BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

SIMPLE, SAFE, AND EFFICIENT MEMORY MANAGEMENT USING LINEAR POINTERS

by

LIKAI LIU

B.A., Boston University, 2004

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
2014

Approved by

First Reader

Hongwei Xi, Ph.D.
Associate Professor of Computer Science

Second Reader

Richard West, Ph.D.
Associate Professor of Computer Science

Third Reader

Mark Crovella, Ph.D.
Professor of Computer Science

Acknowledgements

I would foremost like to thank my advisor Professor Hongwei Xi for his patience when my progress is slow, for his discipline when my research direction goes off tangent, for his encouragement when I run out of steam, and most importantly, for advocating my work. I would also like to thank my former advisor Professor Assaf Kfoury for his instruction that laid the foundation for my understanding of programming language theories. Hongwei and Assaf were both pivotal in teaching me formal logic, including Linear Logic in particular, which is the basis of this dissertation.

I would like to thank Professor Richard West for nurturing my knowledge in operating systems which deeply underpins my dissertation. I would like to thank Professor Martin Herbordt of College of Engineering for introducing me the concepts of parallel computing and the hardware issues. I would like to thank Professor Mark Crovella for kindly being a reader as well as generously providing me with computing resources from his research group for running benchmarks. Likewise, I would like to thank Professor Stan Sclaroff and Professor Margrit Betke of the Image and Video Computing Group for generously allowing me to use their computing resources as well.

I would like to share the accomplishment of this dissertation with my parents whom without doubt I am deeply indebted to. This accomplishment is made possible because of my dad's hard work and my mom's meticulous upbringing and endless

sacrifices in order so that I could enjoy a stable household and a privileged education as I grew up. I look up to my grandfather Gong-Gong—who is now a historian studying the history of Taiwan since retirement from law practices—as a gentle, scholarly role model who inspired my intellectual curiosity. I also fondly enjoy the cuisine of my loving grandmother A-Ma.

I would like to thank friends and colleagues, former and current, at school and at work, for the intellectual discussions and just for keeping me company. I would like to acknowledge Ruggles Baptist Church for being my spiritual home since 2005.

Last but not the least, I thank God who is the creator of heaven and earth, who is the begining and the end, the God of Abraham, Issac and Jacob. When me and my parents suffered financial hardship and uncertainty, He provided and continues to provide us with more than what we could ask for.

> "And he said unto me, My grace is sufficient for thee: for my strength is made perfect in weakness. Most gladly therefore will I rather glory in my infirmities, that the power of Christ may rest upon me."—2 Corinthians 12:9, KJV.

SIMPLE, SAFE, AND EFFICIENT MEMORY MANAGEMENT USING LINEAR POINTERS

(Order No.)

LIKAI LIU

Boston University Graduate School of Arts and Sciences, 2014

Major Professor: Hongwei Xi, Ph.D., Associate Professor of Computer Science

ABSTRACT

Efficient and safe memory management is a hard problem. Garbage collection promises automatic memory management but comes with the cost of increased memory footprint, reduced parallelism in multi-threaded programs, unpredictable pause time, and intricate tuning parameters balancing the program's workload and designated memory usage in order for an application to perform reasonably well. Existing research mitigates the above problems to some extent, but programmer error could still cause memory leak by erroneously keeping memory references when they are no longer needed. We need a methodology for programmers to become resource aware, so that efficient, scalable, predictable and high performance programs may be written without the fear of resource leak.

Linear logic has been recognized as the formalism of choice for resource tracking. It requires explicit introduction and elimination of resources and guarantees that a resource cannot be implicitly shared or abandoned, hence must be linear. Early languages based on linear logic focused on Curry-Howard correspondence. They began by limiting the expressive powers of the language and then reintroduced them by allowing controlled sharing which is necessary for recursive functions. However, only

by deviating from Curry-Howard correspondence could later development actually address programming errors in resource usage.

The contribution of this dissertation is a simple, safe, and efficient approach introducing linear resource ownership semantics into C++ (which is still a widely used language after 30 years since inception) through linear pointer, a smart pointer inspired by linear logic. By implementing various linear data structures and a parallel, multi-threaded memory allocator based on these data structures, this work shows that linear pointer is practical and efficient in the real world, and that it is possible to build a memory management stack that is entirely leak free. The dissertation offers some closing remarks on the difficulties that must be addressed in order to support formal reasoning about a concurrent linear data algorithm.

Contents

List of Tables **xii**

List of Figures **xiii**

List of Abbreviations **xiv**

1 Introduction **1**

1.1 The Memory Allocation Problem 2

1.2 Memory Related Program Errors 4

1.3 Ways to Find Memory Errors and Their Limitations 6

1.4 Linear Logic 10

1.5 Inspirations of Linear Logic 12

1.6 Linear Object Ownership Semantics 17

1.7 Memory Hierarchy 18

1.8 Advances in Garbage Collection 24

1.9 Argument Against Parallel Garbage Collection 27

1.10 Organization of This Dissertation 30

2 Linear Ownership Semantics **31**

2.1 Object as a Smart Pointer 37

2.2 Linear Pointer 42
2.3 Linear Base Class 44
2.4 Linear Pointer and Linear Chromatic Pointer 49
2.5 Erasure 54
2.6 Idioms for Manipulating Linear Pointers 57
2.7 Idioms for Borrowing 60
2.8 Conclusion 62

3 Using Linear Pointer **64**

3.1 Singly Linked List 66
3.2 Singly Linked Segment 73
3.3 Singly Linked List Sorting 77
3.4 Augmented Linked List 81
3.5 Binary Search Tree 84
3.6 Splay Tree 88
3.7 Conclusion 96

4 Design of the Memory Allocator **98**

4.1 Zones and Zone Map 101
4.2 Heaps and Pool Management 105
4.3 Linearity Issues 109
4.4 Optimization 111
4.5 Benchmark and Results 115
4.5.1 Packed Size Classes 117
4.5.2 Lower Best-Fit Size Classes 122
4.5.3 Upper Best-Fit Size Classes 127
4.5.4 Upper Best-Fit Plus One 132

4.6 Conclusion 135

5 Linearity and Concurrency **137**

5.1 A Historical Perspective 138

5.2 Hardware Intrinsics 140

5.3 Linear Atomic Value 145

5.4 Linear Slots 148

5.5 Double-Ended Queue 150

5.5.1 Structural Invariants 152

5.5.2 The Implementation 154

5.5.3 Discussion 158

5.5.4 Related Work 159

5.6 Singly Linked List 162

5.7 Linear Transactional Pointer 165

5.8 Towards Theorem Proving 174

5.9 Conclusion 182

6 Conclusion **184**

A Comparing Memory Management Scalability Using Cilk and JCilk **186**

A.1 Description of Memory Management Methods 189

A.2 Scalability Metrics 192

A.3 Results 193

A.4 Conclusion 198

B Linear Pointer in C++11 **200**

Bibliography **206**

Vita **225**

List of Tables

4.3 Allocator versions used in the benchmark. 116

4.4 Error count. 117

4.5 Wall-clock time for packed configuration. 119

4.6 User CPU time for packed configuration. 121

4.7 Maximum resident set size for packed configuration. 123

4.8 Wall-clock time for lower best-fit configuration. 124

4.9 User CPU time for lower best-fit configuration. 125

4.10 Maximum resident set size for lower best-fit configuration. 128

4.11 Wall-clock time for upper best-fit configuration. 129

4.12 User CPU time for upper best-fit configuration. 131

4.13 Maximum resident set sizes, “no exchange.” 133

4.14 Maximum resident set sizes, “exchange.” 134

A.1 Elapsed wall-clock time. 195

A.2 Maximum resident set size. 196

A.3 Number of elapsed wall-clock seconds garbage collecting the young generation. 197

A.4 CPU utilization $(\mu + \sigma)/\tau$. 197

List of Figures

2.1 Temporary, lvalue reference, and rvalue reference assignments. 35
2.2 Temporary, lvalue reference, and rvalue reference for function argument and return value. 36
2.3 Pointer semantic classifications. 39

3.1 Top-down splay. 90

4.1 Utilization for packed configuration. 120
4.2 Utilization for lower best-fit configuration. 126
4.3 Utilization for upper best-fit configuration. 130

A.1 Speed up T_1/T_P based on elapsed wall-clock time. 195
A.2 Utilization normalized to the number of workers. 198

List of Abbreviations

ABA A-B-A (illustration of value state changes; not an abbreviation).

AMD Advanced Micro Devices (organization).

ANSI American National Standards Institute (organization).

ATS Applied Type System (programming language).

AVL Abelson, Velskii and Landis (tree data structure).

BIBOP Big Bag of Pages (memory allocation).

CPU Central Processing Unit (computer architecture).

DAG Direct Acyclic Graph (graph theory).

DDR Double Data Rate (computer architecture).

DRAM Dynamic Random Access Memory (computer architecture).

ELF Executable and Linkable Format (file format).

FIFO First-In First-Out (queuing).

GB Gigabytes, or $2^{30} = 1,073,741,824$ bytes (unit of data).

GC Garbage Collection (memory management).

GCC GNU Compiler Collection (software).

GHz Gigahertz, or $10^9 = 1,000,000,000$ cycles per second (frequency).

GLIBC GNU C Library (software).

GNU GNU is Not Unix (organization).

IBM International Business Machines (organization).

IPI Inter-Process Interrupt (computer architecture).

JVM Java Virtual Machine (software).

KB Kilobytes, or $2^{10} = 1,024$ bytes (unit of data).

LIFO Last-In First-Out (queuing).

LISP List Programming (programming language).

MB Megabytes, or $2^{20} = 1,048,576$ bytes (unit of data).

MHz Megahertz, or $10^6 = 1,000,000$ cycles per second (frequency).

MIT Massachusetts Institute of Technology (organization).

ML Meta-Language (programming language).

NUMA Non-Uniform Memory Architecture (computer architecture).

OS Operating System (computer architecture).

RAM Random Access Memory (computer architecture).

SML/NJ Standard ML of New Jersey (programming language).

STL Standard Template Library (software).

THE Tail Head Exception (concurrent queue data structure).

TLB Translation Look-aside Buffer (computer architecture).

VM Virtual Machines (computer architecture).

Chapter 1

Introduction

This dissertation describes a method of memory management using linear pointers which is a sweet spot for simplicity, safety, and efficiency. Linear pointer is a pratical alternative to formal verification, compile-time linear type checking, and run-time memory debugging for detecting memory errors. The idea is based on linear ownership semantics as a programming discipline inspired by Linear Logic, and the idea is simple to implement. One can argue that memory safety of a program is uncertain if one cannot ensure the safety of the underlying memory management subsystem. To prove that the new method provides the claimed safety benefit, a memory allocator is implemented using linear pointer, and its performance is compared to the state of the art parallel memory allocators to show that efficiency can be achieved.

Linear pointer makes manual memory management simple and safe. It allows low-level and high-performance programs be written without being subject to the achilles' heel of poor performance of garbage collection when running on today's parallel computers.

Contribution of this Dissertation This dissertation introduces a new smart pointer in C++ called linear pointer to ensure the safety of the program by means of linear resource management. Unlike existing smart pointers, linear pointer enjoys the erasure property which allows the program to achieve maximum efficiency. The erasure property allows the program to retain the same behavior with and without linear checking, and therefore a programmer can leverage the debug build of the program to find linearity violations but switch to a release build for production without the runtime overhead of linearity checking.

1.1 The Memory Allocation Problem

Memory is used by computer programs to store initial input, intermediate values, and final results of computation. In an abstract sense, memory is a look-up table from an integer numbered *address* to a *byte* which is the unit of data. One way a program can store an arbitrary amount of data is by grouping bytes occupying consecutive memory addresses into a *word*, and words into an *object* which is a representation of high-level program state. An object can refer to another object through a *pointer*. A pointer is merely a word stored in memory that is interpreted as the memory address of another object. Based on this one-dimentional memory, objects and pointers form *data structures* which allow the representation of higher-dimensional data.

Objects are created and destroyed during the course of a program's computation. When an object is destroyed, it is necessary to reuse the space for objects yet to be created so that the length of computation is not limited by the total size of all objects ever created; instead, computation could run indefinitely if memory used only by live objects is in an equilibrium (although in reality memory demand can fluctuate). Much of the memory management complexity is hidden by a memory allocator behind two

simple operations: given a natural number n, the function *alloc*(n) yields a pointer to an object of at least n bytes (the object is uninitialized, and it is the program's responsibility to initialize it); and given a pointer p, the function *free*(p) consumes the object at address p.

The one-dimensional view of memory is nevertheless the bane of the memory allocation problem, which is to decide object *placement*. Objects are typically fixed in placement once created until they are destroyed. This is largely a design decision due to the intricacy of updating object pointers. Merely moving the object content from one location to another is not enough, but all pointers to this object everywhere must be identified and updated to reflect the object's new memory address. If the object is shared by multiple threads of execution, then the update must appear atomically to all threads at once. Under this design decision, *fragmentation* could result from non-optimal placement: as objects are allocated and deallocated, gaps between objects begin to form that are too small for program use, yet the gaps altogether add up to significant amount of waste. The amount of waste depends on the allocation-deallocation sequence as well as the placement strategy.

In order for memory allocator to reuse memory, the allocator keeps a record of which memory addresses are unoccupied, by constructing an indexing data structure using free memory. In a sense, the memory allocator and the program take turns owning a block of memory: memory is free when it is owned by the allocator, and the memory is in use when it is owned by the program. Program's performance, which is measured by the amount of memory it uses and the overall running time, is heavily influenced by the placement strategy and the efficiency of the indexing data structure of the memory allocator. The emphasis is to index free memory in a way such that placement decision for a new allocation request can be made the most efficiently.

Both placement strategies and indexing mechanisms have been extensively sur-

veyed by Wilson, Johnstone, Neely and Boles in *Dynamic Storage Allocation: A Survey and Critical Review* [131].

1.2 Memory Related Program Errors

Memory is a resource governed by the law of conservation. A given memory location is either owned by the program or the memory allocator, but not both. The program has the further obligation that each object allocation must have a corresponding deallocation (and vice versa). Programs that fail to meet this obligation are subject to the following pitfalls.

- Objects are allocated but never freed (i.e. orphaned), resulting in *memory leak*.
- Objects may be in possession by different parts of the program. Any one of the stake holders may free the object, which can result in the object being returned to the memory allocator several times. If the memory allocator is not prepared to handle this situation, this may corrupt the free memory index. This problem is called *double free*.
- Objects referenced by multiple stake holders may be freed by one holder while the object is still being accessed by others. The memory allocator assumes that it has sole ownership of this memory and overwrites the object for indexing free memory, corrupting its content. This memory may be subsequently allocated for a different object. The original stake holders are said to have *dangling pointer* because the pointer no longer points to a valid object.

A program exhibiting any of these symptoms is hard to debug because the program often does not abort immediately. Instead, the program continues to make progress

in an inconsistent state and computes an incorrect result. This incorrect result sometimes causes another seemingly valid part of the program to abort. As a result, automatic memory management solutions are developed to mitigate the memory leak, double free, and dangling pointer issues.

Sometimes the programming paradigm makes it inevitable for a program to suffer memory leak. For example, in functional programing, computation is carried out by reducing mathematical expressions into a value. The computation is *pure* in the sense that reduction keeps track of no states and induces no side-effects. Functional programs manipulate data structures by creating a new copy of all modified objects in the data structure while sharing pointers to unmodified objects. Old objects are simply discarded because the program has no knowledge about whether the object might still be referenced by another data structure. To control memory usage, functional languages employ a form of automatic memory management that scans memory for all reachable objects and reclaims the unreachable ones. This is called *garbage collection*. The first garbage collector was invented for the functional programming language LISP in 1960 [92, 93]. Garbage collection has also been adapted for use by the "traditional" imperative languages (e.g. C, Pascal) in 1988 [19], and integrated into the imperative language Java in 1995 [57].

Sometimes the program requires only simple sharing and needs a way to find out when an object can be safely freed. As an alternative to garbage collection, the program can use reference counting for automatic memory management [29]. In this scheme, each object has a counter that is incremented by one when a new reference to the object is created, and decremented by one when a reference is removed. The object is freed when its reference count drops to zero.

There is another similar "missing object" error that is of interest: *null pointer dereference*. Often, a program would put zero as a special value for a pointer to

indicate the absence of an object that is normally expected to be present. This is called a `NULL` pointer. Memory at address zero would be reserved and not used to place any object. The `NULL` pointer could be passed from one part of the program to another to indicate an error condition; for example, *alloc*() would return `NULL` when all memory is exhausted. A program that forgets to check for `NULL` pointer would attempt to access memory at the zero location or locations of small offsets from zero (i.e. members of this fictitious object). For this reason, most modern operating systems configure the hardware to trap the program if the program tries to access this reserved memory. Note that automatic memory management alleviates the memory leak, double free, and dangling pointer problems, but a program using automatic memory management may still attempt null pointer dereferencing. Null pointer deferencing is relatively easy to diagnose and fix because the program aborts immediately and leaves a data path that can be traced to the source of the `NULL` pointer.

1.3 Ways to Find Memory Errors and Their Limitations

Besides garbage collection, there are other approaches for finding memory errors in a program that uses manual memory management. It can be done through compile-time static analysis, formal verification, and run-time error detection.

Static Analysis By constructing a control-flow graph that models how values are passed from one variable to the next, static analysis formulates resource tracking as a graph reachability problem. This control-flow graph must be generated through a whole program analysis. The graph-based approach makes static analysis neither

provably sound nor complete because the control-flow graph could not model loops, nor could it model runtime data structures which are not compile-time program variables. As a result, static analysis often has false positives (declaring a leak while there is not) and false negatives (not detecting a leak when there is). Furthermore, it is difficult to assess the effectiveness of static analysis algorithms because each positive within a corpus must be manually verified, and it is unknown how many false negatives are left in a corpus. Sabre [113] is the state of the art static analysis tool; the paper contains a survey of other static analysis designs.

Type Theory When given a programming language defined mathematically using grammar and structural typing rules, and a program written in that language, the type-theoretic approach verifies correctness of the program by recursively checking the sub-expressions of the program according to the structural rules of the language. The language rules would be able to express inductively defined data structures used by the program just as it would for recursively defined functions in the program. The structural typing rules of the language makes it possible to prove the soundness of the type system using mathematical induction [97]. Once proven sound, these systems will not have false positives, and their false negatives are generally well-understood just by examining the structural rules. Some of these methods are described in the following section.

Formal Verification At a lower level than the type theoretic approach, formal verification expresses properties of the program in a formal logical system and check the proof that the program satisfies these properties using an automated theorem prover. In general, although proof search is intractible and may even be undecidable for some logical systems, once a proof is produced, checking the proof can be done in

polynomial time. Proof written by a human is machine-checked by the theorem prover program in order to eliminate human errors. The correctness of the theorem prover itself has to be verified separately. This approach is similar to type checking because of the Curry-Howard correspondence that a type is a theorem, and a program having that type is a proof of that theorem. Some popular theorem prover such as Isabelle [94] and Coq [16] are also called proof assistant because they can reconstruct some parts of the missing proof, which reduces the human burden of writing the proof.

Formal verification has been done for the memory allocator of an embedded operating system called Topsy [91] using Coq, and for the memory allocator of the seL4 microkernel [118] using Isabelle. Other parts of seL4 were also subsequently verified [77]. Both Topsy and the seL4 allocator essentially represent a heap model where allocated blocks are non-overlapping, and that allocation and deallocation change only the state of the block in question but preserve the rest of the heap. They both use separation logic to model imperative heap mutation in a high order logical framework similar to how monad works. However, this method of reasoning about the heap has the following limits:

1. The heap is finite sized and cannot expand dynamically.

2. Blocks within the heap are modeled as an ordered sequence from lower to upper address.

As a consequence of this encoding, both malloc() and free() operate on blocks structured like a singly linked list which requires $O(n)$ traversal. In particular, the proof of free() is essentially a proof of insertion sort because the insertion into the free list must maintain block address order. Their methodology precludes using a more efficient indexing mechanism for free blocks such as segregated free list or binary search tree. The adventurous reader could consult their formal model and proof [119]

spanning over 100 pages.

Run-time Error Detection It is possible to detect memory access violation in run-time by annotating each memory location with a status, whether it is allocated but uninitialized, allocated and initialized, and freed. Run-time error detection may use binary instrumentation to add the instructions necessary to update and check the state transition of this status whenever a memory location is accessed. One may also modify the compiler to insert instrumentation in compile time.

Run-time error detetion techniques do not actually track object ownership transfers, so the approach is rather limited. Although the memory location status allows dangling pointer dereference errors to be caught immediately when it happens, leaks could only be detected by finding unreachable objects at a coarse interval and when the program terminates. In post-mortem analysis, even though a leaked object can be associated with a call-site or stack-trace, the contextual information such as function arguments or local variables on the stack frame that might help debugging the leak are no longer present, or would be extremely costly to retain. The deferred reporting makes it hard to debug memory leaks.

Purify [64] is the earliest run-time error detection tool using a conservative mark-and-sweep garbage collector for tracing object reachability in order to detect memory leaks. Like some of its predecessors, it also detects array bounds errors using red-zones around memory blocks, and uninitialized memory access using tagged shadow memory. The memcheck tool in Valgrind [102] instruments malloc() and free() to detect allocated but not yet freed memory at program termination. Dr. Memory [20] is a refinement of the technique used in Purify such that it identifies "possible leaks" which are memory blocks referenced only by a pointer to the middle of the block, but eliminates many common cases that cause this to happen such as C++

new[], pointer to base class in multiple inheritance, and memory layout of certain STL classes. Address sanitizer detects run-time memory error using compiler-inserted instrumentation [101].

1.4 Linear Logic

In 1987, a formalism to reason about programs manipulating resources that obey the law of conservation was introduced by Jean-Yves Girard, which he calls Linear Logic [55, 56]. Linear Logic is a logical system that strengthens classical logic in the sense that propositions in Linear Logic must to be used in a proof exactly once, except through explicit exponential operators. In classical logic, logical propositions may be implicitly used more than once (contraction) or none at all (weakening). Girard observed that application of Linear Logic to programming languages is imminent because of Curry-Howard correspondence, which relates an intuitionistic logic proof to a computer program. An example of the correspondence is the modus ponens, which is related to function application.

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \quad \text{corresponds to} \quad \frac{\Gamma \vdash (\lambda x.M) : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x.M)\ \ N : B}$$

In 1988, Yves Lafont, a student of Girard, proposed Linear Abstract Machine which runs a simply-typed linear lambda calculus [80]. In this language, merely accessing an object during computation will consume it. The main idea is to allow in-place modification (destructive update) of data structure in a functional programming language without using reference assignment, since assignment is impure and causes side-effect. The linearity prohibits sharing and orphaning of data structure objects. In-place modification is achieved by reusing memory immediately after it has been

consumed by the access. Certain objects may be deep-copied and disposed using built-in primitives, which corresponds to exponential operators for contraction and weakening. Lafont also pointed out that a linear program produces data structure that is necessarily a *tree* due to the absence of sharing, and no garbage collection is needed in this language.

However, Lafont's language is too restricted; it is trivially strongly normalizing because there are no recursive constructs. Fixed-point combinators cannot be written in linear lambda calculus because the combinators are not linear. They use variables f and x more than once.

$$\begin{aligned} \mathbf{Y} &= \lambda f.\,(\lambda x.f\ \ (x\ x))\ \ (\lambda x.f\ \ (x\ x)) \\ \mathbf{Z} &= \lambda f.\,(\lambda x.f\ \ (\lambda y.\,(x\ x)\ \ y))\ \ (\lambda x.f\ \ (\lambda y.\,(x\ x)\ \ y)) \end{aligned}$$

This is a consequence of treating functions in the language as linear objects. Although certain kind of objects can be copied, functions are not one of them. Making a copy of a function requires making a copy of everything in its environment.

Philip Wadler followed up in 1990 with a simply-typed language of mixed linear and non-linear terms and types [124]. The idea is that recursive functions are permitted in the non-linear language, but not in the linear language. However, restrictions apply: non-linear functions cannot have linear variables in the environment. If a non-linear function is never called, then the linear variable is never consumed, which leads to resource leak. If a non-linear function is called more than once, then the linear variable is consumed more than once, which leads to double-free. This prompted further work concerning how to interpret non-linear functions in a linear setting [3, 90, 120].

Wadler mentioned array access as an example use of linearity in a functional

language. In an imperative language, array updates are inherently done in-place, and updates to a shared array are visible throughout the program as side-effect. In a functional language, array updates must be without side-effect, so the entire array must be copied. An alternative is to represent an array as a sequence of complete binary-tree data structure, with $\Theta(\lg n)$ lookup and update time, which is described in Chris Okasaki's thesis [95] as random access list. If the array is linear, then it can be updated with $\Theta(1)$ time in-place because no other parts of the program has a reference to the array, and hence would not see the side-effect. However, strictly-linear array lookup would consume the entire array, and Wadler proposes some form of controlled sharing (a let-binding that temporarily treats a linear value in the sub-context as non-linear) to mitigate this problem.

One line of follow-up research explores the Curry-Howard correspondence for intuitionistic linear logic, leveraging the exponentials in linear logic to express controlled sharing [125, 126, 127, 3]. It has been shown that reference counting can be used to support memory management needs with explicit sharing and orphaning [25, 26]. Another line of research on the issue of in-place update explores type-inference of the mixed language (simply-typed [125], Hindley-Milner polymorphism [121]) so that the compiler can determine at compile time whether a value is linear, and hence can be destructively updated as an optimization.

1.5 Inspirations of Linear Logic

Sometimes it is useful to escape the framework of Curry-Howard correspondence which tends to distracts us from practical issues of resource conservation. For example, it is common to assume that linear objects may reference both linear and non-linear objects, and non-linear objects may only reference non-linear objects. This

is sound in theory, but in practice this is not how memory management works. It is okay for garbage collected objects to reference manually allocated objects as long as the garbage collector calls the object's finalizer to free the manually allocated object. On the other hand, it is not safe for a manually allocated object to reference a garbage collected object because the garbage collector does not know how to traverse the references made by manually allocated objects, and the referenced object would be garbage collected under our noses, leaving a dangling pointer. If we see manually allocated objects as linear, and garbage collected objects as non-linear, then we should allow non-linear objects to reference both non-linear and linear objects, but linear objects may only reference linear objects. This contradicts the theory.

Breaking away from Curry-Howard correspondence also gives us languages that are merely inspired by linear logic but have interesting properties or applications. For example, in 1988, Sören Holmström presented a variant of Lafont's linear lambda calculus that enforces linearity in the operational semantics instead of in the type system [70]. Programs in this language is computed by interchanging objects with collection using composers and decomposers. Objects are linear values. Collection is a mapping from variable names to objects. Composer creates an object by removing a subset of the collection, and decomposer adds name-value bindings to the collection after destroying the object. Although not originally intended by Holmström, this presentation is perhaps more suitable for an imperative linear language.

In 1992, Henry Baker's work shows that even a dynamically typed language like LISP can leverage linearity to simplify its memory management [9], that the language can be used to implement the quick sort algorithm [11], and the language can translate efficiently to a stack-based machine [10]. Baker also suggested that linear objects are suitable for parallel computing because the fact that linear objects can only be owned by one thread means their accesses do not require synchronization or cache coherence

[12].

