mark Hill, Michael Swift, and David Wood. LogTM–SE: Decoupling hardware transactional memory from caches. In *13th International Symposium on High Performance Computer Architecture*, 2007.

[151] Alexandre Zamulin. Formal semantics of java expressions and statements. *Programming and Computer Software*, 29(5), 2003.