In 1999, Karl Crary, David Walker, and Greg Morrisett devised a Calculus of Capabilities [34] for region-based memory management. Region memory management was conceived to be a middle ground between stack-based allocation and garbage collection [117]. Regions are arranged last-in first-out like a call stack, except a region has unbounded size. The original idea is to let the compiler infer object lifetime automatically and decide which region on the stack will be used to allocate the object. Although region inference has short-comings [116], region memory management has been adopted as a manual memory management technique where objects are allocated from the same region if they are expected to be all freed at once; rather than freeing objects individually, we just free the region which relinquishes the memory for all objects in the region. As a manual memory management technique, regions no longer need to be stack-like. Reference counting on region handles determine if a region can be safely freed [129].

In Calculus of Capabilities, instead of enforcing linearity of objects, it only enforces the linearity of *capabilities* for accessing a region *handle*. Handles are not linear, so they can be passed around. However, capabilities are required for dereferencing region handles. Capabilities are erased after the program is compiled, leaving only region handles at runtime. The distinction of linear capability and non-linear handle later gave rise to Alias Types [108] in 2000, which distinguishes linear location constraints from non-linear pointers at the object level rather than the region. A location constraint indicates that an object of a particular type can be found at some abstract memory location (this is similar to a "collection" in Holmström's language). A pointer is an integer representing a memory address, but the pointer value is dependent on the location it represents, and this is reflected on the pointer type itself (pointer type is a form of dependent type). Otherwise pointers can be shared. The

following year, Alias Types is extended to be able to define recursive data structures [128] such as singly linked list and binary trees, and then found its way into Cyclone [69] in 2004, a "safe" dialect of C, by Morrisett et al.

In 2001, Alias Types inspired Vault [39], a language for writing device driver and operating systems by Fähndrich et al. at Microsoft. In Vault, linear capabilities are called tracked keys, which are used for two disjoint purposes: for tracking the aliasing of non-linear object references, and for tracking the state transition of a linear object (e.g. a file object can be either opened or closed, but not both at the same time). When tracking state transition, an object method is annotated with preconditions and postconditions expressed in terms of tracked keys. Precondition indicates that a method may only be called when certain tracked keys are present, consumes the keys, and yields the new keys in the postcondition. Later on, controlled aliasing of linear objects in a local scope is achieved through Adopt and Focus mechanism [50], in order to allow non-linear objects to enforce state transitions as well. This is similar to Wadler's let-binding which allows temporary non-linear semantics of a linear value. The notion of object method state transition later made its way into Singularity OS and the Sing# programming language [49].

A shortcoming of these linear inspired type systems is the difficulty of dealing with conditional branching. In [69],

> "At join points in the control-flow graph, our analysis conservatively considers a value consumed if there is an incoming path on which it is consumed. For instance, if p is not consumed and we write:
>
> if (rand()) free(p);
>
> "then the analysis treats p as consumed after the statement. In this situation, we issue a warning that p might leak, since the type-states do

not match."

While this shortcoming is generally a non-issue if we require that both conditional branches introduce or consume the same resources, the same workaround cannot be applied to functions that "may or may not" produce or consume resources. Such cases arise frequently in practice. Consider an example of the push() method of a limited capacity stack object:

```
1 // push(x) ⇒ true, x consumed and pushed.
2 // push(x) ⇒ false, x not consumed; stack full.
3 bool push(Item *x);
```

This very simple interface description defeats all type systems with linear resource tracking but without the ability to express pre- and post-conditions and the ability to associate these predicates to the transferring of ownership. For the interface of push() to be expressible, the transferring of resource ownership has to be linked to the return value of push() via a constraint system.

This was not achieved until Dengping Zhu and Hongwei Xi introduced stateful view into a programming language called Applied Type System ($\mathcal{ATS}$) [135] in 2005. In $\mathcal{ATS}$, dynamic terms of the language are separated into two realms: proof terms and program terms. Proof terms are erased after compilation, leaving only program terms. Proof terms are characterized by props, while program terms are characterized by types. Stateful views are linear proof terms. Both props and types are static terms based on Dependent Type, which allows a static term to reflect the values of the dynamic term it characterizes.

Through dependent type, $\mathcal{ATS}$ is able to guarantee the consistency of complex data structure, e.g. the coloring of linear red-black tree. The type system can relate pre- and post-condition of a function to stateful view using dependent type, which makes it possible to precisely express push() above. Another example of the expressive

power of dependent type and stateful view is reasoning about pointer arithmetics, which is accomplished by modeling an array like a singly linked list, except there is no "tail" pointer. An array of type a of positive integer length n at location l can be decomposed into an element of type a at location l, followed by an array of type a of length $n-1$ at location $l+1$ (at that time, it was assumed that locations are word aligned, so $l+1$ is the word after l in memory). Controlled sharing of stateful view in a local scope is added later through sharing modality [104].

1.6 Linear Object Ownership Semantics

Recent development of $\mathcal{ATS}$ [134] takes a more pragmatic view of the relationship between linear proof terms and linear proof variables. Linear proof terms can be seen as compile-time objects stored in compile-time linear proof variables. The linear proof variable is said to be the owner of the linear proof term stored in it. When assigning linear proof variables, the ownership is transferred from one variable to another, but the destination cannot already be an owner of a linear proof term. The source variable's ownership is relinquished. Furthermore, proof variables can be *passed by reference* to a function, in a similar way C++ can pass arguments by reference (e.g. compare pass by value `int x` to pass by reference `int& x`). This allows a function to preserve the linear proof term if the function does not consume the term. Linearity is enforced by ensuring no local linear proof variables retain ownership of any linear proof term when they go out of scope, and the proof variables passed by reference will still hold linear proof terms of the specified type. This transferring of ownership is called *linear object ownership semantics*.

Even with this pragmatic view, "linear types as are [sic] supported in $\mathcal{ATS}$ can be very demanding of a programmer's skill and are sometimes too complex for practical

uses[1]." This is evident after much hard work implementing various data structures such as Splay Tree [133] that could be used by a memory allocator to index free objects, a free-list based garbage collector [82, 83, 132], and a memory allocator implementing binary buddy system [38]. Linear types in $\mathcal{ATS}$ is also too limited in the case of concurrent data structures, which are used by modern memory allocators such as StreamFlow [100] to achieve non-blocking synchronization. The difficulty lies in the inability of the type system to reason about volatile invariants. This will be discussed more later in this dissertation. As a result, it was determined that a memory allocator would be implemented in C++ first, then rewritten in $\mathcal{ATS}$ as future work.

In order to use linear object ownership semantics in C++, this dissertation proposes linear pointer as a smart pointer class `linear_ptr<Tp>` similar to `auto_ptr<Tp>` given `Tp` as the type of the object referenced by the pointer (`Tp` could be the abbreviation for either template parameter or typename parameter), but the pointer will not automatically free memory when orphaned. Instead, linear pointer implements runtime checking in the overloaded operators that will enforce linearity of objects. There is an atomic version, `linear_atomic_ptr<Tp>`, to be used with non-blocking algorithms. This runtime checking can be turned off in release mode by changing a compile-time flag. The implementation will be described in detail later.

1.7 Memory Hierarchy

We briefly review the memory hierarchy of computer systems here in order to develop intuitions for hardware characteristics and how these characteristics affect software performance. Our account here is based on Peter Denning's locality principle [42].

[1]Hongwei's own opinion, communicated to me in a private correspondence.

Computer storage system covers a spectrum of storage technologies, each presenting a different cost and performance trade off. At one end of the spectrum, we have a large amount of cheap but slow memory, and at the other end, we have a small amount of expensive but fast memory. Computation is limited by memory speed, and since processors are faster than the fastest memory (this may change as new memory technologies such as memristor [111] are developed and made commercially available), it is only practical to carry out computation on the fastest memory available. To given an idea of the magnitude of storage speeds, the storage layers in a modern computer (circa 2011) typically consists of a magnetic hard disk, with 10ms access time; a dynamic random access memory (DRAM) as *main memory*, with 10ns access time; and cache memory made of static RAM, with 1ns access time. The processor itself executes a handful of instructions every 1ns [2].

The question is, at a given cost, how to effectively leverage all storage technologies in order to maximize the amount of stored information and maximize computation throughput. Software developers used to manually split a large program into overlays, which are portions of the program that can be loaded from slow into fast memory at a given time, but overlay planning is time consuming. A mechanism in use since the late 1950's is to let hardware abstract out the storage layers, presenting the software with a unified memory address space, with a *page* as a unit of memory (a page is typically 4096 bytes). The intent was to simulate a larger amount of expensive but fast memory backed by the cheap but slow storage, swapping content between the fast memory and the slow storage on demand. However, computer systems often run multiple programs concurrently, all sharing slices of processor time and memory as a scarce resource managed by the operating system. Without an understanding how

[2] 1ms (millisecond) is 10^{-3} (0.001) of a second; 1μs (microsecond) is 10^{-6} of a second; 1ns (nanosecond) is 10^{-9} of a second.

program's memory access pattern changes over time, a naive strategy for swapping can grind the memory stack for all programs to a halt, where the computer spends more time swapping than carrying out computation.

Denning observed that programs often exhibit a phased behavior, much like overlays, even when the program is not explicitly split into overlays: one phase of the program tends to access a common set of pages, and when it moves on to a different phase, it tends to access a different set of pages. He hypothesizes that the phased behavior is a result of the *divide and conquer* strategy for solving computational problems, where a large problem is broken down into smaller sub-problems. As the program moves from one sub-problem to another, the phase progression changes the demand of system resources. In 1968, he formalized the notion of working set [40], which is a set of pages accessed by a program over a window of time. Since then, operating systems have been relying on working set theory for providing *virtual memory* and dynamically balancing the scheduling of programs against their resource demands [41]. In practice, each page in memory has an "accessed bit" which indicates if the page has been accessed by the program. This bit is cleared periodically, and pages with the accessed bit set are the ones in the working set window since the last reset. Virtual memory performs better when the program concentrates its access in small number of pages, which tends to be the case if the program solves problems using the divide and conquer strategy. The program is said to exhibit good *locality of reference* when its memory accesses are concentrated.

A program typically obtains memory from the operating system in page-sized units, but the program's dynamic storage allocation needs are driven by data structure objects it creates during the course of a computation, and the object sizes are much smaller, often as small as 8–16 bytes. This is where memory allocator comes in. In this context, memory allocator is an agent that sits between the operating system

and the program, requesting memory from the operating system in batches, dividing memory into small chunks, and use the small chunks to fulfill objet creation needs of the program. Recall that the memory allocator also decides where in memory a new object is to be placed. A memory allocator may not be able to significantly improve a program's locality of reference by clever placement strategy, but it can always make locality worse if placement is spread out across many memory pages. It can also increase the program's working set if it has to touch a large number of pages in order to perform its memory management duties.

Proposition 1. *A memory allocator should minimize object placement spread when possible, in order to minimize the program's working set size.*

Proposition 2. *A memory allocator should adhere to the program's memory access pattern and not access memory outside of the program's working set.*

We briefly discussed cache memory at the beginning of this section, as one of the storage layers in the memory hierarchy of a modern computer. Cache memory is used to accelerate main memory due to the speed disparity between main memory and processor. Main memory access can take 20 instruction cycles or more. The speed disparity only worsens as processors become faster, as the demand of greater computing power rises. However, processor speed increase seems to have reached a limit (mostly due to heat and power consumption concerns), and now it is common to put multiple processors in a computer system in order to increase computing power because computations can now be carried out in parallel. This increased parallelism is facilitated to a program in the form of *threads* (as in "thread of computation"). Most multi-processor computer systems let all processors share the same main memory which they can all access, one at a time, via a shared memory bus; this is called *shared-memory* architecture. In the shared-memory architecture, main memory access

essentially serializes computation no matter how many processors there are. Main memory access is a bottleneck that reduces parallelism. The percentage of time a parallel program spends in non-parallel parts limits the maximum theoretical speed-up of the program; this is known as Amdahl's Law [5].

Fortunately, each processor often comes with its own local cache memory, which can alleviate main memory bottleneck by handling most memory accesses in the cache. When multiple processor caches happen to store the same memory locations, caches synchronize with each other through cache-coherence protocols on the memory bus. Data is transfered between main memory and cache memory in units of a cache line, which is typically 64 or 128 bytes. This is also the unit of synchronization for cache-coherence protocol. This is large enough to store several program objects. If the cache line is shared by multiple processors but the objects in the cache line are only used as private objects and not intended to be shared, the cache line still needs to be synchronized through the memory bus; this is called *false sharing*. This can degrade parallel computing performance by inducing unnecessary cache-coherence synchronization.

Proposition 3. *A memory allocator should try to allocate thread-local objects in thread-local memory, in order to prevent false sharing. Furthermore, a memory allocator should miminize accessing another thread's local memory while performing memory management duties.*

Some hardware tries to alleviate memory bus contention by pairing each processor with its own memory controller and main memory, but nevertheless allow all processors to access each other's main memory through an interconnect, which is slower than a processor accessing its own main memory. This is called *non-uniform memory access* (NUMA). To a program, processor-specific memory is still mapped to a shared

address space, but the memory access latency will differ depending on the distance between the processor and the desired memory bank. Ideally, a NUMA-aware program would drive a processor to access its own main memory as much as possible. This is achieved by a cooperation of the program and operating systems as follows:

1. A program's thread would set its own processor affinity to a specific processor. Otherwise the operating system takes liberty to schedule the execution of a thread on any processor that happens to be idle, at the expense of saturating the interconnect if the thread is migrated further away from its intended processor-specific memory.

2. A *first-touch* policy establishes affinity between the processor and the virtual memory mapping. On both uniform and non-uniform memory access systems, when a thread requests memory, the operating system normally delays mapping the physical memory to the virtual address space until the page is first accessed. The backing memory is only mapped to the virtual address upon this first access. Under first-touch policy, the backing main memory will be mapped from the physical memory that belongs to the processor that currently executes the thread that initiated the access. Alternatively, sometimes the operating system would simply fulfill backing memory by round-robin over the physical memory owned by all processors, if the program is not NUMA-aware. At least this load-balances memory use at the expense of increasing load on the interconnect.

Proposition 4. *Although a memory allocator should not change a thread's processor affinity, it should place objects on processor-specific memory in case if the program is designed to run on NUMA architecture.*

Further detail about the memory hierarchy can be found in a computer architecture text book, for example *Computer Systems—A Programmer's Perspective* by Bryant

and O'Hallaron [21]. Further detail about shared-memory architecture can be found in a parallel programming text book, for example *Parallel Computing Architecture, a Hardware/Software Approach* by Culler and Singh [35].

1.8 Advances in Garbage Collection

Although memory allocator has been widely researched and surveyed in literature, it is becoming increasingly difficult to justify research in this area due to the shift of software engineering practice, where more programs and programming languages are relying on garbage collectors. The first garbage collector was invented for the programming language LISP in 1960 [92, 93] out of necessity because of the prevalence of object sharing in a functional programming language, and the difficulty in ascertaining object ownership in the presence of sharing. When a program uses automatic memory management, it still acquires memory through *alloc*(), but *free*() is now optional. When *alloc*() runs out of memory, the garbage collector attempts a collection by tracing all objects that are reachable by the program and subsequently discarding unreachable ones. The concept of garbage collection is based on the observation that unreachable objects will never be used again by the program, so memory used for storing these objects may be safely reclaimed to store new objects. Automatic memory management greatly reduces the risk of unintentional memory leak (it is still possible to leak memory by either referencing objects from a global variable, or retaining objects on the call-stack of a run-away thread that is perpetually blocked but never terminates), and the convenience of not needing to scrutinize over object ownership greatly increases programmer productivity. Furthermore, since a garbage collector already traverses object pointers, it can move objects during collection to reduce fragmentation, such as *mark-and-compact* [112, 28] and *semispace copying*

garbage collector [51, 24].

The first garbage collectors had inherent deficiencies. Tracing all reachable objects is expensive when the number of reachable objects is high. Copying and compaction can reduce fragmentation, but they involve moving a large amount of memory content, which is also expensive. All this is performed inside the *alloc*() function call when program runs out of memory, so the program is forced to block until *alloc*() returns, during which time the program would become unresponsive. Also, tracing all reachable objects in memory involves touching memory pages not in the current working set and goes in contrary to locality principle. For this reason, garbage collection is known to cause virtual memory thrashing problems. Even though garbage collector has the ability to move objects to improve their locality of reference, the collection itself is a locality hazard.

As garbage collection methods mature, many of these deficiencies are addressed. *Generational garbage collection* [87, 122] addresses the problem that tracing, copying and compaction are expensive operations, by discriminating objects by their lifetime: long-lived objects do not need to be traced often, and short-lived objects do not survive long enough to be traced or copied. All new objects are first allocated from the *young* heap, which is garbage collected in a *minor* collection. A minor collection anticipates that most young objects become garbage, so only a small number of reachable objects are traced and copied. The surviving objects may be moved to the *aged* heap, which is collected less often in a *major* collection. Ideally, the frequency of the major collection should be in proportion to the lifetime of the aged objects in order to minimize the number of times the long-lived objects are traced. When to move surviving objects from the young heap to the aged heap is determined by a *tenure* policy. For the distinction between minor and major collection to be effective, the tenure policy must accurately discriminate long-lived objects from short-lived ones.

Failure to meet this requirement causes major collection to occur more frequently, diminishing the performance advantage of generational garbage collector. Modern implementation [96] uses an "ergonomic" policy that attempts to meet pause time and throughput goals while minimizing footprint [123].

Concurrent and incremental garbage collection [45, 46, 13] addresses the problem that program becomes unresponsive during collection, by allowing the mutator (the program) and the collector to run at the same time (concurrent) or interleaved (incremental). This requires careful synchronization between the mutator and the collector through memory barrier, so the program runs slower, but the collection cycle pause is greatly reduced. This allows the use of automatic memory management with real-time applications, where each task in the program has to be completed before a predictable deadline. Real-time applications traditionally shun garbage collection because the unpredictable pause time results in missed deadline.

More recently, to solve the problem that object tracing causes virtual memory thrashing, the Bookmarking Collector [68] cooperates with the operating system virtual memory facility to maintain a summary of the page about to be swapped out to the disk, and future collection would proceed using the summary instead of bringing the page into main memory.

For more detail about garbage collection, Paul Wilson did a survey of uniprocessor garbage collection techniques prior to 1992 [130], chronicling mark-and-sweep, mark-and-compact, copying, generational, and incremental garbage collectors. A similar account can also be found in Andrew Appel's *Modern Compiler Implementation in ML* [7, chapter 13]. Richard Jones dedicated an entire book to garbage collection in 1999 [75] with a follow up book in 2011 [74] that contains recent development in parallel, concurrent, and real-time garbage collection.

1.9 Argument Against Parallel Garbage Collection

Advances in garbage collection technology makes it seem that no performance problem introduced by an early garbage collector is unsolvable by its successor. However, there are several ways garbage collection *could* be detrimental to the scalability of a program running on a multi-processor computer. Since object reachability is a global property, collector and mutators all share memory and all need to have a consistent view. Furthermore, when the collector scans or copies reachable objects, the same objects may be accessed by the mutator in the mean time, which requires synchronization between collector and mutators. The synchronization causes communication overhead over the memory bus due to cache coherence protocol. It is particularly pronounced with concurrent garbage collection, where memory accesses for mutator and collector interleave in a fine grained manner. If we give up on the concurrent garbage collector, the alternative method employing the stop-the-world strategy with parallelized object traversal has the trade-off between fine-grained load balancing at the cost of more synchronization, or less synchronization with coarsed-grained load balancing. Fine-grained load balancing allows work to be more equally distributed among workers. On the other hand, coarsed-grained load balancing may cause unequal distribution of work, which reduces parallelism.

Although scalability limitations of garbage collected parallel programs are theoretically conceivable, quantifying their effects on run-time performance is difficult. The chief difficulty is that programming languages are often either entirely manually or entirely automatically memory managed. In 2005, Hertz and Berger compared performances of uniprocessor garbage collectors and manual memory allocators [67] by first monitoring allocation and object reclaimation activities of a garbage collector,

which are converted to *alloc*() and *free*() call traces. These call traces are then fed into a simulator that drives a manual memory allocator. However, the same methodology cannot be easily adapted for multi-processor memory management. The simulator will have a different memory access pattern and scalability characteristics than the original program. Furthermore, languages have different parallel computation models and sometimes different memory consistency models that affect how a program may be parallelized; consequently, some languages are simply more "scalable" than others by their intrinsics, without considering memory management, so the effect of automatic versus manual memory management in a parallel setting is hard to isolate.

Nonetheless, there are anecdotal evidences suggesting that, for a parallel program running on a multi-processor shared-memory computer, garbage collection can be detrimental to scalability.

- Anderson [6] compared two garbage collectors: JGC which is a generational "block-based, stop-the-world, mark-sweep-compact GC" where thread-local nurseries are assigned without thread-affinity, and TGC which assigns each thread a fixed nursery. His result shows that matrix multiplication on both JGC and non-concurrent TGC scales linearly and enjoys a speed up of P on P processors, up to $P \leq 23$. Matrix multiplication is memory bandwidth heavy, but all the memory access is confined to pre-allocated arrays of memory, and it does not place very much burden on the garbage collector. On the other hand, the experiment result of Barnes-Hut n-body simulation [14] shows that neither JGC nor TGC are scalable; they only enjoy a maximal speedup of 5 when running on 5 or more processors. The Barnes-Hut algorithm is known to scale nearly linearly to the number of processors [35, pp. 432]. The algorithm uses octtree to represent particles in a 3-dimensional space and is more demanding on memory

management. The same paper also compared non-concurrent TGC with two variants of concurrent TGC and found that, in the case of matrix multiplication, the concurrent variants do not scale when the number of processors is greater than 7.

- Siebert's JamaicaVM features a parallel and concurrent garbage collector [105] that shows some promise on scalability. A suite of benchmark shows that the garbage collector is $\Theta(P)$ scalable: for 8 processors, linear speedup between 4 and 8 is achieved. However, closer inspection reveals that each benchmark consists of eight independent instances of the same sequential program, running on eight threads. Even though the threads share the memory address space, they do not share data structures. Such embarrassingly parallel benchmark should be able to achieve ideal speedup, that is, by a factor that is the number of processors. However, the parallel, concurrent garbage collector still manages to hinder ideal speedup of many benchmarks but one.

- Gidra et al presented a study of garbage collector scalability on NUMA machines [54], comparing OpenJDK's scavenge collector, concurrent mark-sweep, and Garbage First collector, all of which are generational and parallel. Their result shows that none of them are scalable; as number of threads increase, so as the time spent in garbage collection. They attribute the scalability bottleneck to the lack of NUMA awareness and the inefficient load-balancing implementation that causes contention in the variable used to keep track of work-queue size and still fails to adequately balance the load.

- Appendix A presents a comparison between Cilk and JCilk, two parallel language extensions that have the same parallel computation model and implemented using the same work scheduling algorithm. Cilk is based on C and

manual memory management. JCilk is based on Java with garbage collection. The comparison shows that although it is possible to trade off reducing garbage collection frequency with increasing memory footprint, increase in garbage collection activity significantly reduces scalability.

In the end, there is no denial that automatic memory management is convenient for rapid prototyping as well as deployment of best-effort applications. But garbage collection impedes scalability due to the additional communication overhead it causes on a multi-processor computer, and this limitation affects all but the very trivial kind of parallel programs.

1.10 Organization of This Dissertation

Chapter 2 describes how to enforce linear ownership semantics in C++ and shows the erasure property of linear pointers. Chapter 3 shows how to implement data structures based on linear pointers. Chapter 4 presents the design and implementation of a parallel scalable allocator and shows that linear pointer can achieve comparable efficiency. Chapter 5 elaborates on using linear pointer with concurrent data structures, investigates the short-comings of formal system when it is used to reason about concurrent algorithms, and offers some suggestions for future improvements that might overcome these short-comings. Finaly, Chapter 6 concludes the dissertation.

Chapter 2

Linear Ownership Semantics

C++ is a language that expects program to manually manage dynamically allocated memory, and memory related errors such as memory leak, double free, and dangling pointer are commonplace. Linear object ownership semantics is a programming practice that prevents these memory related errors from happening. It is a policy about object pointer handling characterized by the invariant that each object has exactly one pointer to it. The program variable that holds the pointer is the owner. When object pointer is assigned from one variable to the next, the pointer in the original variable is erased, and the ownership is passed on to the assignee. Furthermore, any variable that retains ownership is prohibited from going out of scope. In C++, objects are explicitly introduced through the *constructor* and eliminated through the *destructor*. Constructor and destructor are defined in a *class* that describes the type of the object. It is easy to see that linear object ownership enforces that each object introduction must have a corresponding elimination, as follows:

- Memory resource is allocated by `operator new`, which invokes the object constructor to initialize the memory and returns the only pointer to the object.

- This pointer can be passed from one variable to another with the corresponding ownership transfer. Since variables that retain ownership cannot go out of scope, the pointer is never abandoned.

- The pointer is eventually passed to `operator delete`, which consumes the only pointer to the object, invokes the object destructor, and deallocates the underlying memory resource.

Traditionally in a linear programming language, linearity is enforced at compile time by the type system. Many such languages feature a type system with Curry-Howard correspondence to intuitionistic linear logic. The type system has built-in formalism for resource conservation of program terms. C++ does not have a linear type system, and handling object pointer in C++ is unsafe with regard to linearity, but C++ has evolved a concept called *Resource Acquisition Is Initialization* [110] that can be leveraged to ensure linearity.

As mentioned before, `operator new` introduces objects that are allocated dynamically on the heap, but this is not the only way objects are introduced. All program variables implicitly introduce objects as well. The compiler determines how to place the variable-identified objects. If the variable is global or static, the memory used by the object is reserved in the "bss" uninitialized data segment of the program. The number of global and static variables are known at link time; they occupy a fixed amount of memory, and they are unable to scale to the program's dynamic memory demands. On the other hand, if the variable is declared as a local variable inside a function, the memory used by the object is allocated on the call stack, which stores the activation frames of the function calls. The number of local variables grow as the depth of the function call-graph grows but in a strictly last-in first-out order.

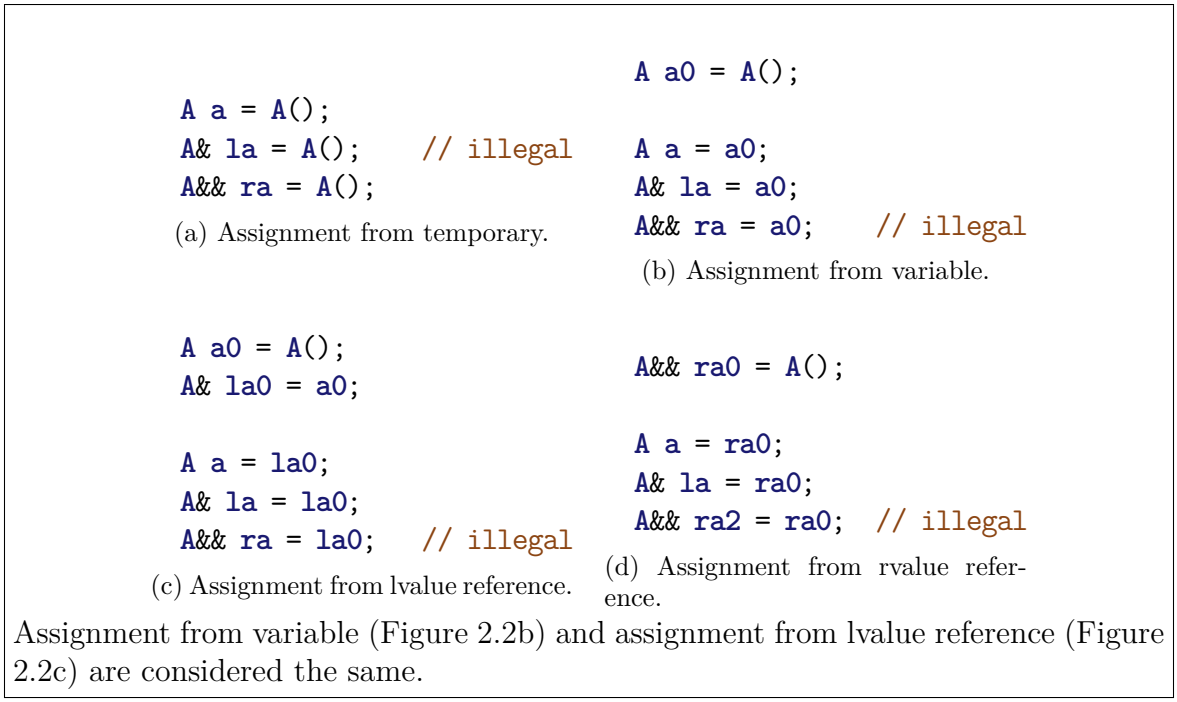
As part of the language's operational semantics, the C++ compiler ensures that the constructor is called whenever the program's control flow enters the scope of the variable, and that the destructor is called whenever the program's control flow leaves the scope of the variable. This is guaranteed even in the face of an exception which causes the control flow to escape the current scope, potentially unwinding the call-stack by an arbitrary number of frames until an exception handler is located. This association of control flow, variable scope, and object construction and deconstruction is the key idea behind resource acquisition is initialization.

It is worthwhile to note that variable-identified objects, which are automatically memory-managed by the compiler, are already linear in the sense that each variable is the sole owner of the underlying object, but this ownership cannot be transferred. There are three ways to work around this problem in C++: to pass an object by pointer, pass by *lvalue* reference, or pass by value. Passing an object by pointer makes the pointer indistinguishable from dynamically allocated objects, and it is an error to free objects placed by the compiler in the data segment or on the stack. Passing an object by reference creates a variable that is an alias to an existing object. It is effective for passing objects from the caller to the callee, but it is an error for a callee to pass an object back to the caller by reference because of object lifetime constraint. By the time the caller receives the reference of the object, which is placed on the activation frame of the callee, the object itself is already destroyed. Passing object by value makes a copy of the object, and it is the only well-defined way for a function to return an object back to the caller. The function would copy out the object to the caller before leaving its activation frame.

Object copying is made possible in C++ by the *const lvalue reference* constructor, also know as the copy constructor, which allows an object to be constructed as a copy of another object. However, a program could spend a significant amount of time

copying objects if ownership transfer happens frequently. Even so, programming using only stack-allocated object is still too restrictive: across a function call, the callee could only return one object back to the caller, and the caller must allocate space for the returned object in advance. This makes it difficult to implement a function whose output size is greater than the input size. In practice, many objects hold pointers to dynamically allocated objects which is managed by the memory allocator, and linear ownership of these pointers are not enforced. In addition, in order to satisfy the semantics of the copy constructor, simply copying the internal pointers is not enough; dynamically allocated objects referenced by these pointers must be deep-copied in the copy consturctor as well. This is a common source for performance problems of containers such as `std::vector<Tp>` and `std::string`, particularly when a temporary object is created for the sole purpose of initializing a more permanent object. In some cases such as Return Value Optimization, the copying could be elided by eliminating the temporary object.

C++11 introduced the *rvalue reference* constructor for move semantics, which allows a newly constructed object to pilfer internal resources that belong to a temporary object, and the temporary object will be in an undefined state. The C++11 standard has complex rules to determine whether an expression can be used as an rvalue reference (see Figure 2.1 and Figure 2.2). For example, a variable cannot be used as rvalue reference because the program might accidentally use the pilfered source object which is in the undefined state, but an rvalue reference returned from a function has no such problem. However, it is still the responsibility of the source object's destructor to recognize this undefined state so that it will not release the internal resources again, which will result in dangling pointer in the new move-constructed object. If anything, move semantics is only sound if linear object ownership is preserved. It is commonly misunderstood that rvalue reference signifies the intent to pilfer resource



(a) Assignment from temporary.

(b) Assignment from variable.

(c) Assignment from lvalue reference.

(d) Assignment from rvalue reference.

Assignment from variable (Figure 2.2b) and assignment from lvalue reference (Figure 2.2c) are considered the same.

Figure 2.1: Temporary, lvalue reference, and rvalue reference assignments.

from the other object, but it does not. It merely allows function return values as well as temporaries to be destructively updated, which was not previously possible with lvalue reference.

Pedantically speaking, C++98 [71, §12.1.10] treats constructor for class `X` accepting the argument `const X&` as well simply `X&` as copy constructor, but here we make a distinction, calling the latter a move constructor instead. In C++11 [72, §12.8.3], a move constructor is a constructor taking `X&&` as the argument. Our terminology here is chosen to reflect the intention; whenever we have a non-const lvalue reference constructor, the intention is to implement move semantics, so it is called a move constructor. Rvalue reference is not used here.

```
void Fa(A a);          A Ca();          A a = A();
void Fla(A& la);       A& Cla();        A& la = a;
void Fra(A&& ra);      A&& Cra();       A&& ra = A();
```

```
Fa(A());
Fla(A());   // illegal
Fra(A());
```

(a) Calling with temporary argument.

```
Fa(Ca());
Fla(Ca());  // illegal
Fra(Ca());
```

(b) Calling with temporary return value.

```
Fa(la);
Fla(la);
Fra(la);    // illegal
```

(c) Calling with lvalue reference.

```
Fa(Cla());
Fla(Cla());
Fra(Cla());  // illegal
```

(d) Calling with lvalue reference return value.

```
Fa(ra);
Fla(ra);
Fra(ra);    // illegal
```

(e) Calling with rvalue reference.

```
Fa(Cra());
Fla(Cra());  // illegal
Fra(Cra());
```

(f) Calling with rvalue reference return value.

- Function call with temporary values (Figure 2.3a) and temporary return values (Figure 2.3b) are consistent with assignment with temporary value (Figure 2.2a).
- Function call with lvalue reference (Figure 2.3c) and lvalue reference return values (Figure 2.3d) are consistent with assignment with lvalue reference (Figure 2.2b).
- But in the case of rvalue reference, function call (Figure 2.3e) is the same as assignment (Figure 2.2d), but rvalue reference is treated differently when it is the return value of a function (Figure 2.3f).

Figure 2.2: Temporary, lvalue reference, and rvalue reference for function argument and return value.

2.1 Object as a Smart Pointer

In C++, an object can be treated as a pointer if it defines operators for dereferencing (`operator*`) and for member access (`operator->`). This is made possible through operator overloading. Formally, any object whose class defines the pointer interface in Listing 2.1 is treated like a native pointer in C++. The pointer interface extends the built-in comparable interface as decribed in Listing 2.2. Note that in the pointer interface, the listing defines the accessors `is_nil()` and `is_invalid()` as well as the mutators `get()`, `release()`, and `reset()`. These methods are not required by C++ for pointers, but their purpose will be explained later.

A pointer object like this is supposed to be backing a variable and not itself dynamically allocated; when that is the case, the object's constructor and destructor are essentially notifications of the program's control flow entering and leaving the scope of the variable. This is called a *smart pointer*. Depending on whether the pointer implements copying semantics or move semantics, it can be classified as implementing a `copy_assignable`, `lvalue_movable`, or `rvalue_movable` interface, as seen in Figure 2.3. The plain old data pointer `Tp*` can be seen as an example of a copy-assignable pointer.

The first smart pointer `auto_ptr<Tp>` was introduced by Gregory Colvin in 1994 [30] and incorporated into the Standard Template Lirary in C++98 [71, §20.4.5]. It is a singleton container that holds a pointer to an object allocated by `operator new`. It could also hold a `NULL` pointer. Its destructor releases the object using `operator delete` if the pointer is not `NULL`. When assigning resource to a pointer that already has ownership to another resource, the original resource is automatically released as well.

Rather than implementing a copy-assignable interface which does not modify the

Listing 2.1: Pointer class interface

```
// Pointer interface synopsis.  Any class that implements this
// interface will appear to C++ as if it's a native object pointer.
//
template<typename Tp>
class pointer : public builtin_comparable<pointer<Tp> > {
 public:
  // Constructor
  explicit pointer(Tp *) throw();  // Value constructor
  pointer(pointer& that) throw();  // Move constructor

  // Assignment
  pointer& operator=(Tp *) throw();  // Value assignment
  pointer& operator=(pointer& that) throw();  // Move assignment

  // Conversion
  operator bool() const volatile throw();  // C++98 section 4.12
  Tp& operator*() const throw();
  Tp* operator->() const throw();

  // Accessors
  bool is_nil() const volatile throw();
  bool is_invalid() const volatile throw();

  // Mutators
  Tp *get() throw();
  Tp *release() throw();
  void reset(Tp *) throw();
};
```

Listing 2.2: Comparable class interface

```
template<typename Self>
class builtin_comparable {
 public:
  bool operator==(const Self&) const throw();
  bool operator!=(const Self&) const throw();
  bool operator<(const Self&) const throw();
  bool operator<=(const Self&) const throw();
  bool operator>(const Self&) const throw();
  bool operator>=(const Self&) const throw();
};
```

```
class copy_assignable {
 public:
  copy_assignable(const copy_assignable&) throw();
  copy_assignable& operator=(const copy_assignable&) throw();
};
```

(a) Copy-assignable interface.

```
class lvalue_movable {
 public:
  lvalue_movable(lvalue_movable&) throw();
  lvalue_movable& operator=(lvalue_movable&) throw();
};
```

(b) Lvalue movable interface.

```
class rvalue_movable {
 public:
  rvalue_movable(rvalue_movable&&) throw();
  rvalue_movable& operator=(rvalue_movable&&) throw();
};
```

(c) Rvalue movable interface

Figure 2.3: Pointer semantic classifications.

source, `auto_ptr<>` implements an lvalue movable interface that transfers object ownership from the source. An `auto_ptr<>` can be passed as an argument to a function, but returning an `auto_ptr<>` from a function requires a complex workaround in the implementation because function return values are not lvalues. For this reason, C++11 superseded `auto_ptr<>` by `unique_ptr<>` [72, §20.7.1] which implemented a rvalue movable interface with vastly simplified implementation. It also allows the pointer to use a different delete policy than `operator delete`.

Although smart pointers could be used to address resource leaks, it is well-known that an auto-deleting smart pointer have correctness problems when used with algorithms that do not know about auto-deletion, and programmers have been advised to avoid using smart pointers with STL containers. For example, an in-place array-based quick-sort algorithm could be implemented as follows.

Listing 2.3: An in-place, array-based implementation of the Quick Sort algorithm.

```
1  template<typename Tp>
2  void swap(Tp& a, Tp& b) {
3    Tp tmp = a;
4    a = b;
5    b = tmp;
6  }
7
8  // Based on the Quick Sort described by [33, pp.146]
9  //
10 // "first" and "last" are indices into the array "xs", and the
11 // indices are inclusive.
12 template<typename Tp>
13 void quicksort(Tp xs[], size_t first, size_t last) {
14   if (first >= last)
15     return;  // Nothing to sort.
16
17   // Partition
18   Tp pivot = xs[last];  // Non-linear!
19   size_t i = first;
20   for (size_t j = first; j < last; j++) {
21     if (xs[j] <= pivot)
```

```
22        swap(xs[i++], xs[j]);
23    }
24    swap(xs[i], xs[last]);
25
26    // Recursively sort subarrays.
27    if (i > 0) quicksort(xs, first, i - 1);
28    quicksort(xs, i + 1, last);
29  }
```

The pivot is read out (ownership transferred) to a temporary variable but never written back to the array. The pivot is subsequently abandoned. If the array is an array of smart pointers with auto-deletion, then all array items would mysteriously disappear after sorting is done because each item is used as the pivot at one point. The problem is corrected if line 18 is changed as follows:

```
18    Tp& pivot = xs[last];  // Tp& instead of Tp
```

Then it would work. We are now *borrowing* an alias of the pivot, but we do not steal its ownership.

Since the observation of the pitfall when using `auto_ptr<>` with STL algorithms, its successors have tried to avoid this pitfall by restricting operator overloading so that certain uses are no longer allowed.

As a restricted form of `auto_ptr<>`, Boost introduced `scoped_ptr<>` [2] which behaves more like a scope-bound object except the object is heap allocated. A `scoped_ptr<>` cannot be copied or assigned, but it does allow a `swap()` method to transfer ownership. The restriction prohibits its use in non `scoped_ptr<>` aware code which makes it safer, but its auto-deletion still imposes problem in a data structure. Imagine a binary search tree rotation that forgets to move one of its subtrees. Rather than flagging this as an error, `scoped_ptr<>` would truncate the abandoned subtree.

It's worthwhile to note that `unique_ptr<>` does not actually implement the lvalue

movable interface. Since C++ does not implicitly promote an lvalue reference to an rvalue reference when looking up operators, this prohibits `unique_ptr<>` from being used like this:

```
1 typedef std::unique_ptr<int> ptr_type;
2 ptr_type p(new int(1));
3 ptr_type q = p;  // not allowed
4 delete q.release();
```

This prevents `unique_ptr<>` from being used by the quick-sort algorithm above. However, a data structure implementation using `unique_ptr<>` still suffers the truncation problem.

A smart pointer not susceptible to the auto-deletion pitfall is `shared_ptr<>` which is reference counted and implements the copy-assignable interface. The constructor increases the reference count, and the destructor decreases the count. When the count reaches zero, the object is deleted. It was introduced by Boost [2] and now incorporated into C++11 STL [72, §20.7.2.2]. However, it is well-known that reference counting carries a significant run-time overhead.

2.2 Linear Pointer

In this chapter, we introduce a novel smart pointer discipline called linear pointer to ensure linearity of resources in a program. It is a run-time based approach that allows memory leaks to be detected immediately when it happens, so the error can be debugged in the context where the problem lies. Unlike the smart pointers mentioned in the previous section, linear pointer has the distinguished property of erasure so that when ownership tracking is turned off, the program behaves identically to that using a raw pointer and suffers no performance penalty. Unlike compile-time ownership tracking, linear pointer is built on top of the semantics of C++, so it does not suffer

semantic gap like type systems or annotations where the expressive power of the static analysis falls short in the expressive power of the language. Linear pointer can be implemented for C++98 or C++11, which ensures that the technique is portable among any standard compliant C++ compilers.

Linear pointer allows flexible transfer of ownership through move constructor and move assignment operator, but linear pointer never automatically deletes the underlying resource. Instead, the destructor contains an assertion that it holds no resource when the control flow leaves the scope of the linear pointer. In a debug build when assertion is turned on, the program is simply aborted at the moment the resource is leaked, as well as when the program tries to dereference a pointer that has no ownership of any resource.

Two special pointer values `nil` and `invalid` are reserved. The `nil` value is treated as a valid linear resource, except that it can be arbitrarily introduced and eliminated. For the sake of convenience, a linear pointer is initialized with the `nil` value if a resource is not explicitly given to its constructor. On the other hand, the `invalid` value represents a dangling pointer and indicates that the ownership has been transferred away. This is used by the move constructor and the move assignment operator to fill the other linear pointer after transferring object ownership away from it. The invalidation only happens when ownership tracking is turned on.

The distinction between `nil` and `invalid` is necessary in order to satisfy the erasure property, namely that if a program behaves correctly with ownership tracking, it must also behave correctly without.

Rather than relying on the dereference operator to detect dereferencing `nil` or `invalid` pointer, we typically assign `nil` to be `0lu` and `invalid` to be `~0lu` so that we could leverage the hardware's memory protection to detect null and dangling pointer access. The concrete representation of `nil` and `invalid` are defined in the

Listing 2.4: Value traits for linear resources.

```
template<
  typename Tp,
  unsigned long int Nil = 0lu,
  unsigned long int Invalid = ~0lu << 12>
struct value_traits {
  typedef Tp value_type;
  static const Tp nil;          // a nil value is for the empty resource.
  static const Tp invalid;      // an invalid value is for uninitialized use.
};

template<typename Tp, unsigned long int Nil, unsigned long int Invalid>
const Tp value_traits<Tp, Nil, Invalid>::nil = (Tp) Nil;

template<typename Tp, unsigned long int Nil, unsigned long int Invalid>
const Tp value_traits<Tp, Nil, Invalid>::invalid = (Tp) Invalid;
```

`value_traits<>` type in Listing 2.4.

2.3 Linear Base Class

Before implementing the linear pointer, we first describe the linear base class which governs the policies for linear ownership semantics. The constructors and assignments cause the origin to lose ownership of the resource, by setting its value to `invalid`. The destructor asserts its value to be `nil` or `invalid`. The C preprocessor macro `NDEBUG` (no-debug, to disable debugging) turns the base class into a regular value, i.e. does not invalidate origin when the ownership is transferred, nor does it assert value to be `nil` or `invalid` upon destruction. This allows linearity checking to be turned off so that optimized build of the program incurs no performance penalty. In order for this to work, we must guarantee that the program's behavior is identical with or without linearity enforcement. This is where the `invalid` value comes into

play. If we transfer ownership away from the origin by setting origin to `nil`, which has a special meaning in a data structure as a terminal node, then the program may erroneously rely on this behavior. But in `NDEBUG`, the origin will retain its old value and will not be changed. The correct behavior for a linear program at this point is to reinitialize the origin to `nil` explicitly. The `invalid` value will cause the incorrect program to abort with linearity checking. A correct program will continue to behave correctly in `NDEBUG` mode.

Linear base class uses `with_value<Tp>` for value storage, which provides a `value()` accessor method that returns an lvalue reference to some storage of type `Tp` but has no linear ownership semantics. The parent class `value_comparable_impl<Self>` implements comparison operators on `value()`, and `serializable<Self>` implements `operator<<` for insertion into an `std::ostream`.

The `Self` template argument refers to the most derived type of the `linear_base` class, such as `linear_ptr<>` and `linear_chroma_ptr<>` which will be introduced below. The `Self` type allows `linear_base` to use the most derived type of itself as an abstract, incomplete type in the class declaration, and as a normal type in method definition; this is possible due to the deferred type checking of template methods in C++. The technique is part of a template programming pattern observed by James Coplien as "curiously recurring" [32], as in many people before him have independently invented the technique. The technique did not have a name, but it is now simply known as the "curiously recurring template pattern." The pattern allows type-based "static virtual method" dispatched in compile time without using virtual table.

Listing 2.5: Linear base value.

```
1 template<class Self, class Traits /* implements value_traits<?> */>
2 class linear_base
3   : protected with_value<typename Traits::value_type>,
```

```
4       public value_comparable_impl<Self>,
5       public serializable<Self> {
```

In the following section, `SELF_TYPE_DECL(Self)` is a macro that provides the `self()` method to access the `this` pointer as the `Self` type. The serialize method allows the `serializable<Self>` parent class to be able to output the linear value to an `std::ostream`.

```
6   private:
7    SELF_TYPE_DECL(Self);
8    friend class value_comparable_impl<Self>;
9    friend class serializable<Self>;
10
11   void serialize(std::ostream& os) const {
12     os << this->value();
13   }
14
15  protected:
16   typedef typename Traits::value_type value_type;
```

The following section allows a linear value to be returned from a function as a return value. This is necessary because linear values do not have a copy constructor. The copy constructor of `linear_base` would take `const linear_base&` as the argument, which means that the constructor could not modify the origin, and hence cannot transfer ownership. Instead, it has a move constructor that takes `linear_base&` as the argument. The problem is that a function return value is a temporary, which is inherently `const` and cannot be used as a non-const reference. To workaround this problem, we allow the compiler to implicitly cast the linear value to a protected `ref` type, which is a temporary container that can be arbitrarily copied only by the compiler, and we allow a linear value to be reconstructed from the `ref` type. This same technique is used by `auto_ptr<Tp>`. An alternative implementation using rvalue reference for move constructor, which would take an argument of type `linear_base&&`, would make the workaround unnecessary.

```
17  protected:
18   struct ref {
19     explicit ref(value_type value) throw() : value_(value) {}
20     value_type value_;
21   };
22
23  public:
24   // Conversion.
25
26   // Cast operator for implicit reference conversion.
27   //
28   operator ref() throw() {
29     return ref(this->get());
30   }
```

The following section defines the constructor, destructor, and move assignment, which describe the resource handling policies of linear ownership semantics. The constructor initializes the value to `invalid` by default unless a resource is given (though this policy will be overridden by linear pointer). The destructor ensures that the value is either `nil` or `invalid` so that no resource is leaked. The assignment operator delegates linear transfer policy to the `get()` method below. Note that any linear classes based on `linear_base` must define their own value constructor, reference constructor (for `ref` type), move constructor, value assignment, as well as reference (`ref`) assignment, due to the way constructor and operator resolution works in C++.

```
31  protected:
32   // Value constructor, destructor, and move assignment.
33
34   explicit linear_base(const value_type& value = Traits::invalid) throw()
35     : with_value<value_type>(value) {}
36
37   ~linear_base() throw() {
38     assert(self()->is_invalid() || self()->is_nil());  // #if !NDEBUG
39   }
40
41   linear_base& operator=(linear_base& that) throw() {
42     reset(that.get());
43     return *this;
```

```
44   }
```

It is worth noting that, due to the lack of a copy constructor, any class that is derived from `linear_base` or contains `linear_base` or its derivatives as a member will not be copyable. When a copy constructor is not explicitly implemented for a class, C++ compiler can automatically synthesize copy constructors, but the synthesis will fail when it tries to find a copy constructor for `linear_base`. As a corollary, all classes having a member that is a linear pointer, which derives from `linear_base`, are automatically non-copyable.

The following section defines the accessors and mutators. The accessors `operator bool()`, `is_nil()`, and `is_invalid()`. These accessors do not modify the state of the linear value in any way. On the other hand, the mutators `get()`, `release()`, and `reset()` modify the linear value in a manner that ensures resource conservation. The `get()` method is used by move constructor, reference conversion, and assignment operator to initiate resource transfer.

```
45  public:
46   // Cast operator for boolean interpretation.
47   //
48   operator bool() const volatile throw() {
49     assert(!self()->is_invalid());
50     return !self()->is_nil();
51   }
52
53   // Accessor and mutator.
54
55   bool is_nil() const volatile throw() {
56     return this->value() == Traits::nil;
57   }
58
59   bool is_invalid() const volatile throw() {
60     return this->value() == Traits::invalid;
61   }
62
63   // Gets the value and invalidates ownership.  In NDEBUG mode, does
```

```
64    // not cause the source to reset.
65    //
66    value_type get() throw() {
67  #if NDEBUG
68      return this->value();
69  #else
70      assert(!self()->is_invalid());
71      value_type tmp = this->value();
72      this->value() = Traits::invalid;
73      return tmp;
74  #endif
75    }
76
77    // Releases ownership and reset value to nil.  Returns the released
78    // value.
79    //
80    value_type release() throw() {
81      assert(!self()->is_invalid());
82      value_type tmp = this->value();
83      this->value() = Traits::nil;
84      return tmp;
85    }
86
87    // Assigns a new value to this object, which must not currently hold
88    // ownership.
89    //
90    void reset(value_type value = Traits::nil) throw() {
91      assert(self()->is_nil() || self()->is_invalid());  // #if !NDEBUG
92      this->value() = value;
93    }
```

And this concludes the `linear_base` class.

```
94  };
```

2.4 Linear Pointer and Linear Chromatic Pointer

The implementation of `linear_ptr` is presented in Listing 2.6, which is a pointer to an object with exactly one owner, using `linear_base` to ensure linear ownership,

but with facilities for pointer dereferencing. The concrete representation of `nil` and `invalid` pointer values are assumed to be pointing to an unmapped memory region, which allows the implementation to rely on the hardware to catch pointer dereference for missing ownership. By default, these are `0lu` and `~0lu << 12` respectively. These are chosen because most operating systems do not map the first and last pages of the address space[1]. By setting invalid to `~0lu << 12` rather than `~0lu`, it is easier to tell when the program attempts to access an offset of the invalid pointer. The invalid pointer address would look like `0xff`...`f`*NNN* where *NNN* is the offset, which is distinguishable from accessing an offset of the nil pointer which would look like `0x00`...`0`*NNN*.

It is also possible to catch bad pointer dereferencing in software by inserting the appropriate checks into `operator*` and `operator->`, in which case the assumption that `nil` and `invalid` belong to unmapped memory region is unnecessary. The reliance on hardware is an implementation decision.

Listing 2.6: Linear pointer.

```
1  template<typename Tp, class Traits = value_traits<Tp *> >
2  class linear_ptr /* implements pointer<Tp> */
3    : public linear_base<linear_ptr<Tp, Traits>, Traits> {
4   private:
5    typedef linear_base<linear_ptr<Tp, Traits>, Traits> super_type;
6
7   protected:
8    Tp *ptr() const volatile throw() { return this->value(); }
9
10  public:
11   typedef Tp element_type;
12
13   // Value constructor, reference constructor, move constructor, value
```

[1]For example, on a 32-bit machine running Mac OS X, the last 80KB except for the last page of the address space is used as the COMMPAGE which stores user-accessible routines for machine specific functions. On a 64-bit machine, the COMMPAGE is located near the limit of the 48-bit address line at `0x00007fffffe00000`. These are defined by the constants `_COMM_PAGE32_BASE_ADDRESS` and `_COMM_PAGE64_BASE_ADDRESS` respectively.

```
14    // assignment, reference assignment.
15
16    explicit linear_ptr(Tp *p = Traits::nil) throw()
17      : super_type(p) {}
18
19    linear_ptr(typename super_type::ref r) throw()
20      : super_type(r.value_) {}
21
22    linear_ptr(linear_ptr& that) throw()
23      : super_type(that.get()) {}
24
25    linear_ptr& operator=(Tp *p) throw() {
26      this->reset(p); return *this;
27    }
28
29    linear_ptr& operator=(linear_ptr& that) throw() {
30      super_type::operator=(that); return *this;
31    }
32
33    linear_ptr& operator=(typename super_type::ref r) throw() {
34      this->reset(r.value_); return *this;
35    }
36
37    // Accessor and mutator.
38
39    // Allows pointer dereference.
40    //
41    Tp& operator*() const throw() { return *this->value(); }
42    Tp* operator->() const throw() { return this->value(); }
43  };
```

It is also possible to implement a linear *chromatic* pointer, which is similar to a linear pointer but the least significant n bits are used to store an integer color. This leverages the fact that many objects are aligned to 2^n memory address. For example, if the object memory addresses are aligned to multiples of 4, then colors can be any value from 0 through 3. The linear chromatic pointer can be used to implement data structure such as the red-black tree [60]. Unlike `linear_ptr`, the type `uintptr_t` is used as the underlying value instead of `Tp *`, and it is required that `nil` is `0lu` so

that the `nil` pointer can be colored properly.

Listing 2.7: Linear chromatic pointer.

```
1  template<typename Tp, typename Cp, unsigned int bits = 2u>
2  class linear_chroma_ptr /* implements pointer<Tp> */
3    : public linear_base<linear_chroma_ptr<Tp, Cp, bits>,
4                         value_traits<uintptr_t> > {
5   protected:
6    typedef value_traits<uintptr_t> traits_type;
7
8   private:
9    typedef linear_base<linear_chroma_ptr<Tp, Cp, bits>, traits_type> super_type;
10
11   friend class serializable<linear_chroma_ptr<Tp, Cp, bits> >;
12
13   void serialize(std::ostream& os) const {
14     os << this->ptr() << '.' << this->color();
15   }
```

The following utility functions convert an `uintptr_t` to a pointer and color, and vice versa.

```
16  protected:
17   static const uintptr_t mask = (1 << bits) - 1;
18
19   static Tp *ptr_of_int(uintptr_t x) throw() {
20     return reinterpret_cast<Tp *>(x & ~mask);
21   }
22
23   // Note that Cp() is the default value of type Cp, e.g. for int it is 0.
24
25   static uintptr_t int_of_ptr(Tp *p, Cp c = Cp()) throw() {
26     uintptr_t x = reinterpret_cast<uintptr_t>(p);
27     assert((x & mask) == 0);
28     assert((c & ~mask) == 0);
29     return x | c;
30   }
31
32   // Protected accessor and mutator.
33
34   Tp *ptr() const volatile throw() { return ptr_of_int(this->value()); }
35   void reset_uint(uintptr_t x) throw() { super_type::reset(x); }
```

These are the constructors and assignment operators.

```
36  public:
37   typedef Tp element_type;
38   typedef Cp color_type;
39
40   // Value constructor, reference constructor, move constructor, value
41   // assignment, reference assignment.
42
43   explicit linear_chroma_ptr(Tp *p = 0, Cp c = Cp()) throw()
44     : super_type(int_of_ptr(p, c)) {}
45
46   linear_chroma_ptr(typename super_type::ref r) throw()
47     : super_type(r.value_) {}
48
49   linear_chroma_ptr(linear_chroma_ptr& that) throw()
50     : super_type(that.super_type::get()) {}
51
52   linear_chroma_ptr& operator=(Tp *p) throw() {
53     super_type::reset(int_of_ptr(p)); return *this;
54   }
55
56   linear_chroma_ptr& operator=(linear_chroma_ptr& that) throw() {
57     super_type::operator=(that); return *this;
58   }
59
60   linear_chroma_ptr& operator=(typename super_type::ref r) throw() {
61     super_type::reset(r.value_); return *this;
62   }
```

The accessors and mutators.

```
63   // Public pointer-based accessor and mutator.
64
65   Cp color() const throw() { return static_cast<Cp>(this->value() & mask); }
66   void set_color(Cp c) throw() { this->value() = int_of_ptr(ptr(), c); }
67
68   // The following parts of a linear_chroma_ptr are semantically
69   // different than that of linear_ptr due to coloring.
70
71   Tp& operator*() const throw() { return *ptr(); }
72   Tp* operator->() const throw() { return ptr(); }
73
74   bool is_nil() const volatile throw() {
```

```
75      return (this->value() & ~mask) == traits_type::nil;
76    }
77
78    Tp* get() throw() { return ptr_of_int(super_type::get()); }
79    Tp* release() throw() { return ptr_of_int(super_type::release()); }
80
81    void reset(Tp *p = NULL, Cp c = Cp()) throw() {
82      super_type::reset(int_of_ptr(p, c));
83    }
```

And this concludes the definition of linear chromatic pointer.

```
84  };
```

2.5 Erasure

Linear pointer has the property that if a program behaves correctly with resource ownership tracking, then it will also behave correctly without the tracking. This property allows `linear_ptr<>` to be used as a debugging tool that can be disabled in a release build of a program which incurs no run-time overhead.

Definition 5 (Erasure)**.** An erasure occurs when a program is compiled with the `NDEBUG` preprocessor constant.

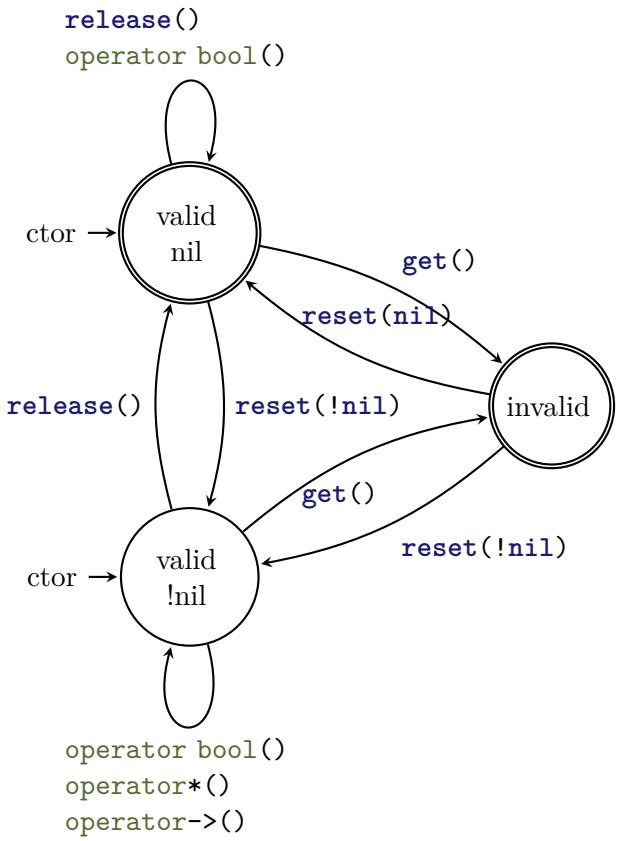
Example 6 (Assertion Erasure)**.** An example of erasure is the `assert()` statement which, in the absence of `NDEBUG`, evaluates the expression and aborts the program if the expression evaluates to false. When `NDEBUG` is set, `assert()` is a no-op, and the expression is not evaluated.

Example (Linear Pointer Erasure)**.** When a linear pointer is erased, it becomes a plain old pointer.

- The value constructor is already the same as that of a plain old pointer.

- The destructor does not check for leaks and becomes a no-op.
- `operator bool()` does not check for invalidated pointer.
- `get()` simply returns the underlying raw pointer.
- `reset()` does not check for leaks and simply sets the raw pointer to its argument, or `nil` if an argument is not given.
- As the result of `get()` and `reset()` erasure, the move constructors become copy constructors, and the assignment operators become copying assignments, which are the same for plain old pointers.

Definition 7 (Correct Program)**.** A correct program observes the following linear pointer finite state machine transition. The accept states indicate the allowed states when the destructor is called.



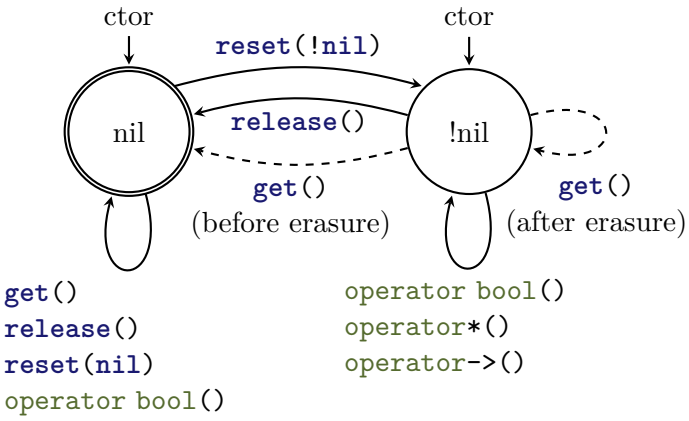
Theorem 8 (Erasure Preserves Correctness)**.** *Before erasure, if a program is correct with respect to linear pointers, then the erased program is also correct.*

Proof. First it is easy to verify that the assertion statements enforce the state transition shown in the definition, and if a program observes the state transition, then the erasure of the assertions will not affect the state transitions.

The only interesting case is `get()`, where after erasure, the invalid state is now implied and not reflected on the raw pointer. Since a correct program can only transition from invalid to either valid nil or valid non-nil states through `reset()`, the implied invalid state is overwritten with an explicit valid state, hence the erasure preserves correctness. □

The proof illustrates why it is necessary to distinguish `nil` and `invalid` as two

different states. If `get()` invalidates a linear pointer by setting the value to `nil`, then we have the following state transition where erasure no longer preserves correctness.



In the diagram, `get()` before erasure and `get()` after erasure put the linear pointer into two different states, and that causes the program's behavior to diverge.

2.6 Idioms for Manipulating Linear Pointers

Linear pointer classes are designed to appear just like native pointers at a type system level, but they have additional facilities that are missing in native pointers. For example, given a pointer `Tp *p`, the `is_nil()` method is a common idiom that corresponds to the expression `p == NULL`, but one cannot write `p.is_nil()` for a native pointer. On the other hand, given a linear pointer `linear_ptr<Tp> q`, one cannot write `q == NULL` either because linear pointer can only be compared with a linear pointer, not with a native pointer. This gives rise to `pointer_traits<>`, which hides the difference between linear and native pointers so that a program can be written once and work with both kinds of pointers.

The pointer traits for a linear pointer (or any smart pointer implementing the pointer interface in Listing 2.1) simply forward the operations to the accessor and

mutator methods. The `reset()` trait is polymorphic in order to accommodate linear atomic pointer which will be described in Chapter 5.

Listing 2.8: Pointer traits for pointer interface.

```
template<typename Ptr /* implements pointer<?> */>
class pointer_traits {
 public:
  typedef Ptr pointer;
  typedef typename Ptr::element_type element_type;

  static bool is_nil(const pointer& ptr) throw() { return ptr.is_nil(); }
  static bool is_invalid(const pointer& ptr) throw() {
    return ptr.is_invalid();
  }
  static element_type *get(pointer& ptr) throw() { return ptr.get(); }
  static element_type *release(pointer& ptr) throw() { return ptr.release(); }

  static void reset(pointer& ptr) throw() { return ptr.reset(); }

  template<typename Arg>
  static void reset(pointer& ptr, Arg arg) throw() { return ptr.reset(arg); }
};
```

The pointer traits for native pointer types are defined as follows. The difference here is that the `get()` trait does not reset the origin pointer value, but `release()` does.

Listing 2.9: Pointer traits for native pointers.

```
template<typename Tp>
class pointer_traits<Tp *> {
 public:
  typedef Tp *pointer;
  typedef Tp element_type;

 private:
  typedef value_traits<pointer> traits_type;

 public:
  static bool is_nil(const pointer& ptr) throw() {
    return ptr == traits_type::nil;
  }
```

```
14
15    static bool is_invalid(const pointer& ptr) throw() {
16      return ptr == traits_type::invalid;
17    }
18
19    static pointer get(const pointer& ptr) throw() { return ptr; }
20    static pointer release(pointer& ptr) throw() {
21      pointer tmp = ptr;
22      reset(ptr);
23      return tmp;
24    }
25
26    static void reset(pointer& ptr, pointer arg = traits_type::nil) throw() {
27      ptr = arg;
28    }
29  };
```

An example use of the pointer traits is another idiom for explicitly transferring ownership and set the origin to `nil`. Recall that assignment makes the origin `invalid` instead of `nil`; this is the case so that linearity enforcement can distinguish lack of ownership from the `nil` resource, which is a data structure sentinel. However, it is common for data structure algorithm to move ownership and immediately set the origin to `nil`. This is captured in the `transfer_linear()` idiom as follows.

Listing 2.10: Transfer linear ownership idiom.

```
1  template<class Ptr /* implements pointer<?> */>
2  Ptr transfer_linear(Ptr& p) throw() {
3    // Do not use p.release(), as the color would be lost in the case of
4    // chromatic pointers.
5    Ptr q(p);
6    pointer_traits<Ptr>::reset(p);
7    return q;
8  }
```

In most cases, resource managed by a linear pointer `p` can be disposed by `delete p.release()`, but it is also desirable to simply destruct the resource in-place and obtain the raw, uninitialized memory for other use. This is exceptionally common

when writing a memory allocator. This is captured in the `destroy_linear()` idiom as follows.

Listing 2.11: Destroy linear resource idiom.

```
template<class Ptr /* implements pointer<?> */>
void *destroy_linear(Ptr lptr) throw() {
  typedef pointer_traits<Ptr> traits_type;
  typedef typename traits_type::element_type element_type;
  element_type *p = traits_type::get(lptr);
  p->~element_type();
  return static_cast<void *>(p);
}
```

2.7 Idioms for Borrowing

We often want to pass a linear object without relinquishing ownership. This entails controlled sharing by creating aliased references to that object. In the general case, the only way to ensure safety in the presence of aliasing is by reference counting. In some specific cases, when a caller passes an alias to a function, the function's (unenforced) convention makes it clear that the alias will not be duplicated, and the alias is only used during the scope of the function call. When this is the case, we could do away without reference counting, but use some resource borrowing idioms to help avoiding programming mistake.

The most common way to pass an object without reliquishing ownership is by passing its lvalue reference. This is useful because one can only delete a pointer, not an lvalue reference.

```
void F(A& arg);

linear_ptr<A> p(new A());
F(*p);  // pass A by reference.
delete p.release();
```

Sometimes it is more convenient to pass a const lvalue reference to the linear pointer, especially for data structure traversal. When passing a const reference to the linear pointer, the compiler enforces that the pointer cannot lose ownership to the resource because it cannot be changed.

```
typedef linear_ptr<binary_tree_node> ptr_type;
size_t height(const ptr_type& root);

ptr_type p(new binary_tree_node());
assert(height(p) == 1u);
delete p.release();
```

However, lvalue reference has limitations. Once a variable is declared as a lvalue reference to some object, the reference cannot be bound to another object. This makes lvalue reference unsuitable for data structure that might want to maintain a "parent pointer" in the node. The parent pointer facilitates in-place removal and sometimes traversal, but the parent pointer cannot be linear, since the grandparent node already has the only linear pointer that points to the parent. In such cases, we may need to sort to using a native pointer to the resource, but there is a way to protect the resource so that it can never be deleted. Listing 2.12 presents a `loan<>` container of an object. Using the *mixin* pattern [107], `loan<Super>` is a subclass of the protected object type `Super` that disallows construction and destruction by declaring them private, so that `loan<Super> *` is a pointer that cannot be deleted.

Listing 2.12: Loan of an object.

```
template<class Super>
class loan : public Super {
 private:
  // Disallow constructor and destructor.
  loan() throw();
  ~loan() throw();
};
```

The `make_loan()` idiom is introduced here for obtaining loan pointer. If we're

given a native pointer, it suffices to do a `static_cast` because `loan<Super>` is a subclass of `Super`. Here the code is shown with `const volatile` qualifier. Each combination of the qualifier must be written for `make_loan()` if we wish to operate on qualified objects, but they are omitted here for brevity.

```
template<class Super>
const volatile loan<Super> *
make_loan(const volatile Super *objp) {
  return static_cast<const volatile loan<Super> *>(objp);
}
```

Obtaining a loan pointer from `linear_ptr<Tp>` requires a little trickery. The `&*objp` expression dereferences `objp` and immediately computes the address of the object. This works because `linear_ptr<Tp>::operator*` returns `Tp&` which can be restored to a native pointer `Tp *` using `operator&`.

```
template<class Super, class Traits>
const volatile loan<Super> *
make_loan(const linear_ptr<const volatile Super, Traits>& objp) {
  return static_cast<const volatile loan<Super> *>(&*objp);
}
```

Linear chromatic pointer can be treated similarly, so it is omitted here for brevity.

2.8 Conclusion

This chapter described how *resource acquisition is initialization* can be used to implement a smart pointer in C++ in order to be notified whenever the program's control flow enters and leaves the scope of a variable. A linear pointer is presented where the constructor and assignment transfer ownership away from another linear pointer, and the destructor ensures that the pointer has no resource ownership. Together, this enforces linear ownership semantics for resources managed by the linear pointer at runtime.

Linear pointer uses value traits for the concrete representation of the special values for `nil`, which represents the empty resource that can be arbitrarily introduced and eliminated, and `invalid`, which represents an uninitialized state of the resource after the ownership is transferred away. The distinction of `nil` and `invalid` allows us to preserve program behavior when linearity enforcement is in effect. Linearity enforcement can be turned off using a compile-time flag so that a program can be compiled without the linearity-checking overhead and still retain the correct behavior.

Pointer traits are defined so that linear pointer and native pointers can be operated alike. This is used to build two more idioms, the `transfer_linear()` idiom which explicitly reinitializes the origin to `nil` rather than leaving it `invalid`, and the `destroy_linear()` idiom which deconstructs a linear resource and returns the underlying raw memory. Several borrowing idioms are discussed to pass a linear resource around a program without reliquishing ownership.

We are now ready to use linear pointer to build data structures such as singly linked list, binary search trees, and hash tables, in a way that linear conservation of resources is guaranteed.

Chapter 3

Using Linear Pointer

Programs organize information in memory as objects and pointers, forming data structures which allow higher dimensional information to be stored and retrieved. Memory allocators also use data structures to organize the storage and retrieval of free memory blocks. Manipulation of data structures is error prone, and programming error often leads to data structure corruption that does not manifest immediately. The program may crash or enter an infinite loop at a later time when traversing the data sturcture, often at an innocuous site that bears little relation to the site where the error is made. This is why memory errors are notoriously hard to debug. By observing that common data structures such as linked list and binary tree never required sharing, linear ownership semantics can be used to eliminate memory errors concerning leak, double free, and dangling pointers. Linear pointer is an effective debugging aid to avoid these memory errors. Even though linearity checking is done in runtime, the program would abort immediately when linearity violation occurs at the error site. This makes it easy to identify and fix programming errors.

This chapter shows how to use linear pointer to implement data structures, so that the implementation's correctness—namely adherence to linear ownership of objects—

may be verified through the use of linear pointers. Although as shown in prior $\mathcal{ATS}$ work, it is well known that linear implementation of data structures is possible [135, 133], linear pointer presents the programmer with a different set of challenges. Unlike $\mathcal{ATS}$ which allows linear ownership and the pointer to be two separate entities, linear pointer uses the pointer itself to represet ownership. In order to traverse data structures without constantly exchanging ownership of nodes between the visitor and the data structure, the notion of *cursor* is developed. A cursor is an lvalue reference or a pointer to a linear pointer that allows moving from pointer to pointer without modifying it. A similar owner cursor technique facilitates in-place editing of data structure through *ownership stealing*, used with what we call an augmented linked list. The chapter will show that ownership stealing is simpler to implement correctly than doubly linked data structures because there are fewer corner cases to consider.

The code presented in this chapter is both pointer and allocator agnostic. Neither properties are required for programs using linear pointers, but they are a requirement for being able to use the implementation for memory allocators.

Pointer agnosticism The implementation does not use linear pointer direclty. Instead, the code is structured as a template parameterized by an arbitrary pointer type, and the template can be instantiated with linear pointers as well as native pointers. The fact that this works is a result of the erasure property of linear pointers, which shows that linearity is really a refinement—a program that observes linear ownership semantics will also behave correctly with non-linear (e.g. reference counted or garbage collected) pointers. Once the code is verified to be linear, it will retain this property without runtime linearity checking.

Allocator agnosticism When implementing data structure code, it is common to expose only a high level interface such as adding and removal of data item, and hide the internal detail of pointers. This requires the implementation to be able to allocate memory for data structure nodes. Instead, an allocator agnostic implementation would require the data item itself to implement a well-defined interface with pointer accessors. Linear ownership transfer is adequate for adding and removing items. If items need to be ordered such as in the case of binary search tree, the data item would expose the interface for ordering as well. If items need to be keyed such as being used with a hash table, then the data item would expose its key. Both singly linked list nodes and binary tree nodes are arbitrary user-supplied classes exposing these well-defined interfaces to access their members.

Verifying that code is linear involves writing some unit tests that instantiate the template with linear pointers and running the tests to check for any linear ownership semantics violation. Throughout the chapter, examples are given to show how using linear pointer helps the programmer debug memory errors.

3.1 Singly Linked List

A singly linked list is a data structure that consist of nodes with one pointer pointing to the next node in the list. The relationship from a node to its successor establishes the ordering of items in the list, but the items are not required to be in any particular order. The list is terminated at the `nil` pointer, which also represents an empty list. Singly linked list is the simplest data structure to implement, but it is not the most efficient for searching because the item to look for might be at any position in the list, which may require searching through the whole list in $\Theta(n)$ time where n is the number of items.

Memory allocators use singly linked list as the "free list" of objects of the same size, so it doesn't matter which object in the list should be used for allocation. Free objects are stored and retrieved in a last-in first-out manner which are $\Theta(1)$ operations. Memory allocator would have multiple free lists, at least one per object size, but there could be more. This implements an allocation strategy called segregate-fits.

A singly linked node is a class that implements an interface with the accessor `next()` for the pointer to the next node. The pointer type as the template argument of the node interface is unconstrained for now.

Listing 3.1: Singly linked node interface.

```
template<class Ptr>
class singly_linked_node {
 public:
  typedef Ptr ptr_type;
  const ptr_type& next() const throw();
  ptr_type& next() throw();
};
```

Here is an example implementation of the singly linked node interface that also carries a key and a value. The pointer discipline is hard coded in this node to use linear pointer. The node also implements a `compare_to()` function that will be used for sorting by the key.

Listing 3.2: A singly linked node implementation.

```
template<typename Key, typename Value>
class Node /* implements
              singly_linked_node<linear_ptr<Node<Key, Value> > >,
              comparable_with<linear_ptr<Node<Key, Value> > */ {
 public:
  typedef linear_ptr<Node<Key, Value> > ptr_type;

  Node(const Key& key_ = Key(), const Value& value_ = Value())
    : key(key_), value(value_) {}

  const Key key;
  Value value;
```

```
13
14   const ptr_type& next() const throw() { return next_; }
15   ptr_type& next() throw() { return next_; }
16
17   ord_t compare_to(const ptr_type& that) const throw() {
18     if (this->key < that->key)
19       return LESS;
20     if (this->key > that->key)
21       return GREATER;
22     return EQUAL;
23   }
24
25  private:
26   ptr_type next_;
27 };
```

The implementation of the singly linked list is a template class with only static methods, parameterized by the pointer type to the node. The node type is implied by the pointer type. The template argument has a recursive constraint that requires the destination of the pointer to implement the singly linked node interface using the same pointer type.

Listing 3.3: Singly linked list template outline.

```
template<
  class Ptr /* implements pointer<? implements
                singly_linked_node<Ptr> > */>
class singly_linked_list {
 public:
  typedef Ptr ptr_type;
  ...
};
```

Within the body of the template, let us first introduce the node insertion function. In the code below, `node` is a pointer to a single node to be inserted at `curr` which is the current cursor. The cursor is a reference to a linear pointer that the function modifies in-place. It can be the pointer to the first node, the next pointer of the last node, or anywhere in between. The `node` pointer will be stored at `curr`, and the

previous value of `curr` is stored at the node's next pointer. This code is no different whether linear pointer is used or not.

```
1  static ptr_type&
2  insert_at(ptr_type node, ptr_type& curr) throw() {
3    assert(node);
4    node->next() = curr;
5    curr = node;
6    return curr;
7  }
```

One potential error that the caller of this function could make is to supply a `node` pointer that points to a multiple-item list (i.e. `node->next()` is not `nil`) rather than a singleton node (i.e. `node->next()` is `nil`). The function is correct with respect to its intended purpose, but the caller might have misunderstood the purpose and thought that the function could insert a list segment in the middle of a list. After the insertion, the rest of the inserted list is truncated by the assignment on line 4 above. Such memory leak would be hard to find without linearity checking. This error is not a case where naive compiler annotation or static analysis can effectively detect. It also would not cause any corruption in the data structure, so the program would carry on until it runs out of memory. Later on, we will write a unit test to show that linearity checking correctly catches memory leak in this case.

Opposite to node insertion is node removal. In the code below, the node at the current cursor `curr` is removed and returned. The cursor is modified to point to the next node. The implementation is naive. It doesn't check that `curr` is a `nil` pointer which would cause NULL pointer dereference when reading `curr->next()`, but asserting that `curr` is not `nil` is rarely necessary because the hardware will abort the program with segmentation fault for us on access.

```
1  static ptr_type
2  remove_at(ptr_type& curr) throw() {
3    ptr_type node(curr);
```

```
    curr = transfer_linear(node->next());
    return node;
  }
```

Here we have the basic operations to manipulate a singly linked list, so let us test our implementation. In our test, we will use the node implementation in Listing 3.2 with both `Key` and `Value` instantiated to the `int` type.

```
typedef Node<int, int> NodeT;
typedef linear_ptr<NodeT> PtrT;
typedef singly_linked_list<PtrT> ListT;
```

To make testing easier, let us write two more functions, one that constructs a list from an array of integers `ks` of length `len`, and one that destructs a list node by node, counts the number of nodes in the list, and checks that the keys of the nodes are in order. The latter function is useful for checking the correctness of singly linked list sorting algorithms, to be discussed later.

```
static PtrT NewList(const int *ks, size_t len) {
  PtrT first;
  for (size_t i = len; i > 0; --i) {
    int k = ks[i - 1];
    ListT::insert_at(PtrT(new NodeT(k, k)), first);
  }
  return first;
}

static void DeleteSortedList(PtrT first, size_t total) {
  ASSERT_TRUE(first);
  int prev_key = first->key;
  size_t count = 0;

  do {
    EXPECT_LE(prev_key, first->key);
    prev_key = first->key;

    PtrT p = ListT::remove_at(first);
    delete p.get();
    ++count;
  } while (first);
```

```
23
24    EXPECT_EQ(total, count);
25  }
```

Let `xs` be a literal integer array that we can feed into the list construction function.

```
static const int xs[] = { 1, 2, 3 };
```

And let `countof()` be a C Preprocessor macro that makes it easier to infer the length of a literal array, defined as follows.

```
#define countof(array) (sizeof(array) / sizeof(array[0]))
```

Then we can write the first test for insert and remove as follows.

```
1  TEST(SinglyLinkedList, InsertRemove) {
2    PtrT nodes = NewList(xs, countof(xs));
3    DeleteSortedList(nodes, countof(xs));
4  }
```

Once we build the test and run it, we get an output like this, which shows that the test runs correctly.

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SinglyLinkedList
[ RUN      ] SinglyLinkedList.InsertRemove
[       OK ] SinglyLinkedList.InsertRemove (0 ms)
[----------] 1 test from SinglyLinkedList (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

The following test shows that attempting to call `remove_at()` with a `nil` pointer will result in segmentation fault. It is called a *death test* where the test passes if the statement causes the program to die in a specific way. In this case, we expect the program to be killed by `SIGSEGV`.

```
1  TEST(SinglyLinkedList, RemoveNil) {
2    PtrT node;
3    ASSERT_EXIT(
```

```
4         ListT::remove_at(node),
5         ::testing::KilledBySignal(SIGSEGV), "");
6   }
```

Revisiting the potential memory leak issue in Listing 3.1 where inserting a node whose `node->next()` is not `nil` causes the rest of the list to be truncated, we write the following death test that verifies that the statement would be aborted with `SIGABRT` with the error message mentioning "is_invalid()" from the linear pointer assertion check. Note that this time, we check the death condition only for the debug build when linearity checking is enabled (i.e. `NDEBUG` is not defined). Otherwise, the test would fail in the release build because linearity checking is disabled.

```
1   TEST(SinglyLinkedList, InsertNotLone) {
2     PtrT nodes = NewList(xs, countof(xs));
3     PtrT first;
4   #if !NDEBUG
5     ASSERT_EXIT(
6         ListT::insert_at(nodes, first),
7         ::testing::KilledBySignal(SIGABRT), "is_invalid()");
8   #endif
9     DeleteSortedList(nodes, countof(xs));
10  }
```

Finally, we run the test again to ensure that all the tests pass.

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from SinglyLinkedList
[ RUN      ] SinglyLinkedList.InsertRemove
[       OK ] SinglyLinkedList.InsertRemove (0 ms)
[ RUN      ] SinglyLinkedList.RemoveNil
[       OK ] SinglyLinkedList.RemoveNil (315 ms)
[ RUN      ] SinglyLinkedList.InsertNotLone
[       OK ] SinglyLinkedList.InsertNotLone (188 ms)
[----------] 3 tests from SinglyLinkedList (503 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (503 ms total)
[  PASSED  ] 3 tests.
```

The `RemoveNil` death test ensures that the test could reliably assert the death condition, and the `InsertNotLone` death test demonstrates that linearity checking causes the program to abort immediately upon memory leak, making the problem trivial to find. The death tests take longer to run because the test is doing a `fork()` before running the death statement and waiting for the child process to run the statement and exit.

3.2 Singly Linked Segment

A singly linked segment is a singly linked pointer and a tail cursor. The tail cursor allows another node or segment to be quickly appended to the segment in $\Theta(1)$ time. This is useful for efficient concatenation. When appending a list, the tail cursor must be fast-forwarded to the end of the list, but repeatedly appending lists only requires an overall $\Theta(n)$ running time. When appending to tail and popping from head, the segment can be used like a queue without doubly linking the list.

This is embodied into the `singly_linked_segment` template class below. The class has two members: a singly linked list that constitutes the segment, and the tail cursor. The segment retains ownership of the list. A singly linked segment is not copyable if it is instantiated with a linear pointer because linear pointers are not copyable. Below is the template outline.

Listing 3.4: Singly linked segment template outline.

```
template<
  class Ptr /* implements pointer<
                 ? implements singly_linked_node<Ptr> > */>
class singly_linked_segment /* implements consume<Ptr> */ {
 public:
  typedef Ptr ptr_type;

  singly_linked_segment() throw()
```

```
    : head_(), tail_(&head_) {}
  ...

 private:
  ptr_type head_;
  ptr_type *tail_;
};
```

For now, the default constructor will initialize the segment with an empty list, where the tail cursor initially points to the head which is the `nil` pointer. The segment maintains an invariant that the tail cursor always points to the `nil` pointer at the end of the list.

Let us first provide a method to populate the segment one node at a time. It puts the node at the tail, then advances the tail to the node's next pointer.

```
1  void append_node(ptr_type node) throw() {
2    *tail_ = node;
3    tail_ = &(*tail_)->next();
4  }
```

The implementation has to observe a slight nuance in order to be linear. If `ptr_type` is a linear pointer, then after the assignment in line 2 above, `node` transfers ownership to `*tail_` and becomes `invalid`, so we cannot use `node->next()` anymore which would dereference an invalid pointer. We have to use `*tail_` instead. Compare this to the following non-linear implementation.

```
1  void append_node_bad(ptr_type node) throw() {
2    *tail_ = node;
3    tail_ = &node->next();  // !
4  }
```

With ordinary pointers, the non-linear implementation is perfectly correct. Linear pointer makes the implementation more strict by disallowing uncontrolled sharing. If we use a linear pointer with the non-linear implementation, `tail_` would be the address of the invalid pointer plus some small offset to the member returned by `next()`,

but without triggering segmentation fault. This means that the invalid pointer in `tail_` could propagate to other method calls of the singly linked segment from a difficult to trace origin. This can be a real problem akin to tracing uninitialized variables. The reason no segmentation fault is triggered is because the address operator "`&`" computes the address obtained from `next()` through `operator->()` without dereferencing the pointer. It is possible to fix this by adding `!is_invalid()` assertion checks in `operator->()`, but this would also make linear pointer slower. Those who value ease of debugging over speed of execution or vice versa can choose their trade-offs.

Before we can write tests for the singly linked segment, we need a way to obtain the list from the segment. The following function makes it possible to append another list to the segment for the last time before getting the combined list back.

```
1  ptr_type clear(ptr_type rest = ptr_type()) throw() {
2    *tail_ = rest;
3    tail_ = &head_;
4    return transfer_linear(head_);
5  }
```

Let us write tests for appending a node. First instantiate the singly linked segment with the node implementation in Listing 3.2 as follows.

```
typedef singly_linked_segment<PtrT> SegT;
```

The append node test builds a segment with nodes keyed from 1 through 5, then clears the segment and deletes the resulting list after ensuring the ordering of the keys and list length.

```
1 TEST(SinglyLinkedSegment, AppendNode) {
2   SegT seg;
3   for (int i = 1; i <= 5; ++i)
4     seg.append_node(PtrT(new NodeT(i, i)));
5   DeleteSortedList(seg.clear(), 5);
6 }
```

The following test illustrates the death pattern of the non-linear append node function. The segment's `tail_` pointer becomes corrupted after the first `append_node_bad()`, but the error does not cause the program to die until another node is appended.

```
TEST(SinglyLinkedSegment, AppendNodeBad) {
  SegT seg;
  NodeT node(1, 1);
#if !NDEBUG
  ASSERT_EXIT({
      seg.append_node_bad(PtrT(&node));  // error here.
      std::cerr << "after␣append_node_bad" << std::endl;
      seg.append_node_bad(PtrT());  // crash here.
    }, ::testing::KilledBySignal(SIGSEGV), "after␣append_node_bad");
#endif
}
```

Like the singly linked list insertion function, `append_node()` does not check that the given node is a singleton node. If the caller gives a `node` where `node->next()` is not `nil`, then the resulting segment would violate the invariant that `*tail_` is a `nil` pointer. Next time `append_node()` is called, the segment would be truncated at `*tail_`. Without linear checking, this would cause memory leak without corrupting the data structure. The caller might make this error if one mistakenly believes that `append_node()` can append a list. Again, this is not a case where naive compiler annotation or static analysis can detect. The following death test shows that linear checking can detect this kind of error.

```
TEST(SinglyLinkedSegment, AppendNodeNotLone) {
  PtrT nodes = NewList(xs, countof(xs));
  SegT seg;
#if !NDEBUG
  ASSERT_EXIT({
      seg.append_node(nodes);
      seg.append_node(PtrT(new NodeT(4, 4)));
    }, ::testing::KilledBySignal(SIGABRT), "is_invalid()");
#endif
  DeleteSortedList(nodes, countof(xs));
}
```

The append function can be generalized to append a list by fast-forwarding the tail cursor to the end of the list as follows.

```
1  void append(ptr_type first) throw() {
2    *tail_ = first;
3    while (*tail_)
4      tail_ = &(*tail_)->next();
5  }
```

Its test is similar to the death test above, except the test will not die this time.

```
1 TEST(SinglyLinkedSegment, AppendList) {
2   SegT seg;
3   seg.append(NewList(xs, countof(xs)));
4   seg.append(PtrT(new NodeT(4, 4)));
5   DeleteSortedList(seg.clear(), 4);
6 }
```

Here is the test output after running the test to ensure that all tests pass.

```
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from SinglyLinkedSegment
[ RUN      ] SinglyLinkedSegment.AppendNode
[       OK ] SinglyLinkedSegment.AppendNode (0 ms)
[ RUN      ] SinglyLinkedSegment.AppendNodeBad
[       OK ] SinglyLinkedSegment.AppendNodeBad (218 ms)
[ RUN      ] SinglyLinkedSegment.AppendNodeNotLone
[       OK ] SinglyLinkedSegment.AppendNodeNotLone (208 ms)
[ RUN      ] SinglyLinkedSegment.AppendList
[       OK ] SinglyLinkedSegment.AppendList (0 ms)
[----------] 4 tests from SinglyLinkedSegment (426 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (428 ms total)
[  PASSED  ] 4 tests.
```

3.3 Singly Linked List Sorting

Sorting is an interesting case study because the problem is easy to describe, and there are plenty of opportunities to make mistakes. As far as a singly linked list is

concerned, the algorithm takes as an input an unsorted list and modifies the list so that the items become arranged in ascending order. The result is a permutation of the original in the sense that items are merely shuffled around, neither added nor removed. We will look at two algorithms, quick sort and merge sort. Both sort the list using the divide and conquer strategy in $O(n \lg n)$ time in the average case.

Sorting requires that any node be comparable to any other node, but otherwise the sorter does not need to know how the comparison is done. Again, let the template class be parameterized by the pointer type, and the sorting algorithms will be static methods of the class.

Listing 3.5: Singly linked list comparable template outline.

```
template<
  class Ptr /* implements pointer<? implements
                singly_linked_node<Ptr>, comparable_with<Ptr> > */>
class singly_linked_list_comparable {
 public:
  typedef Ptr ptr_type;
  typedef singly_linked_list<ptr_type> list_type;
  typedef singly_linked_segment<ptr_type> seg_type;
  ...
};
```

Quick sort starts by choosing a pivot point (typically the first node of the list), then divides the rest of the list into two, one with nodes lesser than the pivot, and the other with nodes greater. Each list is recursively quick sorted, then the lesser list, the pivot, and the greater list are concatenated in order.

Listing 3.6: Quick sort of a singly linked list.

```
1   static void
2   quicksort(ptr_type& first) throw() {
3     // Empty list or singleton list are already sorted.
4     if (!first || !first->next())
5       return;
6
7     // Divide
8     seg_type seg;
```

```
9       ptr_type pivot = list_type::remove_at(first);
10
11      ptr_type *curr = &first;
12      while (*curr) {
13        if (pivot->compare_to(*curr) == LESS)
14          seg.append_node(list_type::remove_at(*curr));
15        else
16          curr = &(*curr)->next();
17      }
18
19      ptr_type lesser = first;
20      ptr_type greater = seg.clear();
21
22      // Conquer
23      quicksort(lesser);
24      quicksort(greater);
25
26      // Combine
27      seg.append(lesser);
28      seg.append(pivot);
29      seg.append(greater);
30
31      first = seg.clear();
32    }
```

Merge sort divides the list using two cursors as it traverses down the list, `mid` which advances one node at a time, and `last` which advances two nodes at a time. When `last` cannot be advanced further, the the list is split at `mid`, sorted separately, and merged by taking the lesser element from the two sorted sublists.

Listing 3.7: Merge sort of a singly linked list.

```
1   static void
2   mergesort(ptr_type& first) throw() {
3     // Empty list or singleton list are already sorted.
4     if (!first || !first->next())
5       return;
6
7     // Divide
8     ptr_type *mid = &first, *last = &first;
9     while (*last && (*last)->next()) {
10      mid = &(*mid)->next();
```

```
11        last = &(*last)->next()->next();
12      }
13
14      // Conquer
15      ptr_type one = first, two = transfer_linear(*mid);
16      mergesort(one); mergesort(two);
17
18      // Merge
19      seg_type seg;
20
21      while (one && two)
22        seg.append_node(
23            list_type::remove_at(
24                one->compare_to(two) != GREATER ?
25                one : two));
26
27      first = seg.clear(one? one : two);
28    }
```

Writing tests for sorting algorithms is easy. First instantiate the singly linked list comparable template, then make up a list in arbitrary order like this:

```
typedef singly_linked_list_comparable<PtrT> ListCT;
static const int ys[] = { 5, 2, 6, 4, 8, 0, 9, 7, 1, 3 };
```

Then just use the `NewList()` and `DeleteSortedList()` helpers we already wrote.

```
1  TEST(SinglyLinkedListComparable, QuickSort) {
2    PtrT first = NewList(ys, countof(ys));
3    ListCT::quicksort(first);
4    DeleteSortedList(first, countof(ys));
5  }
6
7  TEST(SinglyLinkedListComparable, MergeSort) {
8    PtrT first = NewList(ys, countof(ys));
9    ListCT::mergesort(first);
10   DeleteSortedList(first, countof(ys));
11 }
```

Run the test and see that both tests pass.

```
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
```

```
[----------] 2 tests from SinglyLinkedListComparable
[ RUN      ] SinglyLinkedListComparable.QuickSort
[       OK ] SinglyLinkedListComparable.QuickSort (1 ms)
[ RUN      ] SinglyLinkedListComparable.MergeSort
[       OK ] SinglyLinkedListComparable.MergeSort (0 ms)
[----------] 2 tests from SinglyLinkedListComparable (1 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 2 tests.
```

We can check that linear pointer is working by deliberately introducing errors into the quick sort code in Listing 3.6. If we remove line 28, we would concatenate only the lesser and the greater lists without the pivot, and the pivot would leak. When we run the test, we would get an error like this:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from SinglyLinkedListComparable
[ RUN      ] SinglyLinkedListComparable.QuickSort
Assertion failed: (self()->is_invalid() || self()->is_nil()),
function ~linear_base, file ...
```

3.4 Augmented Linked List

Now that we've seen a singly linked list implementation using linear pointers, the reader may ask "what about doubly linked list?" Whereas a singly linked node has only the next pointer, a doubly linked node has both the next and the previous pointer. The previous pointer allows bidirectional traversal, but most often it is used for in-place removal of a node if only a reference to the node is available, but not a reference to the pointer that owns the node. To remove a node, one will connect the predecessor's next pointer to the successor, and the succesor's previous pointer to the predecessor, isolating the current node.

The tricky part for removal is to handle the corner cases for the first node that has no predecessor, and the last node that has no successor. But there are pointers elsewhere that point to the first and last nodes, and they must be updated. Insertion also has the analogous considerations. What if we are trying to insert before the first node or after the last node? What if the list is initially empty? If the corner cases are not handled correctly, it would lead to data structure corruption that manifest later appearing as a different problem. Although doubly linked list is a standard data structure taught as part of an undergraduate curriculum in computer science, its redundancy and corner cases make it tedious to implement correctly.

A careful reader will notice that linear pointer cannot be used with doubly linked lists: only one of the previous or the next pointer can be linear because exactly one linear pointer to the node is allowed. The other pointer must be the "loan" pointer. Alternatively, if we only need in-place removal but not bidirectional traversal, then in lieu of the previous pointer, we can just store the reference to the owner pointer so that we can perform *ownership stealing*. The benefit, however, is the drastic reduction of corner cases. There is only one corner case: dealing with the `nil` pointer which has no owner. Not handling this case causes an immediate segmentation fault.

In order for a node to be used as part of an augmented linked list, it must implement the ownership interface like this:

Listing 3.8: Ownership interface.

```
1 template<class Ptr>
2 class ownership {
3  public:
4   typedef Ptr ptr_type;
5
6   ptr_type *owner() const throw();
7   ptr_type *& owner() throw();
8 };
```

A template for augmented linked list follows the same fashion as the singly linked

list. The class will only have static methods.

Listing 3.9: Augmented linked list template outline.

```
template<
  class Ptr /* implements pointer<? implements
                 singly_linked_node<Ptr>, ownership<Ptr> > */>
class augmented_linked_list {
 public:
  typedef Ptr ptr_type;
  typedef typename Ptr::element_type node_type;
  ...
};
```

Node insertion and removal are very similar to that of singly linked list. The only differences are the lines that are marked with "`// !`" which update the ownership. All the other lines are exactly identical. The corner case handling of the `nil` pointer is guarded by an if-statement.

```
  static ptr_type&
  insert_at(ptr_type node, ptr_type& curr) throw() {
    assert(node);

    if (curr) curr->owner() = &node->next();  // !
    node->next() = curr;

    node->owner() = &curr;  // !
    curr = node;

    return curr;
  }

  static ptr_type
  remove_at(ptr_type& curr) throw() {
    ptr_type node(curr);
    node->owner() = NULL;  // !

    curr = transfer_linear(node->next());
    if (curr) curr->owner() = &curr;  // !

    return node;
  }
```

To remove a node in the absence of a cursor to its owner, we will recover the owner from the node itself. The in-place removal is simply:

```
1  static ptr_type
2  remove_at(node_type& node) throw() {
3    assert(node.owner() != NULL);
4    return remove_at_unsafe(*node.owner());
5  }
```

As an exercise to the reader, unit tests can be written in the same fashion as those for the singly linked list, so they are omitted here. All that is required is to add the `owner()` method to `class Node` in Listing 3.2, and instantiate `ListT` from augmented linked list instead of singly linked list.

For sorting, it is more efficient to temporarily regard an augmented linked list as a singly linked list, then fix the node ownership after the whole list is sorted, using the following function.

```
1  static void
2  relink(ptr_type& first) throw() {
3    for (ptr_type *curr = &first; *curr; curr = &(*curr)->next())
4      (*curr)->owner() = curr;
5  }
```

3.5 Binary Search Tree

A binary tree is a data structure consisting of nodes with two pointers to the *left* and *right* subtrees. The `nil` pointer indicates the empty tree or the leaf. A binary search tree is a binary tree where the nodes have a total order when compared with one another, and ordering constraint is imposed on the left and right subtrees. In a binary search tree, the left subtree contains only the nodes that compare less than the parent, and the right subtree contains only the nodes that compare greater than the parent.

The ordering constraint facilitates searching; during traversal, only one of the subtrees need to be visited depending on whether the node to look for is less than (left) or greater than (right) the current node. If the tree is perfectly balanced, each descend eliminates half of the tree to visit. In general, the number of visits during traversal is bounded by tree height. A perfectly balanced binary tree of size n has height $\lg n$, so traversal is in $O(\lg n)$ time. Even if a tree is not perfectly balanced, as long as the tree height is within a constant factor to the height of the perfectly balanced tree, the node lookup time is still $O(\lg n)$ since the height is $c \lg n$ for some constant factor c. This approximately balanced binary search tree underlies the performance characteristics of variants such as AVL tree [4], red-black tree [60], and splay tree [106]. Tree rotation is used to adjust the tree height in order to keep the binary search tree approximately balanced.

If the binary search tree must also allow nodes that compare equal, then we relax the ordering to be less than or equal to and greater than or equal to, for the left and right subtrees respectively. This relaxation has to be done for both left and right subtrees in order to accommodate tree rotation. Otherwise it would be impossible to keep the tree approximately balanced if there is a long series of equal nodes in the tree. Tree rotation preserves the stable order of the equal nodes in a series, but the node closest to the root may be any node in the series, not necessarily the first nor the last. This requires special consideration if the lookup, insertion, and removal were to observe the stable order.

Memory allocator uses binary search tree to index free objects of various sizes. Although several objects of the same size could be present in the tree, the stable order is unimportant. Objects can also be disambiguated by comparing their size as well as their memory address. Binary search tree allows lookup of a node that is the least greater than, or the greatest less than the desired node. The former is used to

implement an allocation strategy known as best-fit.

In a similar minimalist fashion as singly linked node, the binary tree node here implements accessors for the left and right pointers only. Here the `is_lone()` predicate returns true if both the left and right subtrees are empty, and false otherwise. Again, the node defines no key nor value.

Listing 3.10: Binary tree node.

```
template<
  class Ptr /* implements pointer<? extends binary_tree_node_impl<Ptr> > */>
class binary_tree_node_impl /* implements binary_tree_node<Ptr> */ {
 public:
  typedef Ptr ptr_type;

  binary_tree_node_impl() throw()
    : left_(), right_() {}
  bool is_lone() const throw() {
    return !left_ && !right_;
  }

  const ptr_type& left() const throw() { return left_; }
  ptr_type& left() throw() { return left_; }
  const ptr_type& right() const throw() { return right_; }
  ptr_type& right() throw() { return right_; }

 private:
  ptr_type left_, right_;
};
```

Here are the binary search tree traversal functions, parameterized by a pointer to the node. In this implementation, a node is not required to be comparable with another node, but instead a comparator is used for this purpose. The reason will become apparent when splay tree is introduced below. The simple traversal here does not attempt to re-balance the tree in any way.

Listing 3.11: Binary search tree traversal.

```
template<
  class Ptr /* implements pointer<? implements binary_tree_node<Ptr> > */>
struct binary_search_tree_find_impl {
```

```
4  public:
5   typedef Ptr ptr_type;
```

The `find()` function takes the root of the tree and returns a cursor, which is a reference to the linear pointer that owns the node to be removed or must take ownership in order to insert a node. The comparator compares itself with the each visited node. The comparator directs the traversal to the left subtree if it returns `LESS`, to the right subtree if it returns `GREATER`, or returns the cursor to the current node if it returns `EQUAL`. If no node compares equal, returns the cursor to the `nil` pointer where the node would be inserted.

```
6   template<class Comparator /* implements comparable_with<Ptr> */>
7   static ptr_type&
8   find(Comparator& cmp, ptr_type& root) throw() {
9     ptr_type *curr = &root;
10    while (*curr) {
11      ord_t dir = cmp.compare_to(*curr);
12      if (dir == LESS)
13        curr = &(*curr)->left();
14      else if (dir == GREATER)
15        curr = &(*curr)->right();
16      else /* if (dir == EQUAL) */
17        break;
18    }
19    return *curr;
20  }
```

The `find_limit()` function looks up a node with a relaxed limit. If the limit is `GREATER`, returns a node that is the minimal greater than or equal to the comparator. If the limit is `LESS`, returns a node that is the maximal less than or equal to the comparator. The function returns the cursor to the `nil` pointer if no such node exists, in which case the cursor also indicates where the node would be inserted.

```
21  template<class Comparator /* implements comparable_with<Ptr> */>
22  static ptr_type&
23  find_limit(Comparator& cmp, ord_t limit, ptr_type& root) throw() {
24    ptr_type *curr = &root, *foundp = NULL;
```

```
25
26      while (*curr) {
27        ord_t dir = cmp.compare_to(*curr);
28        if (dir != limit || dir == EQUAL)
29          foundp = curr;
30
31        if (dir == LESS)
32          curr = &(*curr)->left();
33        else if (dir == GREATER)
34          curr = &(*curr)->right();
35        else /* if (dir == EQUAL) */
36          break;
37      }
38      return (foundp != NULL)? *foundp : *curr;
39    }
```

Several other tree traversal functions can be written in terms of find(), but the discussion is postponed for now.

```
40  };
```

It is worth noting that the way `find()` returns a cursor for the purpose of both node insertion and removal will not work for AVL tree or red-black tree, since the tree restructuring they perform has to be done differently depending on whether the node is to be inserted or removed. However, `find()` is analogous to the splay operation on a splay tree, which restructures the tree during lookup. The cursor can still be used to insert or remove a node after the splay takes place.

3.6 Splay Tree

A splay tree, due to Sleator and Tarjan [106], is a self-adjusting binary search tree with the splay operation which moves a node to the root during traversal in a way that halves the distance to the root for all the nodes along the traversed path. The splay operation deals with zig-zig and zig-zag traversal differently, but both can be

expressed in terms of tree rotation. Splay tree has the "memory effect" where frequently accessed nodes are closer to the root and faster to access, a property which no other approximately balanced binary search tree has.

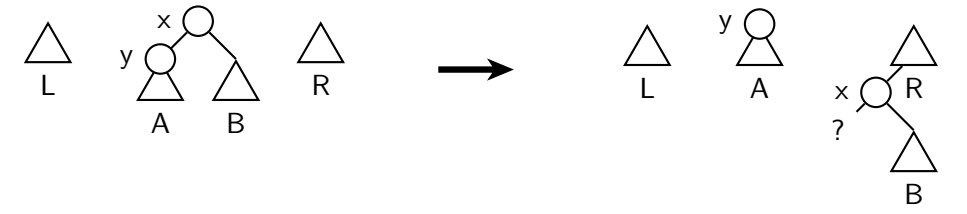
Splaying can be done in two ways, the bottom-up fashion where the tree structure is adjusted as the node ascends to the root, or the top-down fashion as shown in Figure 3.1 where the main tree is split into left and right subtree accumulators as the traversal descends towards the node, with the final tree consisting of the found node at the root with the accumulated left and right subtrees combined the node's left and right subtrees. In order to perform the final combination, an important ordering invariant (Proposition 9) must be maintained throughout. As the size of the size of the main tree diminishes, the left subtree grows towards the right, and the right subtree grows towards the left.

Proposition 9 (Top-Down Splay Invariant)**.** *During top-down splay, the left subtree accumulator nodes are all less than the nodes in the main tree, and the right subtree accumulator nodes are all greater than the nodes in the main tree.*

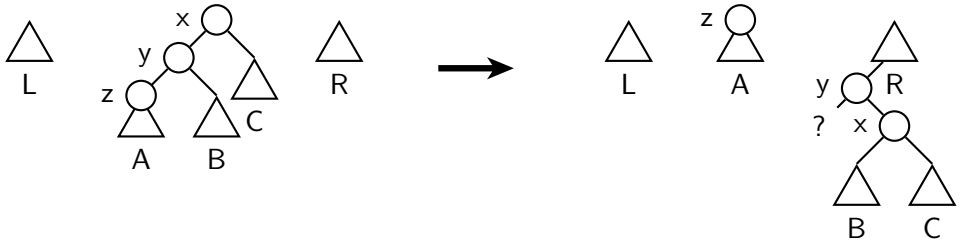
The accumulation of subtree is called *linking*. When linking occurs, the node to be accumulated is placed at the growth point, denoted by a question mark. Zig-zig also applies *rotation* before linking. Linking and rotation are the primitive splay tree operations. The top-down splay implementation here makes heavy use of cursor for tracking the growth point for linking, as well as the pivot point for rotation.

Listing 3.12: Splay tree operations.

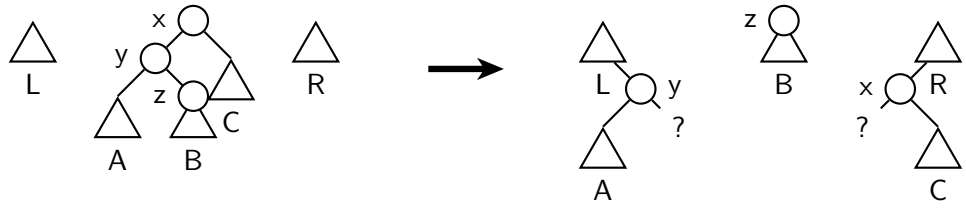
```
1 template<
2   class Ptr /* implements pointer<? implements binary_tree_node<Ptr> > */>
3 struct simple_splay_tree_operations
4     /* implements splay_tree_operations<Ptr> */ {
5   typedef Ptr ptr_type;
6
7   static void move(ptr_type& to, ptr_type& from) throw() {
```



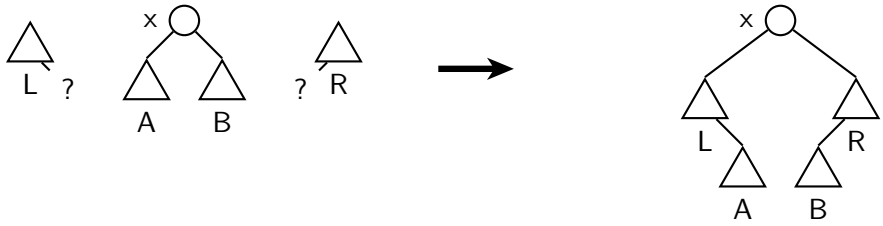
(a) zig



(b) zig-zig



(c) zig-zag



(d) final

Symmetric counterparts of zig, zig-zig and zig-zag are omitted.
Reproduced from [106] under fair use; permission pending.

Figure 3.1: Top-down splay.

```
8       to = from;
9     }
10
11    static void rotate_left(ptr_type& curr) throw() {
12      ptr_type x(curr), y(x->right());
13      x->right() = y->left();
14      y->left() = x;
15      curr = y;
16    }
17
18    static void rotate_right(ptr_type& curr) throw() {
19      ptr_type y(curr), x(y->left());
20      y->left() = x->right();
21      x->right() = y;
22      curr = x;
23    }
24
25    static ptr_type *link_left(ptr_type& curr, ptr_type *leftp) throw() {
26      *leftp = curr;
27      curr = (*leftp)->right();
28      return &(*leftp)->right();
29    }
30
31    static ptr_type *link_right(ptr_type& curr, ptr_type *rightp) throw() {
32      *rightp = curr;
33      curr = (*rightp)->left();
34      return &(*rightp)->left();
35    }
36  };
```

These operations can be implemented for augmented splay tree as well, in the same fashion as augmented linked list. Although these functions have been unit tested, the tests are omitted here. The reader is encouraged to try writing his own tests and find ways to deliberately break these functions and see if linear pointer can catch these breakages.

Based on the linking and rotation, we can implement `find()` and `find_limit()` with splaying as follows. Note that these functions are actually interchangeable with their binary search tree cousins.

Listing 3.13: Splay tree find implementations.

```
template<class Oper /* implements splay_tree_operations<?> */>
struct splay_tree_find_impl {
  typedef Oper oper_type;
  typedef typename oper_type::ptr_type ptr_type;
```

The `find()` function here is equivalent to the same function for binary search tree, except that top-down splaying is performed, which changes the structure of the tree. In the following code listing, the variables `left` and `right` are used to store the roots of the accumulated left and right subtrees, and `leftp` and `rightp` are the cursors within the left and right subtrees respectively where the linking should take place. These cursors always point to `nil` pointers.

```
  template<class Comparator /* implements comparable_with<Ptr> */>
  static ptr_type&
  find(Comparator& cmp, ptr_type& root) throw() {
    if (!root)
      return root;

    ptr_type left, right, curr(root);
    ptr_type *leftp = &left, *rightp = &right, *foundp = &root;

    do {
      ord_t dir = cmp.compare_to(curr);
      if (dir == LESS) {
        if (!curr->left()) {
          foundp = left? leftp : &curr->left();
          break;
        }

        ord_t dirl = cmp.compare_to(curr->left());

        if (dirl == LESS && curr->left()->left())
          oper_type::rotate_right(curr);

        rightp = oper_type::link_right(curr, rightp);

        if (dirl == GREATER && curr->right())
          leftp = oper_type::link_left(curr, leftp);
```

```
31
32        } else if (dir == GREATER) {
33          if (!curr->right()) {
34            foundp = right? rightp : &curr->right();
35            break;
36          }
37
38          ord_t dirr = cmp.compare_to(curr->right());
39
40          if (dirr == GREATER && curr->right()->right())
41            oper_type::rotate_left(curr);
42
43          leftp = oper_type::link_left(curr, leftp);
44
45          if (dirr == LESS && curr->left())
46            rightp = oper_type::link_right(curr, rightp);
47
48        } else /* if (dir == EQUAL) */ {
49          // foundp = &root;
50          break;
51        }
52      } while(true);
53
54      oper_type::move(*leftp, curr->left());
55      oper_type::move(curr->left(), left);
56      oper_type::move(*rightp, curr->right());
57      oper_type::move(curr->right(), right);
58      oper_type::move(root, curr);
59
60      return *foundp;
61    }
```

The trickier corner case arises when the equal node could not be found. By contract of the binary search tree `find()`, the function must return a cursor to the `nil` pointer where such node may be inserted. The handling is done on lines 17–20 as well as lines 33–36, with special consideration to prevent either cursor to the stack variables `left` and `right` leaking to the caller. Otherwise, these cursors would reference invalidated pointers after the function returns. Everything else is straightforward

adaptation of the pseudo-code presented by Sleator and Tarjan.

The cursor returned by `find()` can indeed be used for insertion just like binary search tree, but that inserts the node towards the bottom of the tree as opposed to the top. That is not how node insertion was originally proposed [106, §3], which splays the tree at the insertion point, splits the tree and uses the new node to join the two trees. The split is possible due to the top-down splay invariant in Proposition 9. The split and join insertion allows the node to be inserted at the root, which may be an advantage if the node is to be accessed immediately after, but splitting and joining also takes extra work. In practice, the performance difference is not obvious.

Here is the `find_limit()` function. Like the same function for binary search tree, the `foundp` cursor is used to tighten the bounds whenever a comparison occurs. It will point to `curr` most of the times, but it needs to be fixed upon linking and rotation. Also, `foundp` may point to the stack variables `left`, `right`, and `curr` respectively. This will be fixed at the end.

```
62  template<class Comparator /* implements comparable_with<Ptr> */>
63  static ptr_type&
64  find_limit(Comparator& cmp, ord_t limit, ptr_type& root) throw() {
65    if (!root)
66      return root;
67
68    ptr_type left, right, curr(root);
69    ptr_type *leftp = &left, *rightp = &right, *foundp = NULL;
70
71    ord_t dir;
72
73    do {
74      dir = cmp.compare_to(curr);
75      if (dir != limit || dir == EQUAL)
76        foundp = &curr;
77
78      if (dir == LESS) {
79        if (!curr->left())
80          break;
```

```
81
82          ord_t dirl = cmp.compare_to(curr->left());
83
84          if (dirl == LESS && curr->left()->left())
85            oper_type::rotate_right(curr);
86
87          if (foundp == &curr)
88            foundp = rightp;
89          rightp = oper_type::link_right(curr, rightp);
90
91          if (dirl == GREATER && curr->right()) {
92            if (dirl != limit)
93              foundp = leftp;
94            leftp = oper_type::link_left(curr, leftp);
95          }
96
97        } else if (dir == GREATER) {
98          if (!curr->right())
99            break;
100
101         ord_t dirr = cmp.compare_to(curr->right());
102
103         if (dirr == GREATER && curr->right()->right())
104           oper_type::rotate_left(curr);
105
106         if (foundp == &curr)
107           foundp = leftp;
108         leftp = oper_type::link_left(curr, leftp);
109
110         if (dirr == LESS && curr->left()) {
111           if (dirr != limit)
112             foundp = rightp;
113           rightp = oper_type::link_right(curr, rightp);
114         }
115
116       } else /* if (dir == EQUAL) */ {
117         break;
118       }
119     } while(true);
120
121     oper_type::move(*leftp, curr->left());
122     oper_type::move(curr->left(), left);
```

```
123       oper_type::move(*rightp, curr->right());
124       oper_type::move(curr->right(), right);
125       oper_type::move(root, curr);
126
127       if (foundp == &curr)
128         foundp = &root;
129       else if (foundp == &left)
130         foundp = &root->left();
131       else if (foundp == &right)
132         foundp = &root->right();
133       else if (foundp == NULL) {
134         if (limit == LESS)
135           foundp = &root->left();
136         else if (limit == GREATER)
137           foundp = &root->right();
138         assert(!*foundp);
139       }
140
141       return *foundp;
142     }
```

And this concludes the splay tree operations.

```
143 };
```

3.7 Conclusion

This chapter presents the implementation of data structures and algorithms that observe linear ownership semantics, and can be used with linear pointers introduced in Chapter 2. Unit testing is a powerful tool for finding programming error. A test is a short routine that exercising only a small pieces of the program at a time, making it easy to narrow down the scope of the error. We write unit tests to show that the implementation is functionally correct, as well as to show that linear pointer is able to find memory errors immediately when it happens. This justifies linear pointer as an effective technique for writing safe and correct programs.

The notion of cursor is developed as a programming technique to allow traversal of data structures without modifying linear pointers. It has the benefit of simplifying corner cases and reducing programming error. Both singly linked list and binary search tree make heavy use of cursor. A singly linked segment is built using singly linked list and a tail cursor. Such segment can pop from head and append to tail, which allows it to be used like a queue. An augmented linked list shows how cursors can be used for in-place modification. Quick sort and merge sort are the two sorting algorithms presented which shows how to use singly linked list and singly linked segment. In a strictly linear system, writing data structure traversal routines would require constanting taking node ownership from the data structure and returning it. This would make traversal extremely tedious. The cursor is a mechanism to relax the linear semantics, by observing that a reference to a linear pointer need not be linear.

Singly linked list and binary search trees are commonly used in a memory allocator. The singly linked list is used for keeping free objects in a segregate-fits strategy. The binary search tree is used for best-fit. Splay tree is a self-balancing binary search tree that avoids worst case search time. This chapter supplies the necessary data structures for a memory allocator based on an implementation that is verified to be safe using linear pointers.

Chapter 4

Design of the Memory Allocator

Back in the introduction, it was mentioned that linear pointer is motivated by the need of a memory management discipline that is simple, safe, and efficient. Linear pointer is a method to assist programmers correctly manage memory manually. The manual management stands in contrast to automatic memory management such as garbage collection. Although garbage collection takes the burden of memory management off the programmer and increases productivity, garbage collection can become a performance and scalability bottleneck which means greater cost in production. Performance penalty over millions of machines would dwarf the initial programmer productivity gain.

Throughout the dissertation, linear pointer is shown to be simple and safe. We argued that it is also efficient due to the erasure property, namely that once a program is tested with linear checking, then linear checking can be disabled without incurring any runtime overhead and still result in a correct program. But is it really efficient in practice?

To answer this question, this chapter describes a memory allocator built using linear pointers and linear data structures. The memory allocator is chosen for showing

the efficiency of linear pointer for two reasons: (1) there are many other extremely competitive memory allocators as a culmination of over three decades of research that can be used for the baseline of the comparison, and (2) the end result is a memory allocator that has no memory leak, thanks to the safety guarantee of linear pointer. Building such an allocator completes the full circle of the following rhetoric: what good is a program with no memory leak, if the memory allocator would leak memory on its behalf?

Our goal is not to exceed existing memory allocators in performance. After all, these allocators have undergone many years of profiling, optimization, and design evolution. Furthermore, the design here is inspired by those of existing allocators, so we do not expect performance breakthrough. The memory allocator here can be summarized as follows, and its construction will be described in more detail in the rest of the chapter.

Segregated object size class allocation. Upon allocation, object sizes are fit into size classes, and each class allocates objects from a different heap. A mixed of strategies are used by different size classes in order to control fragmentation. The packed heap classes use singly linked lists to index objects of the same size without header. Medium sized objects use best-fit heap classes with boundary tags [78, pp. 442] to delimit coalesceable objects, and objects are indexed using a top-down splay tree [106]. The size classes of medium sized objects progress in powers of 2, so that the worst worst-case fragmentation is bound to 3 times the used memory without paying the upfront cost of internal fragmentation. The large object heap class allocates memory in page sized units directly from virtual memory but uses a hash table of splay trees to keep track of allocated objects and their sizes.

Address space configuration. The allocator uses a fixed zone size of 2^{24} bytes, or 16MB, and zones are naturally aligned to this size. Each zone is dedicated to allocate heaps of a particular size. The allocator uses zones to allocate heaps sized 2^{14} through 2^{20} bytes in powers of two. The following table summarizes the relation between allocation strategy, object size classes, and heap size.

Strategy	Object Size Classes	Heap Size
Packed heap	$8 \ldots 64$ in multiples of 8	2^{14}
	$72 \ldots 128$ in multiples of 8	2^{15}
Best-Fit heap	$2^8 \ldots 2^{12}$ in powers of 2	$2^{16} \ldots 2^{20}$
	$2^{13} \ldots 2^{16}$ in powers of 2	2^{20}
Large allocation	>16KB in 4KB units.	N/A

Static preference in heap selection. Address space is divided into zones, and zones are divided into heaps appropriate of the object size class allocation strategy. The heaps are recycled as they are depleted and refilled. The bias prefers choosing a heap from the lowest address, which minimizes the spread of program objects throughout the address space. This improves locality and reduces memory footprint. The policy approximates first-fit at a global scale. A precedent of this strategy was mentioned in PhkMalloc [76].

Thread-local heaps. Lock contention and false sharing of cache line are two main performance hazards of a multi-threaded program. To avoid these problem, the allocator provides each thread with their private heaps so threads can manage their local memory without synchronizing with other threads. This design is prevalent in multi-threaded memory allocators such as Hoard [15] (2000–2009), TcMalloc [53] (2005–current), JeMalloc [48] (2006–current). To effectively deal with producer-consumer threads where an object is allocated in the producer thread, migrated to a consumer thread, and subsequently freed, a remote free mechanism is used as described by authors of the Streamflow allocator [100] (2006–2010). Both packed and best-fit heaps are thread local. Large allocation uses a shared hash table with fine-grained per-slot locks.

4.1 Zones and Zone Map

Zone is the coarsest level at which the allocator organizes address space. Each zone has a fixed size of 2^{24} bytes, or 16MB. Information about a zone is stored in a zone map, which is a dictionary mapping from zone number (or a rounded-down address) to some arbitrary value type. Each zone contains heaps of the same sizes but not necessarily the same type. Different zones may have different heap sizes. In our configuration, there are zones allocating heaps of sizes 2^{14} through 2^{20} bytes in powers of two. We store the exponent of the size in base 2 in the zone map.

Heaps in the zone are allocated in the fashion of a singly linked list in stack (last-in first-out) order. However, instead of using zone memory for the linked data structure, there is an external zone descriptor representing the linked list using an array of heap indices. In case the zone memory has already been swapped out, this allows us to check that all heaps are returned to the zone before we unmap the zone

memory without causing thrashing. Storing both zone map data structure and zone free list of heaps separately from program memory reduces the risk that program buffer overrun might corrupt zone metadata.

The zone map allows the heap header to be quickly located. Given the pointer to an object, if the zone map lookup succeeds and returns x, then the least x bits of the pointer is masked to yield the address of the heap header. All initialized heaps contain a heap header with a magic word which is chosen to be a pointer to the address of the magic word itself, since a circular reference like that is extremely rare in a normal program. If the magic word is valid, then the heap header may be used to determine heap type, free an object, and query object size. In order to keep things simple, we let heap sizes 2^{14} and 2^{15} to be used for packed heaps, and greater heap sizes to be used for best-fit heaps, so we can determine the heap type by heap size alone.

A naive implementation of the zone map might use the address space itself to do the mapping. Let each zone be prefixed by a zone header that contains the desired information about the zone. All heaps allocated from the zone would reserve space for this header, but only the first heap's zone header is used. To lookup information about the zone, simply truncate the lowest zone size bits of any pointer, which gives the address of the first zone header. This would work only if the address space contains nothing but zones. In reality, the address space also contains code, data, and stack. The allocator also maps large objects without using zones. Large objects are a challenge because zone lookup may crash the allocator when the program tries to free a legitimate object, unless large objects are allocated from zones dedicated to them, meaning that they would be limited to zone size.

It might be acceptable if a program crashes with access violation due to a zone lookup, in which case the zone is definitely invalid, but invalid zone lookup does not

guarantee a program to crash and therefore cannot be detected reliably. One way to improve the odds is by using the magic word technique to detect non-zone memory. The advantage of the naive approach is that it is extremely fast.

A reliable zone map would have to be stored in a separate data structure. For a machine architecture with 32-bit address space, there are 256 zones, so the zone map is small enough to maintain in an array. For a 64-bit machine, a shared hash table with individually locked splay tree buckets is used to maintain the zone map. To reduce overhead, each node in the splay tree contains zone records of several contiguous zones. To speed up the lookup, each thread keeps a thread-local cache of the zone map. The cache is a two-way set associative cache. Each cache line holds two mapping entries, using swap to maintain the property that one is the most recently used than the other. Upon eviction, the least recently used entry is replaced by the newly looked up mapping, but it is also promoted as the most recently used entry. A 32 entry two-way cache for 16MB zones will retain a maximum working set size of 1GB.

Cache invalidation is required when a zone is deleted from the map, but not when a zone is added. That is because the newly added zone will not be present in any zone map cache until it is accessed. However, when a zone is deleted, any cache that contains a mapping for that zone is no longer valid. The simple solution is to increment a version counter whenever a zone is deleted, and a thread would flush its zone map cache whenever it detects a change in the version counter. Zone deletion unmaps the virtual address occupied by the zone, which is an expensive operation as it involves Translate Look-aside Buffer (TLB) shoot-down coordinated by the operating system through inter-processor interrupts (IPI). Therefore, the cost of zone map cache flush is relatively insignificant. In a sense, zone map cache may be considered a shadow of the TLB.

In order to ensure that zones are naturally aligned to zone size, a virtual memory twice the size of the zone is requested first, and the excess portions are trimmed off. Although memory mapping needs are fulfilled by the operating system's virtual memory manager, this allocator maintains an mmap() cache layer to reduce the number of mapping and unmapping system calls that is a consequence of the alignment requirement of the zones. This cache layer maintains a small number of spans (there are 16 spans in our configuration) sorted in increasing address order.

For mapping, pages are allocated, similar to first-fit, from the lowest addressed span that could satisfy the request. If no span is large enough, the request is passed on to the operating system. This search is $O(n)$ where n is the number of spans. For unmapping, pages are merged to the adjacent spans found using binary search, in $O(\lg n)$ time. If no adjacent spans are found, the largest span is evicted, and the unmapped pages are inserted as a span using insertion sort, in $O(n)$ time. Since n is small, the cost is insignificant compared to the system call overhead that the span cache saves. The mmap() cache is shared by zone, large object allocation, as well as the memory backing the allocator's own data structures.

The mmap() cache reduces the cost associated with TLB shoot-down by reducing the number of unmap operations. When the OS unmaps memory, it must tell the processor to update its view of the address space by flushing the TLB. When there are threads running on multiple processors, the OS must ensure that all processors sharing the same address space flush their TLB, by issuing inter-processor interrupts and wait for the interrupt handlers on each processor flush their TLB. The TLB shoot-down is costly not only because priming the TLB after a flush slows down the program, but also the waiting.

4.2 Heaps and Pool Management

A heap specification consists of the implementation of the heap, which is either a packed heap or a best-fit heap, and a pool of heaps that provisions the heap from a specific zone type, and therefore sets the size of the heap. There is one heap specification for each object size class. The heap implementation is built on top of a base class defined as follows:

```
class heap_base
  : public safe_cast_base,
    public remote_free_access_impl {
 public:
  heap_base(const heap_info& info) throw()
    : info_(info) {}

  // Implements parts of heap_status_access
  volatile size_t& amount_collect() volatile throw() { return amount_collect_; }

  // Implements access to heap_info.
  const heap_info& info() const volatile throw() { return info_; }

 private:
  const heap_info& info_;
  size_t amount_collect_;
};
```

The base class `safe_cast_base` maintains the magic word for locating valid heap given an object pointer: its constructor initializes the magic word, and its destructor erases the magic word so that it is no longer valid. The `remote_free_access_impl` contains a linear atomic pointer to the first node of the remote free list, manipulated using non-blocking list insertion to be described in Section 5.6. The `heap_info` describes the characteristic of the heap: the size class index, heap size, and the object size it allocates. In addition, the `heap_base` also implements part of the heap status that tracks the number of bytes of the remote free objects. Together with the following

structure that tracks the number of bytes allocated in the thread-local fashion, a heap would implement the `heap_status_access` interface used by `heap_status<>` which wraps any heap implementation to account for the number of bytes used. The reason the interface has a split implementation is that we can align the shared and local parts of the heap base into different cache lines to reduce false sharing.

```
18 class heap_access_local {
19  public:
20   // implements parts of heap_status_access
21   size_t& amount_used() throw() { return amount_used_; }
22   size_t amount_used() const volatile throw() { return amount_used_; }
23
24  private:
25   size_t amount_used_;
26 };
```

Heaps are maintained in a recycle pool which indexes available heaps, which are not being used by any thread for allocation, but are candidates when a thread needs a heap. Since any heap can be remotely freed, a heap can replenish even when no thread has acquired it, so heaps cannot generally be indexed by availability. When a thread determines that a heap has depleted, it disposes the heap back into the pool and acquires another heap with available capacity. A reused available heap is obtained from the lowest address; therefore, all heaps are indexed by their addresses.

When a heap is released, the pool would scan the available heaps when the number of available heaps to the number of depleted heaps surpasses a certain ratio. The scan would set aside a small number of highest addressed heaps for examination and separate full and non-full heaps. A heap is full if no objects are allocated from the heap. The full heaps are released back to the zone. The non-full heaps are placed into a graduate heaps pool so they are not scanned again in a while. The graduate heaps would become available again when the available heaps are exhausted. Heap scan frequency and rate would affect the program's apparent memory usage as accounted

by the operating system, but this has not been studied in detail in this work. The nature of these parameters is like garbage collection: the allocator could do more work to tighten memory usage which sacrifices throughput, or to increase throughput by using more memory. The difference is that since pool management is outside of the critical path of object allocation, these overheads cause only minor impact on program performance.

It makes little sense to index depleted heaps because scanning them for availability would take more time as the program's actual memory usage grows. Instead, depleted heaps are temporarily removed from the pool and become self-owned, i.e. the heap contains a member pointer that owns itself. A remote free would touch the heap, and if the heap is self-owned, would check the heap's availability and reclaim the heap as an available heap if the amount of available bytes to the heap size reaches a certain ratio. As a potentially faster alternative, rather than accounting for a heap's availability, a touch would simply reclaim the heap when remote frees have taken place some number of times.

A heap in the pool can be indexed either in a binary search tree (splay tree) or as part of a singly linked list. The following node definition reuses the left pointer of the binary tree node for singly linked list.

```
27 template<class Ptr>
28 class heap_node
29   : public self_key<Ptr>,
30     public with_value<Ptr>,  // self-ownership
31     public binary_tree_node_impl<Ptr> {
32  private:
33   typedef Ptr ptr_type;
34
35  public:
36   // Implements singly_linked_node<Ptr> by reusing left() as next().
37   ptr_type& next() throw() { return this->left(); }
38   const ptr_type& next() const throw() { return this->left(); }
39   volatile ptr_type& next() volatile throw() { return this->left(); }
```

```
40    const volatile ptr_type& next() const volatile throw() {
41      return this->left();
42    }
43  };
```

The `self_key<Ptr>` base implements an ordered key based on the address of the heap. The node also carries a linear pointer for self-ownership of disposed heap.

The heap specification consists of the pointer type of the heap, the heap's implementation which is either packed or best fit, the heap access which defines the start and stop addresses of the heap, the final heap type, and the pool type. It looks roughly like this, with the concrete definition of the pool omitted.

```
44  template<
45    template<class Self> class Impl,
46    template<class Self> class Access>
47  struct heap_spec {
48    class heap_type;
49    typedef linear_atomic_ptr<heap_type> ptr_type;
50
51    typedef Impl<heap_type> impl_type;
52    typedef Access<heap_type> access_type;
53
54    class heap_type
55      : public heap_base,
56        public heap_node<ptr_type>,
57        public cache_aligned<heap_access_local>,
58        public access_type,
59        public heap_status<heap_type, impl_type> /* extends impl_type */ {
60     public:
61      heap_type(const heap_info& info) throw()
62        : heap_base(info) {}
63
64      using heap_node<ptr_type>::next;
65    };
66
67    class pool_type;
68  };
```

The resulting memory layout of the heap on a 64-bit machine would be similar to what the following table shows. We use `la_ptr<>` as an abbreviation for `linear_atomic_ptr<>` (see Section 5.3).

Offset	Type	Description
0x0000	uintptr_t	Magic word.
0x0008	la_ptr<free_list::node>	Remote free list.
0x0010	const heap_info&	Reference to heap description.
0x0018	size_t	Number of remote-freed bytes.
0x0020	la_ptr<heap_type>	Self-ownership.
0x0028	la_ptr<heap_type>	Binary tree left, or singly linked list next.
0x0030	la_ptr<heap_type>	Binary tree right.
...		Pad to cache line.
0x0040	size_t	Number of locally allocated bytes.
0x0048	void *	Heap memory start address.
0x0050	void *	Heap memory end address.
0x0058		Other heap-type specific fields.
...		Heap memory start.
...		Heap memory end.

4.3 Linearity Issues

Linear ownership semantics is all about resource management; in this case, the resource managed by the allocator is memory. The allocator is like a broker which obtains memory from the operating system in whole sale, divides them into smaller chunks, and distributes them to the application program. Linear pointers are used

from zones to heaps to individual objects in the heap where possible.

The memory allocator itself makes use of a metadata allocator for zone map nodes, zone descriptor, large object descriptor, as well as thread local descriptor. The metadata allocator is similar in principle to CustoMalloc [59] where an allocator is synthesized according to the object types used by the program. The synthesis takes place using C++ template instantiation of a class with a static member which is the homogeneous allocator of the desired object size. All instantiations for metadata objects of the same size will use the same allocator. The memory is backed by mmap() cache.

A zone consists of the memory obtained from the mmap() cache and the zone descriptor metadata which stores the allocation map for heaps within the zone. The association of the zone memory and zone descriptor cannot be checked by linear pointer, but we encapsulate this association using "zone factory" which contains the introduction and elimination functions for a zone. The introduction calls mmap(), sets the zone map according to the resulting address, and allocates the zone descriptor for that address. The elimination clears the zone map for the address, unmaps the zone, and frees the zone descriptor. When implementing the zone factory using a theorem-proving language, the introduction requires the linear proof transfer axiom $\text{mmap}(p) \otimes \text{zonemap}(p) \otimes \text{desc}(p)@q \multimap \text{zone}(p)@q$, where p is the zone's address, $\text{mmap}(p)$ is the linear proof obtained from mmap(), $\text{zonemap}(p)$ is the linear proof obtained from the zone map setter (so the caller does not forget resetting), $\text{desc}(p)@q$ is a zone descriptor obtained from the metadata allocator for zone at address p but the descriptor itself is at address q, and finally $\text{zone}(p)@q$ is the introduced token for zone at address p. The elimination requires the axiom in reverse: $\text{zone}(p)@q \multimap \text{mmap}(p) \otimes \text{zonemap}(p) \otimes \text{desc}(p)@q$.

After a zone is created, the zone's ownership is managed using linear pointer. A

zone becomes part of a homogeneous size allocator where zone memory is divided into equally sized blocks for heap allocation. Linear pointer does not check that zone memory division is complete and non-overlapping, but after a heap is initialized, the heap's ownership is managed by the pool using linear pointer. A heap would further sub-divide its memory for individual object allocation. The sub-division is again not verified by linear pointer, but the internal data structures for holding the free objects are. Because the legacy malloc() and free() interface does not preserve linear pointer, an object's linear pointer is destroyed before the program acquires it from malloc(); likewise, an object regains its linear pointer after the program releases it back to free().

4.4 Optimization

Serious performance degradation can result from seemingly benign C++ language constructs. In the course of optimizing the allocator, a number of such hazards have been identified. Many of the language constructs causing the hazard are tempting to use because of the simplicity and the limitations of the malloc() and free() interface. There is a single entry point for all the program's memory allocation needs, regardless of the thread, the locality of the object, and the size class. There is only one singleton instance of the allocator because there is only one address space. This single entry point has to dispatch the request to the appropriate heap. The dispatch starts from a global function call, routes through thread-local context, and finally arrives at the destination heap according to object size class. Optimizations described in this section can account for up to 20% performance improvement.

Static construction An object declared at the compilation-unit level, whether a static object or a global one (the only difference is that global objects have an external linkage), are initialized right after the program is loaded into memory but before the program starts running, but the order of initialization is undefined in terms of which compilation unit is initialized first. This often causes problems when a static object uses another one. As a result, the "construction on first use" idiom emerged [27, 10.15, 10.16], which wraps a static object inside a function like this:

```
class Thing {...};

Thing& thing() {
  static Thing the_thing;
  return the_thing;
}
```

C++ defers the construction of `the_thing` until `thing()` is first called. The actual code generated has to protect each access to `thing()` with an initialization guard like this:

```
static bool thing_init = false;

// Assume memory for the thing is well-aligned.
static char memory_for_the_thing[sizeof(Thing)];

Thing& thing() {
  if (thing_init)
    return *(Thing *) memory_for_the_thing;
  new (memory_for_the_thing) Thing();
  return *(Thing *) memory_for_the_thing;
}
```

The overhead of the initialization guard adds up when the functional units of the allocator are broken down into several classes. The "construction on first use" idiom is still useful in the presence of template instantiation because it is convenient to let the compiler generate static objects depending on the exact instantiation (e.g. for metadata allocator synthesis) as long as static object access is outside of the critical

path. Components in the critical path would be declared as members of the global allocator class. The global allocator class is created during shared library initialization and destroyed during shared library uninitialization.

```
class global_alloc {...};

static global_alloc *the_alloc = NULL;

__attribute__((constructor)) void libikai_malloc_init() {
  void *p = init_malloc(sizeof(global_alloc));
  the_alloc = new (p) global_alloc;
}

__attribute__((destructor)) void libikai_malloc_fini() {
  the_alloc->~global_alloc();
  the_alloc = NULL;
}
```

There is no need for initialization guard when accessing `the_alloc` because the construction and destruction times of the global allocator is well-defined. This technique is supported by ELF as well as Mac OS X. Similar technique is available on Windows by implementing `DllMain()`.

Thread local storage and dynamic linking ELF defines four models for accessing thread local storage [47]—general dynamic, local dynamic, initial exec, and local exec—in increasing order of performance at the expense of restriction, namely whether a function declared in one compilation unit could access the thread-local variable declared in a different one, and whether the function or the thread local storage could be dynamically loaded. Both general dynamic and local dynamic require several indirect lookups and address computations before the thread local variable could be accessed. Local exec, however, is too restrictive, as it requires the allocator to be statically linked to the executable. Since the memory allocator is made available to the program as a shared object loaded before the program runs, initial

exec is the most appropriate and provides good performance. A symptom of the bad model choice is that the `__tls_get_addr()` function gets called an excessive number of times, or that functions accessing thread local storage are slower than expected. Although Mac OS X does not support ELF-style thread local storage, the complexity of `pthread_getspecific()` is comparable to initial exec [88].

Another issue with dynamic linking is the resolution of symbol, which entails computing the symbol's address after load and populating a jump table to point to the address. For a shared library with many symbols but few of them actually used by the program, resolving symbols on program load is costly with no benefit, so they are instead resolved lazily on first use. The method is similar to the "construction on first use" idiom and requires a stub with initialization guard. On the other hand, the memory allocator only exposes a few functions, so it makes sense to precompute all symbols on load. This avoids the initialization guard overhead as well as an additional indirect branch. During profiling, excessive number of cycles being attributed to the `_dl_runtime_resolve()` function indicates performance problem with the lazy symbol resolution.

Although these issues have grave performance impact, addressing them is delightfully simple: just pass the right command line flags when building. For ELF thread local storage, pass `-ftls-model=initial-exec` to the compiler; for eager symbol resolution, pass `-l now` to the linker.

Indirect branches With dynamic linking, there is always at least one level of indirect branch in order to call a function in a shared library whose symbols are resolved in the run time. Indirect branches are also prevalent where virtual methods are used as well as a switch statement with many cases. Indirect branching is a performance hazard because it stalls a processor's instruction execution pipeline. Modern processors

fetches, decodes, executes, and store the results of instructions in multiple stages for performance. Branching can stall the pipeline because the destination might depend on the result of prior computation. Conditional branches can be speculatively evaluated because there are only two possible branches. With indirect branches, branch prediction is hard because the destination is unknown. If a processor has a 20 stage pipeline, indirect branch could cost 20 cycles on a misprediction.

Since there is only a small number of allocation strategies in use, indirect branches can be replaced by conditional branches during dynamic dispatch. Furthermore, small objects are favored for being the most likely in branch prediction because misprediction penalty is greater for them. While dynamic method resolution using virtual table is one way to achieve polymorphism, it can be replaced by curiously recurring template pattern [32] which is a compile-time type-based static method resolution mechanism.

4.5 Benchmark and Results

The memory allocator designed here, given the name of "Libikai," is compared with other allocators in terms of performance and memory use. The other allocators are GNU Glibc allocator, Hoard, JeMalloc, StreamFlow, and TcMalloc. Glibc is based on PtMalloc2, which is a parallel adaptation of Doug Lea's allocator. Furthermore, there are two flavors of Libikai, one that has linearity checks turned on (denoted as "libikai$^{-\circ}$"), and one has not. Both flavors were compiled without other assertion checks. This allows us to see the actual performance difference caused by linearity checking. The precise versions used for these allocators are listed in Table 4.3. They were all compiled using GCC 4.3.6 with the exception of Glibc which was compiled using GCC 4.7.3.

Allocator	Version
GNU Glibc	Ubuntu EGLIBC 2.17-0ubuntu5
Hoard	git:4f5b5ab (Aug 5, 2013)
JeMalloc	git:0ed518e (Jun 2, 2013)
Libikai	git:1acaa1f (Aug 30, 2013)
StreamFlow	git:fa2c3c5 (Mar 9, 2013)
TcMalloc	git:819a2b0 (Aug 29, 2013)

Table 4.3: Allocator versions used in the benchmark.

The benchmark designed for this purpose is a program that takes an allocation pattern configuration and mimics the allocation behavior of another program. The configuration is an object size histogram with lifetime quantile of each object size. The program runs in a loop. In each cycle, it first determines the desired object size by sampling the histogram, then its lifetime by interpreting the quantile as a cumulative distribution function. Once an object is allocated, it is placed into a circular array holding singly linked lists. The program maintains a cycle counter t which is the index into this array. Objects with lifetime l is placed into the array index $(t + l)$ mod n where n is the array size. Objects in the singly linked list at $t \mod n$ are freed at the beginning of cycle t.

There are two variants of this benchmark, one that runs independent allocation and free over multiple threads ("no exchange"), and one where threads also send the objects they allocated to any one of the other threads using the same non-blocking singly linked list insertion as remote free ("exchange"). In the latter variant, the object's lifetime is stored as the first word of the object. At each cycle, the thread would additionally check its receiving list of objects and store them into the array by their lifetime. This specially crafted benchmark is designed to stress test Hoard, StreamFlow, and Libikai whose design goal is to make producer-consumer allocation pattern efficient. Note that this benchmark has no speed-up to speak of. Each thread

Variant	Alloc	Config	# Threads			
			8	16	24	32
mt	hoard	bestbuddy	17	19	20	20
		bestbuddy+1	5	18	20	20
mtnx	hoard	bestbuddy	8	19	20	20
		bestbuddy+1	14	19	20	20

Table 4.4: Error count.

performs the same amount of work, so the amount of work increases proportionally to the number of threads.

The machine used for the benchmark was an Amazon EC2 virtual machine of instance type cc2.8xlarge, which provided two 8-core hyper-threaded Intel Xeon E5-2670 running at 2.60GHz, for a total of 32 virtual CPUs. A total of 58GB memory was available to use by the virtual machine; it has no swap memory configured. The machine was running Ubuntu Linux 13.04 at kernel 3.8.0-19-generic. Each test was run 20 times total. Some of the tests failed, and each failed iteration was retried for up to three times. Table 4.4 shows the error counts for the particular configuration and number of threads. Only Hoard resulted in any errors. The values in the table are the means and standard deviations over the successful runs. Wall-clock time and user CPU time are measured in seconds. Memory usage is measured in bytes.

4.5.1 Packed Size Classes

We first present the benchmark result against an object size configuration that only includes every packed heap size classes for our allocator, which are sizes 8 though 128 in multiples of 8. With the exception of Glibc, all the other allocators use the same headerless packed heap strategy for these object sizes with minor differences in size classification granularity. Glibc (based on Doug Lea's allocator) uses segregated free list for approximated best-fit allocation. Table 4.5 shows the wall-clock time spent

running the benchmark, comparing the "no exchange" and the "exchange" variants.

With "no exchange," Libikai with linearity checks runs from 12% to 37% slower than with linearity checks turned off, with the disparity increasing as the number of threads increases. A plausible explanation is hyper-threading. The additional linearity check puts more work on the computation core which is shared every two threads. As the number of threads increase, the likelihood that two threads contend with the same core also increases. However, "exchange" is more bottlenecked by the communication overhead between cores, so the effect of linearity checking is diminished and even reversed at higher thread counts.

With "no exchange," Libikai is slower than StreamFlow and TcMalloc but is faster than Glibc, Hoard and JeMalloc. StreamFlow is the fastest. With "exchange," Libikai is the fastest. It is surprising that StreamFlow which implements remote free does not show a significant edge against TcMalloc which does not. Overall, it is not surprising to see that "exchange" takes more time than "no exchange" for all allocators. Glibc and Hoard especially struggle with exchange compared to their own performances against "no exchange."

A more interesting picture emerges when considering the utilization which is the sum of system and user CPU time divided by wall-clock time and the number of threads, as shown in Figure 4.1. With "no exchange," except for Glibc whose utilization plummets as the number of threads, all allocators are able to remain highly utilized, as expected with thread-local heap design. With "exchange," only StreamFlow and Libikai retain high utilization due to remote free. Remote free is only able to finish sooner due to the increased utilization despite using more CPU time than other allocators. The user CPU time is shown in Table 4.6.

Table 4.7 compares the maximum resident set size with "no exchange" and "exchange." In our setup, this value is the same as peak memory usage because the

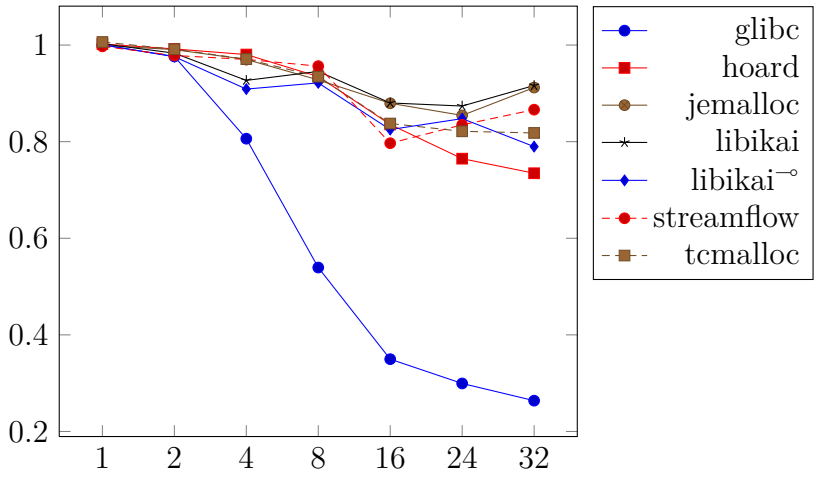
Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.223	0.230	0.334	0.553	0.845	0.976	1.128
±	0.007	0.008	0.030	0.038	0.072	0.036	0.057
hoard	0.319	0.322	0.340	0.397	0.558	0.699	0.920
±	0.006	0.005	0.006	0.034	0.048	0.034	0.045
jemalloc	0.202	0.201	0.210	0.230	0.321	0.474	0.729
±	0.008	0.007	0.004	0.026	0.017	0.032	0.034
libikai	0.228	0.286	0.319	0.329	0.393	0.485	0.550
±	0.006	0.024	0.009	0.012	0.041	0.021	0.016
libikai$^{-\circ}$	0.258	0.324	0.377	0.391	0.490	0.578	0.756
±	0.008	0.036	0.022	0.013	0.051	0.014	0.043
streamflow	0.225	0.234	0.231	0.233	0.300	0.357	0.409
±	0.006	0.005	0.006	0.003	0.031	0.010	0.016
tcmalloc	0.202	0.202	0.211	0.226	0.285	0.363	0.427
±	0.006	0.007	0.004	0.013	0.027	0.010	0.012

(a) “no exchange”

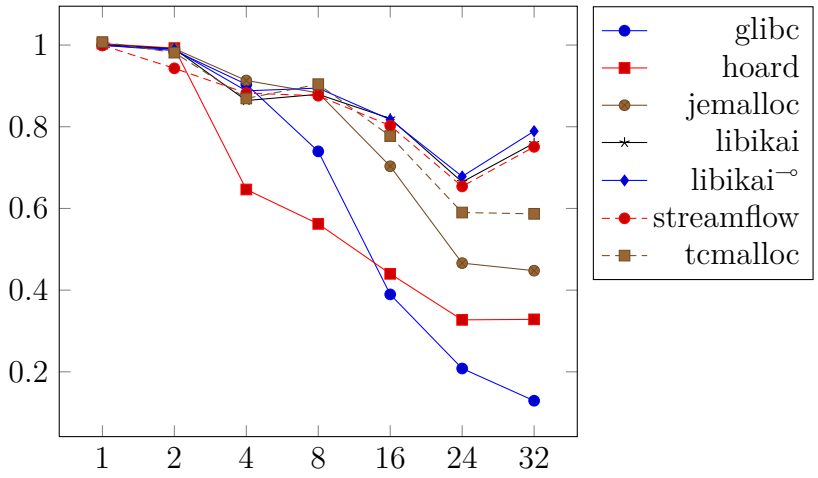
Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.247	0.655	0.962	1.472	3.659	7.689	13.23
±	0.008	0.178	0.038	0.048	0.262	0.475	0.827
hoard	0.343	0.581	1.230	1.541	2.300	4.388	5.923
±	0.005	0.081	0.101	0.192	0.375	0.404	1.418
jemalloc	0.222	0.401	0.659	0.799	1.237	2.196	2.167
±	0.006	0.081	0.016	0.027	0.120	0.096	0.102
libikai	0.249	0.522	0.665	0.857	1.156	1.549	1.443
±	0.006	0.110	0.065	0.016	0.041	0.048	0.062
libikai$^{-\circ}$	0.285	0.562	0.713	0.825	1.078	1.521	1.315
±	0.006	0.104	0.011	0.028	0.068	0.066	0.049
streamflow	0.248	0.556	0.734	0.886	1.206	1.693	1.504
±	0.007	0.103	0.020	0.024	0.067	0.077	0.065
tcmalloc	0.225	0.375	0.600	0.774	1.099	1.635	1.584
±	0.005	0.087	0.076	0.009	0.026	0.045	0.032

(b) “exchange”

Table 4.5: Wall-clock time for packed configuration.



(a) “no exchange”



(b) “exchange”

Figure 4.1: Utilization for packed configuration.

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.219	0.433	1.054	2.357	4.683	6.965	9.472
±	0.005	0.008	0.061	0.100	0.163	0.106	0.373
hoard	0.314	0.627	1.292	2.707	6.247	10.65	16.03
±	0.006	0.008	0.028	0.083	0.605	0.362	1.093
jemalloc	0.199	0.390	0.795	1.627	3.549	6.907	10.48
±	0.008	0.013	0.015	0.063	0.248	0.164	0.173
libikai	0.226	0.560	1.175	2.471	5.507	10.09	16.04
±	0.006	0.043	0.038	0.037	0.090	0.149	0.193
libikai$^{-\circ}$	0.256	0.628	1.362	2.866	6.436	11.71	19.01
±	0.010	0.066	0.036	0.037	0.115	0.125	0.262
streamflow	0.223	0.452	0.888	1.765	3.789	7.084	11.22
±	0.007	0.009	0.020	0.017	0.100	0.165	0.203
tcmalloc	0.201	0.394	0.809	1.666	3.758	7.049	11.02
±	0.006	0.013	0.014	0.036	0.186	0.170	0.134

(a) “no exchange”

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.244	1.274	3.437	8.642	22.58	37.84	53.59
±	0.008	0.349	0.387	0.462	0.349	0.621	0.894
hoard	0.339	1.141	3.043	6.638	15.20	27.57	39.87
±	0.007	0.161	0.085	0.213	0.756	0.402	1.439
jemalloc	0.219	0.790	2.393	5.611	13.83	24.32	30.72
±	0.006	0.161	0.163	0.158	0.115	0.320	0.277
libikai	0.247	1.031	2.283	6.006	15.11	24.58	34.93
±	0.005	0.216	0.338	0.282	0.691	0.245	0.357
libikai$^{-\circ}$	0.283	1.110	2.518	5.883	14.05	24.63	33.04
±	0.006	0.203	0.265	0.326	0.202	0.296	0.530
streamflow	0.245	1.043	2.579	6.177	15.43	26.44	36.02
±	0.007	0.193	0.333	0.426	0.863	0.759	0.402
tcmalloc	0.224	0.731	2.069	5.571	13.59	22.93	29.42
±	0.005	0.169	0.333	0.099	0.130	0.246	0.194

(b) “exchange”

Table 4.6: User CPU time for packed configuration.

machine has no swap space. Also, the number of objects allocated and retained per thread is constant, so total memory usage increases as the number of threads increases. With "no exchange," Glibc is the most memory efficient as Doug Lea's allocator is reputed to be. All other allocators use similar amounts of memory. With "exchange," Glibc experiences a blow up of up to 7 times. Although Hoard's design goal is to prevent blow-up with producer-consumer pattern exhibited by "exchange," at 5 times blow-up it is clearly ineffective. The most effective memory use is StreamFlow and Libikai at less than 2 times blow-up. Memory usages of Libikai with and without linearity check are statistically indistinguishable.

4.5.2 Lower Best-Fit Size Classes

The benchmark is run against a configuration of object sizes $2^b \cdot k$ for $b \in \{6, 7, 8\}$ and $k \in \{4, 5, 6, 7\}$, as well as 2^{11}, which spans the lower best-fit size classes 2^8 through 2^{11} for our allocator. At these size class, all other allocators still use the same respective strategy as described previously. This is mainly a benchmark comparing packed and best-fit strategies. Table 4.8 shows the wall-clock time. Table 4.9 shows the user CPU time. Best-fit with splay tree operation uses significantly more CPU time than packed strategy which uses only singly linked lists, although the wall-clock time continues to be dominated by utilization as shown in Figure 4.2. Splay tree rotation uses many more pointer moves than singly linked list, so the difference of Libikai with and without linear checking is larger, between 70% to 130%.

Table 4.10 shows the memory usage. Due to the small number of object sizes in this configuration, the space savings of best-fit strategy is not apparent. The memory saving reported in the "exchange" variant is similar to the packed heap size classes, mostly due to remote-free which allows objects to move freely between threads for

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	3.17M	6.33M	12.6M	25.3M	49.6M	61.1M	75.8M
±	0	0	0	0	1.62M	0.98M	3.47M
hoard	3.35M	6.68M	13.3M	26.6M	53.2M	79.9M	105M
±	0	1.47K	8.44K	21.1K	36.2K	39.5K	1.81M
jemalloc	3.06M	5.91M	11.6M	23.1M	45.9M	68.7M	91.9M
±	0	8.54K	36.5K	50.7K	78.1K	83.9K	895K
libikai	3.41M	6.57M	12.8M	25.1M	49.7M	74.3M	98.8M
±	0	22.3K	40.0K	68.3K	69.8K	111K	112K
libikai$^{-\circ}$	3.46M	6.59M	12.8M	25.2M	49.8M	74.5M	98.8M
±	0	40.0K	36.4K	50.8K	82.5K	116K	135K
streamflow	2.69M	5.61M	11.4M	22.9M	46.0M	69.2M	91.7M
±	3.87K	3.55K	15.4K	71.4K	191K	220K	353K
tcmalloc	2.98M	5.93M	11.9M	23.9M	47.7M	71.4M	95.3M
±	3.58K	10.3K	30.3K	32.5K	59.0K	70.5K	116K

(a) “no exchange”

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	3.16M	6.88M	25.5M	50.4M	144M	314M	551M
±	0	800K	15.2M	16.7M	11.1M	12.5M	26.4M
hoard	3.36M	7.02M	44.0M	74.4M	152M	375M	513M
±	0	232K	7.81M	24.6M	46.1M	36.0M	133M
jemalloc	3.08M	6.35M	21.1M	37.2M	100M	256M	281M
±	0	115K	11.2M	12.4M	28.8M	26.0M	23.0M
libikai	3.41M	6.60M	30.6M	44.6M	92.0M	218M	155M
±	0	24.5K	12.9M	16.7M	24.4M	14.6M	18.5M
libikai$^{-\circ}$	3.45M	6.63M	23.9M	38.1M	80.9M	201M	120M
±	0	34.2K	12.9M	19.8M	25.6M	21.7M	21.7M
streamflow	2.71M	9.35M	24.2M	43.4M	82.2M	187M	133M
±	2.25K	1.44M	10.5M	14.4M	19.2M	13.9M	12.9M
tcmalloc	3.01M	6.33M	24.5M	30.6M	85.7M	213M	208M
±	7.52K	509K	11.9M	4.82M	10.2M	13.0M	10.1M

(b) “exchange”

Table 4.7: Maximum resident set size for packed configuration.

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.320	0.755	2.611	5.038	10.04	14.69	19.87
±	0.007	0.138	0.129	0.112	0.197	0.603	0.434
hoard	1.034	1.061	1.213	1.652	2.427	3.526	5.260
±	0.006	0.014	0.152	0.118	0.231	0.322	0.982
jemalloc	0.246	0.252	0.284	0.326	0.515	0.735	1.225
±	0.006	0.006	0.025	0.030	0.034	0.075	0.239
libikai	0.516	0.669	0.740	0.826	1.031	1.301	1.630
±	0.006	0.057	0.012	0.049	0.079	0.027	0.175
libikai$^{-\circ}$	1.155	1.359	1.515	1.629	1.970	2.473	2.704
±	0.010	0.085	0.044	0.163	0.296	0.084	0.045
streamflow	0.302	0.314	0.333	0.376	0.505	0.625	0.759
±	0.005	0.005	0.007	0.050	0.050	0.027	0.049
tcmalloc	0.223	0.234	0.255	0.306	0.487	0.813	1.113
±	0.002	0.010	0.005	0.024	0.035	0.042	0.117

(a) “no exchange”

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.365	1.879	4.075	9.492	23.94	40.42	58.13
±	0.004	0.411	0.546	1.911	3.039	6.526	4.369
hoard	1.046	1.428	2.233	2.833	4.220	7.748	7.679
±	0.007	0.006	0.248	0.140	0.461	0.506	0.482
jemalloc	0.275	0.502	0.847	1.125	1.848	3.633	3.718
±	0.005	0.067	0.188	0.192	0.336	0.187	0.218
libikai	0.554	0.804	1.072	1.219	1.608	2.046	1.962
±	0.006	0.134	0.021	0.015	0.136	0.042	0.075
libikai$^{-\circ}$	1.186	1.526	1.764	1.948	2.282	2.989	3.086
±	0.007	0.126	0.021	0.100	0.192	0.048	0.089
streamflow	0.338	0.532	0.715	1.035	1.386	2.155	1.669
±	0.008	0.117	0.187	0.092	0.146	0.153	0.053
tcmalloc	0.250	0.441	0.723	0.868	1.343	2.145	2.080
±	0.005	0.083	0.055	0.053	0.076	0.130	0.118

(b) “exchange”

Table 4.8: Wall-clock time for lower best-fit configuration.

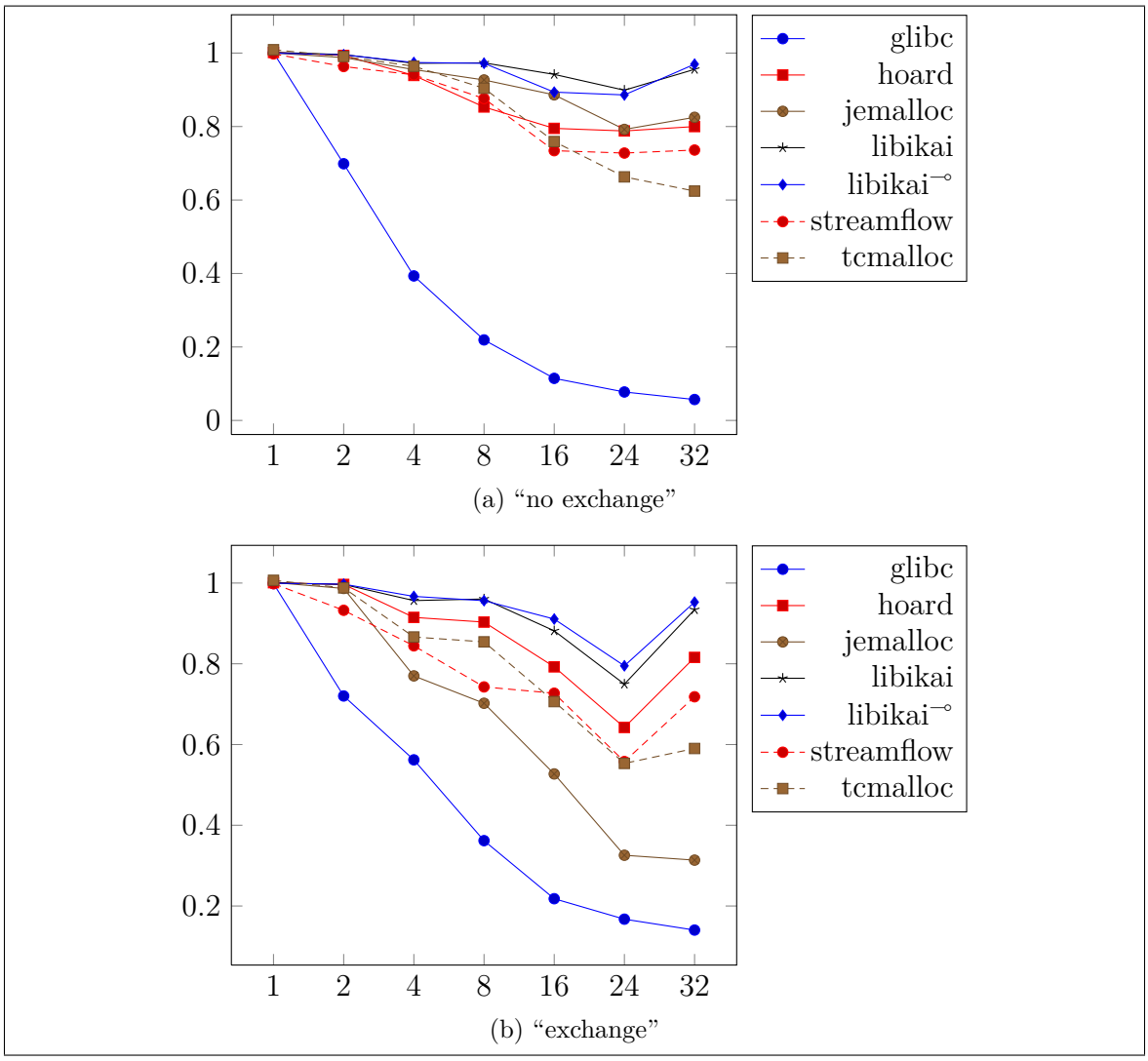
Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.294	1.028	4.017	8.714	17.97	26.58	34.70
±	0.007	0.114	0.261	0.154	0.321	0.460	0.654
hoard	0.949	1.950	4.195	9.257	19.57	34.02	54.46
±	0.018	0.046	0.267	0.257	0.978	0.896	1.010
jemalloc	0.222	0.448	0.972	2.074	4.815	9.381	13.91
±	0.008	0.018	0.053	0.078	0.238	0.364	0.513
libikai	0.497	1.293	2.797	6.258	15.16	27.39	48.65
±	0.007	0.115	0.062	0.096	0.185	0.412	5.433
libikai$^{-\circ}$	1.135	2.662	5.828	12.51	27.82	51.91	82.86
±	0.013	0.167	0.061	0.289	0.527	1.376	0.950
streamflow	0.281	0.563	1.152	2.433	5.450	9.983	16.56
±	0.006	0.011	0.025	0.164	0.484	0.225	0.341
tcmalloc	0.205	0.425	0.904	2.041	5.042	10.57	17.38
±	0.007	0.022	0.023	0.060	0.156	0.251	0.604

(a) “no exchange”

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.334	1.246	2.492	6.981	23.23	54.24	93.40
±	0.009	0.292	0.609	0.993	1.679	2.466	2.975
hoard	0.971	2.473	6.855	16.11	34.39	59.49	87.15
±	0.015	0.033	0.673	0.379	0.852	1.363	0.799
jemalloc	0.250	0.950	2.407	5.934	14.64	25.73	33.82
±	0.008	0.139	0.222	0.199	0.189	0.288	0.316
libikai	0.533	1.559	4.019	9.199	22.23	35.46	57.30
±	0.007	0.267	0.161	0.093	0.104	0.346	1.617
libikai$^{-\circ}$	1.164	3.007	6.740	14.72	32.82	55.67	92.87
±	0.012	0.249	0.179	0.093	0.177	0.805	1.443
streamflow	0.320	0.939	2.285	5.812	15.45	27.17	37.25
±	0.009	0.208	0.453	0.562	0.175	0.343	0.442
tcmalloc	0.232	0.838	2.393	5.718	14.39	25.98	35.33
±	0.008	0.166	0.167	0.167	0.126	0.560	0.637

(b) “exchange”

Table 4.9: User CPU time for lower best-fit configuration.



(a) “no exchange”

(b) “exchange”

Figure 4.2: Utilization for lower best-fit configuration.

better reuse.

4.5.3 Upper Best-Fit Size Classes

At object sizes starting from 2^{12}, Hoard would allocate them from mmap(). JeMalloc and StreamFlow starts allocating objects 2^{12} and larger using a binary buddy algorithm in powers of two pages. StreamFlow also stopped using remote free; each allocation and deallocation also involves modifying the global BIBOP (big bag of pages) map. TcMalloc still uses packed heap with coarser spacing for size classes up to 32K (it has a total of 170 such size classes). For larger sizes it uses a singly linked list of page spans. There is no change in the allocation strategy of Glibc (based Doug Lea's allocator); it still uses binned free list with approximated best-fit allocation. This benchmark tests against object sizes 2^{12} to 2^{15} in powers of two.

The performance readily reflects the differences of implementation strategy. Table 4.11 shows the wall-clock time. StreamFlow becomes even slower than Glibc at some of the higher thread counts. JeMalloc also takes a significant hit in performance penalty. Using the same best-fit strategy, Libikai is now faster than these allocators, although it is still no match for TcMalloc whose strategy proves to be the most resilient. Hoard begins to experience segmentation fault at 8 threads or more.

Utilization continues to dominate wall-clock time, as seen in Figure 4.3. User CPU time in Table 4.12 reveals that best-fit with remote-free actually uses significantly more CPU time than JeMalloc and TcMalloc. Libikai is only able to compensate by having higher utilization.

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	30.9M	61.9M	123M	246M	466M	646M	806M
±	0	0	0	117K	5.13M	8.20M	7.06M
hoard	37.1M	74.1M	147M	295M	590M	884M	1.15G
±	0	93.4K	233K	223K	676K	500K	8.05M
jemalloc	31.7M	63.3M	126M	252M	506M	758M	0.99G
±	0	4.73K	63.4K	136K	474K	565K	504K
libikai	37.1M	72.9M	144M	287M	571M	853M	1.11G
±	0	250K	461K	623K	1.04M	1.31M	1.48M
libikai$^{-\circ}$	36.5M	72.8M	144M	287M	571M	851M	1.11G
±	0	215K	454K	497K	873K	1.03M	1.23M
streamflow	35.4M	70.5M	141M	283M	567M	851M	1.11G
±	3.07K	107K	99.6K	328K	505K	1.02M	870K
tcmalloc	32.0M	63.5M	126M	253M	505M	754M	0.98G
±	0	55.8K	100K	147K	346K	328K	319K

(a) “no exchange”

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	30.9M	216M	559M	1.39G	2.09G	2.25G	2.25G
±	0	100M	326M	519M	700M	681M	222M
hoard	37.1M	74.1M	220M	338M	805M	2.35G	1.20G
±	0	139K	84.2M	64.6M	250M	129M	73.9M
jemalloc	31.8M	65.9M	347M	633M	1.36G	3.62G	3.71G
±	0	1.91M	241M	299M	483M	291M	340M
libikai	37.0M	72.9M	165M	292M	721M	2.02G	1.11G
±	0	275K	53.0M	18.6M	244M	79.6M	1.08M
libikai$^{-\circ}$	37.0M	72.8M	149M	303M	683M	2.07G	1.11G
±	0	168K	8.92M	71.9M	222M	42.4M	975K
streamflow	35.4M	108M	270M	727M	1.14G	2.64G	1.37G
±	3.33K	20.8M	150M	273M	329M	213M	62.5M
tcmalloc	32.0M	65.3M	213M	363M	915M	2.04G	1.33G
±	3.58K	4.29M	118M	144M	181M	155M	178M

(b) “exchange”

Table 4.10: Maximum resident set size for lower best-fit configuration.

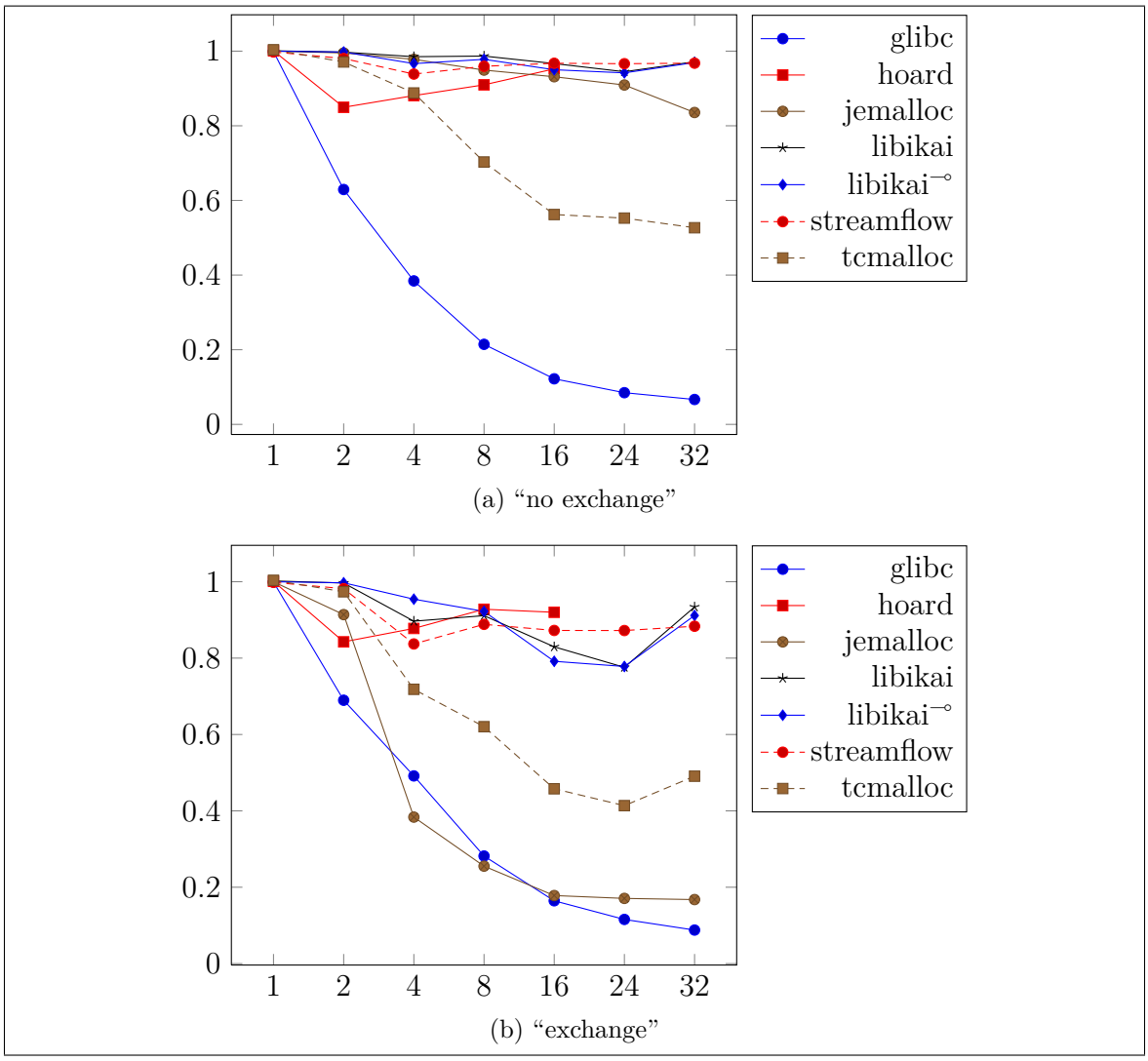
Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.638	2.031	7.971	17.80	38.42	63.50	91.60
±	0.005	0.387	0.443	0.512	0.847	1.695	3.269
hoard	1.446	3.274	6.625	14.24*	33.34*	-	-
±	0.006	0.296	0.276	0.245*	0.000*	-	-
jemalloc	0.637	0.871	1.230	2.859	11.24	23.65	38.11
±	0.007	0.012	0.023	0.179	0.605	0.675	1.219
libikai	0.531	0.666	0.889	1.463	5.221	12.04	25.38
±	0.009	0.019	0.013	0.026	0.146	0.253	0.513
libikai$^{-\circ}$	0.809	0.990	1.233	1.676	6.124	14.95	30.27
±	0.007	0.018	0.089	0.039	0.163	0.276	0.452
streamflow	0.504	0.970	2.872	11.61	45.67	99.54	181.9
±	0.006	0.225	0.485	0.813	1.324	1.499	3.966
tcmalloc	0.377	0.475	0.656	1.414	3.305	5.216	8.189
±	0.007	0.006	0.021	0.154	0.118	0.122	0.173

(a) “no exchange”

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.693	3.577	8.455	29.69	76.84	148.2	241.7
±	0.014	0.795	1.729	4.006	14.25	19.76	17.54
hoard	1.444	3.739	7.905	13.65*	60.34*	-	-
±	0.014	0.358	2.640	0.071*	0.000*	-	-
jemalloc	0.666	1.360	7.553	17.73	45.67	71.58	100.4
±	0.007	0.115	1.213	3.567	7.434	13.44	10.77
libikai	0.572	0.878	1.293	1.798	5.051	10.65	22.83
±	0.006	0.081	0.215	0.274	0.252	0.426	0.530
libikai$^{-\circ}$	0.840	1.139	1.513	1.951	6.355	13.51	27.36
±	0.007	0.079	0.088	0.242	0.417	0.558	0.760
streamflow	0.534	0.964	3.356	13.53	50.61	112.0	199.7
±	0.005	0.158	1.335	0.917	1.118	2.075	3.880
tcmalloc	0.401	0.711	1.363	2.259	5.112	9.189	8.864
±	0.007	0.079	0.368	0.497	0.627	0.904	0.326

(b) “exchange”

Table 4.11: Wall-clock time for upper best-fit configuration.



(a) “no exchange”

(b) “exchange”

Figure 4.3: Utilization for upper best-fit configuration.

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.395	1.800	9.241	23.15	51.52	76.82	96.82
±	0.020	0.183	0.604	0.679	2.240	1.662	2.256
hoard	1.042	4.320	8.920	14.52*	30.83*	-	-
±	0.034	0.127	0.282	0.530*	0.000*	-	-
jemalloc	0.533	1.214	2.592	5.408	12.92	24.69	37.15
±	0.017	0.037	0.088	0.221	0.289	0.290	0.681
libikai	0.464	1.188	3.204	10.90	78.82	268.7	780.5
±	0.016	0.038	0.082	0.196	3.694	5.731	13.53
libikai$^{-\circ}$	0.736	1.836	4.474	12.49	91.30	333.7	930.4
±	0.010	0.038	0.152	0.493	3.379	6.071	14.14
streamflow	0.426	1.738	10.46	88.54	705.7	2306	5631
±	0.012	0.443	1.902	7.003	26.57	48.76	149.3
tcmalloc	0.313	0.818	2.213	6.650	16.86	28.25	40.15
±	0.012	0.017	0.059	0.315	0.463	0.440	0.646

(a) “no exchange”

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	0.436	1.184	2.929	11.28	40.82	103.7	188.9
±	0.023	0.381	0.956	1.528	4.772	9.539	8.887
hoard	1.037	4.858	10.52	24.97*	38.44*	-	-
±	0.030	0.278	0.269	2.492*	0.000*	-	-
jemalloc	0.563	1.805	7.442	20.70	62.10	104.3	138.5
±	0.013	0.113	0.732	3.805	8.625	14.96	8.764
libikai	0.499	1.604	4.271	12.31	63.81	186.6	673.4
±	0.011	0.163	0.280	0.465	6.516	5.996	23.74
libikai$^{-\circ}$	0.765	2.131	5.460	13.67	76.26	240.6	785.9
±	0.012	0.159	0.134	0.238	4.637	12.20	24.13
streamflow	0.456	1.742	10.61	94.57	701.6	2336	5634
±	0.012	0.318	2.918	5.849	20.74	62.89	176.4
tcmalloc	0.337	1.283	3.562	10.17	27.57	48.07	60.91
±	0.011	0.130	0.058	0.591	0.972	1.857	1.140

(b) “exchange”

Table 4.12: User CPU time for upper best-fit configuration.

4.5.4 Upper Best-Fit Plus One

This benchmark tests against object sizes $2^k + 1$ for $k \in \{12, \ldots, 15\}$. These are the sizes of the previous subsection plus one. These sizes are chosen to contrast the differences of best-fit and buddy allocation strategy in terms of memory use. Since buddy allocation rounds up objects to power-of-two sizes, $2^k + 1$ would double the apparent memory usage in theory.

With "no exchange," the most notable increase in memory usage can be seen for JeMalloc. Even so, the amount merely increased by ~28%. Adding one to the size actually decreased the memory used by StreamFlow. It turns out that StreamFlow reserves parts of the object memory to store size information, so allocating 2^k sized objects already incurred the penalty, but that does not explain why adding one to object sizes makes memory usage lower. A plausible explanation is that at these object sizes, some of the memory pages have never been touched by the benchmark, so the operating system never had to allocate physical memory for them. This may also explain why TcMalloc memory usage is the lowest. There are no significant differences in the memory usage between 2^k and $2^k + 1$ sizes for Glibc, Hoard, Libikai, and TcMalloc respectively when compared to themselves.

With "exchange," memory usage is a lot less predictable. The major factor being the ability to quickly move and reuse an object freed by a different thread. This may explain the the blow-up of memory usage with 24 threads followed by a decline with 32 threads. With 24 threads, some threads may be starved because of the hyper-threading, so objects accumulate in these threads' holding area more. With 32 threads, the starvation is again balanced equally among all threads.

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	376M	754M	1.43G	2.80G	5.61G	8.10G	11.6G
±	0	589K	26.1M	85.2M	211M	74.3M	43.2M
hoard	122M	243M	486M	1.13G*	2.51G*	-	-
±	0	297K	511K	3.19M*	0*	-	-
jemalloc	127M	255M	509M	1.00G	1.99G	2.96G	3.97G
±	0	154K	672K	1.17M	2.74M	5.13M	18.2M
libikai	147M	291M	573M	1.11G	2.21G	3.31G	4.42G
±	0	1.78M	812K	902K	2.16M	1.99M	2.44M
libikai$^{-\circ}$	151M	290M	573M	1.11G	2.22G	3.31G	4.42G
±	0	1.78M	1.02M	847K	2.01M	2.80M	3.57M
streamflow	146M	285M	565M	1.11G	2.21G	3.33G	4.42G
±	3.30K	2.25M	9.13M	8.79M	14.0M	13.4M	22.2M
tcmalloc	123M	246M	492M	979M	1.90G	2.85G	3.80G
±	7.83K	119K	341K	632K	1.17M	979K	1.18M

(a) Upper best-fit.

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	376M	754M	1.43G	2.79G	5.65G	8.10G	11.6G
±	0	626K	32.7M	113M	222M	73.5M	46.7M
hoard	122M	243M	516M	1.26G*	2.77G*	-	-
±	0	434K	2.18M	7.28M*	0*	-	-
jemalloc	163M	328M	650M	1.27G	2.52G	3.63G	5.07G
±	0	584K	5.74M	6.95M	24.3M	10.9M	9.64M
libikai	150M	291M	573M	1.11G	2.22G	3.32G	4.42G
±	0	1.28M	0.99M	1.15M	2.23M	2.33M	4.25M
libikai$^{-\circ}$	149M	290M	573M	1.11G	2.22G	3.32G	4.42G
±	0	1.27M	1.24M	1.47M	1.92M	2.22M	2.26M
streamflow	127M	252M	503M	0.98G	1.96G	2.94G	3.92G
±	3.25K	1.32M	1.72M	4.66M	6.42M	4.85M	10.5M
tcmalloc	123M	246M	491M	978M	1.90G	2.85G	3.79G
±	7.83K	86.2K	312K	598K	1.16M	1.00M	574K

(b) Upper best-fit plus one.

Table 4.13: Maximum resident set sizes, “no exchange.”

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	375M	1.05G	2.60G	6.79G	10.3G	15.2G	18.7G
±	0	346M	1.00G	1.53G	2.20G	2.10G	962M
hoard	122M	245M	608M	1.16G*	4.98G*	-	-
±	0	773K	380M	21.5M*	0*	-	-
jemalloc	127M	260M	2.24G	4.88G	12.1G	16.6G	21.6G
±	0	2.16M	354M	1.19G	2.46G	5.02G	2.91G
libikai	147M	291M	788M	1.42G	5.37G	15.6G	5.96G
±	0	1.09M	420M	802M	2.03G	963M	1.44G
libikai$^{-\circ}$	148M	291M	637M	1.32G	7.67G	16.4G	10.3G
±	0	1.65M	156M	513M	982M	767M	878M
streamflow	146M	332M	1.42G	3.64G	10.4G	16.5G	21.3G
±	3.07K	3.06M	960M	1.49G	2.77G	1.77G	3.05G
tcmalloc	123M	251M	990M	1.68G	3.64G	5.89G	3.83G
±	8.04K	19.1M	557M	723M	947M	495M	36.0M

(a) Upper best-fit.

Alloc	# Threads						
	1	2	4	8	16	24	32
glibc	375M	0.99G	2.66G	6.95G	10.3G	14.4G	18.3G
±	0	194M	932M	1.35G	1.90G	2.44G	627M
hoard	121M	247M	549M	1.40G*	5.31G*	-	-
±	0	6.02M	40.1M	419M*	172M*	-	-
jemalloc	163M	334M	1.08G	3.21G	14.4G	23.1G	28.3G
±	0	9.04M	654M	919M	1.21G	2.99G	4.69G
libikai	147M	291M	1.04G	1.41G	4.54G	15.6G	6.04G
±	0	1.70M	523M	432M	1.42G	823M	1.42G
libikai$^{-\circ}$	148M	290M	751M	1.18G	7.85G	16.3G	10.1G
±	0	1.21M	278M	125M	1.21G	762M	1.01G
streamflow	126M	307M	2.17G	4.61G	10.3G	14.0G	19.0G
±	2.87K	1.68M	1.03G	1.37G	2.63G	1.92G	2.62G
tcmalloc	123M	247M	878M	1.66G	3.94G	6.17G	3.92G
±	7.11K	126K	507M	689M	903M	463M	443M

(b) Upper best-fit plus one.

Table 4.14: Maximum resident set sizes, “exchange.”

4.6 Conclusion

A common theme that recurred in the results across all size classes is that, in a multi-threaded program, user CPU time is less of a factor for an allocator to be fast. Utilization dominates wall-clock time. On the one hand, in data center applications where performance per watt is significant, it may still be preferable to have lower user CPU time even when overall progress is slower due to lower utilization. On the other hand, higher user CPU time does not necessarily mean the CPU is using more power, only that some of the fine-grained synchronization stall on the memory bus is now accounted as user CPU time. The experiments here did not measure the power consumption, so this may be a subject for future study.

Also a common theme is that remote free is able to lower memory usage by facilitating object movement across threads, which allows them to be better reused. One concern is that remote-free might cause increased contention in the memory bus when a cache line ping-pongs between different remote threads. Although the thread that owns the heap does not examine remote-freed objects until the heap is exhausted, the increased contention decreases overall memory access throughput. This might explain the significant blow-up of user CPU time of Libikai.

In terms of memory allocation strategy, the benchmark shows that best-fit can be better than buddy allocator in terms of both performance and memory usage. However, the best performance and memory usage is still achieved by packed strategy—which is $O(1)$ with a small amount of internal fragmentation—even for large object sizes. This is why TcMalloc consistently outperforms all allocators across all size classes.

Finally, the packed size class benchmark shows that an allocator implemented using linear pointer can perform comparably or even outperform non-linear imple-

mentations of the same strategy when linear checking is disabled. This is expected because of the erasure property of linear pointers. On the other hand, linear checking causes between 30% to 200% overhead. This is a significant improvement over dynamic binary instrumentation techniques that can experience 2× to 10× slow down. Although it is not clear how much time the authors of the other memory allocators spent debugging memory leaks, no time was spent in debugging memory leaks at all when implementing Libikai. All potential problems were caught during unit testing with linear checking turned on.

Chapter 5

Linearity and Concurrency

Multiprocessor computers have become ubiquitous because the hardware has hit a physical limit in how fast it can go. In order to continue to deliver value to the customers, chip makers nowadays design a processor as a package of multiple sub-processors, each of which can run a thread of computation in parallel, as a multicore processor. A computer can also contain multiple processors that are multicore, and each processor controls its own section of the main memory. Processors can access each other's main memory through a high speed interconnect, so to the program all memory is shared. However, it still requires programs to be adapted to take advantage of this parallel, shared memory architecture. In order to serve these programs, the underlying memory management must ensure that it scales at least as well as the program does. Programs and the memory allocator subsystem must be able to use linear pointers concurrently and continue to enjoy safety and efficiency.

To facilitate parallel programs using linear pointers, linear pointer is extended with atomic semantics: first a simple swap with one volatile and one non-volatile linear pointer, then also a transactional swap with two volatile linear pointers. A volatile pointer is a shared state that can be concurrently observed and mutated by

more than one thread. The simple swap is a hardware intrinsic, and the transactional swap is a complex operation that has to be implemented as a semi-blocking algorithm. Although other atomic hardware intrinsics exist, swapping is chosen as the fundamental atomic operation because it preserves linear ownership. Two algorithms using linear atomic pointers are shown as examples using linear atomic pointers: a double-ended queue suitable for work stealing scheduling, and singly linked list insertion. Although only a casual proof of correctness is given, towards the end of the chapter we will discuss what work is required before a formal system can reason about concurrent algorithms, and propose directions for future work.

5.1 A Historical Perspective

Henry Baker, one of many who are inspired by Linear Logic and recognize its usefulness outside of type-theoretical setting, suggested that linear objects avoid synchronization in a concurrent system because there is no sharing [12]. However, concurrent systems are only interesting as long as different threads of computation are able to communicate and share results with one another. At a higher level, it may seem that it is possible to share results in a linear manner, when an object is merely passed from one thread to another along with the object's ownership. Going deeper to the implementation level where the mechanism of result passing is concerned, it would become apparent that even signaling the availability of the result itself requires synchronization, let alone passing the result. The reason is simple. If we consider two threads to have a producer-consumer relationship, the supplying of the result from a producer and the demanding of the result for the consumer are asynchronous events.

What Baker means in the context of linear objects and synchronization is not that synchronization can be avoided when passing ownership from one thread to another,

but that a thread, as long as it retains ownership of an object, can expect to enjoy exclusive access to it; however, this is only true in a strict linear system that disallows controlled sharing. With controlled sharing such as reference counting or idioms for borrowing as described in Section 2.7, an object's state may be leaked to any thread that happens to have a shared reference to the object. Exclusiveness of access is particularly important for shared-memory parallel computers where communication by shared state is the normal way concurrent computations are carried out. Linearity narrows the sharing down to the mechanism used for object passing, which makes it feasible to reason about the correctness of a concurrent program. One only needs to show that the program obeys linearity, and that the mechanism for passing linear objects is correct. This is preferable to having to prove that arbitrary interleaving of concurrent program state transitions with multiple threads all yield well-defined results.

Assuming exclusive access to an object also enables many compiler optimization that could make the program more efficient by reducing the number of memory access or reordering them. If an object is volatile—meaning it could be mutated outside the current thread of computation—then such optimizations could result in incorrect behavior. Furthermore, if an object is indeed used for sharing states between threads of computation, then compiler optimization techniques that reorder memory access could result in undefined object states being accidentally shared across threads.

Depending on the abstraction presented to the programmer, linear object passing can be done explicitly via a shared queue data structure, or implicitly by having a compiler transform a program with parallel spawn-and-sync semantics to pass objects using a shared queue under the hood. Although the queue itself is shared, an interesting question is how much linearity can be preserved by the implementation of queue insertion and removal operations.

5.2 Hardware Intrinsics

All concurrent algorithms assume that the hardware provides atomic operations at some lower level. An atomic operation is indivisible, in the sense that it always executes completely before its side-effects, such as changing value of a memory location, may be observed from another process. In the case of atomic operation on a machine word that consists of multiple bytes, the modification is either observed on all the bytes or none of the bytes.

It is possible to write a concurrent algorithm with minimal hardware requirement, assuming the hardware provides only atomic reads and writes for machine words; some examples are Dijkstra's original solution to the mutual exclusion problem [44] and Lamport's bakery solution [81]. In practice, computer hardware also provides primitives for atomic unconditional and conditional exchange. An unconditional exchange swaps the value of a register and a memory location. A conditional exchange, also known as "compare and swap," only happens if the memory location contains some expected value. Compare-and-swap was introduced in 1974 as part of IBM System/370 instruction set [89], and has since been made available in almost all hardware.

Having unconditional and conditional exchange drastically simplifies concurrent algorithms. We can use the number of shared states required to achieve mutual exclusion as a measurement of complexity. Mutual exclusion is the problem of ensuring that, among the concurrent processes, at most one process is allowed to be in the critical section, while others may run in the remaining section. Other processes attempting to enter the critical section while a process is already inside all have to wait for that process to exit.

It was shown by James Burns and Nancy Lynch in 1980 that 2^N states, where

N is the number of concurrent processes to be determined in advance, is required and sufficient for an asymmetric mutual exclusion algorithm [22]. The algorithm is asymmetric because it gives lowered numbered processes higher priority in entering the critical section. A symmetric algorithm, on the other hand, provides equal opportunity for all processes. For the number of shared states for symmetric algorithms, Lamport's bakery algorithm in 1974 requires $2^N \times N^N$ states for using a boolean array of N variables as well as an integer array of N variables. Dijkstra's original solution in 1965 requires $N \times 3^N$ states for using one integer and a tri-state array of N variables expressed as two boolean arrays. In 1981, Burns simplified Dijkstra's algorithm to require only $N \times 2^N$ states [23]. It is not known whether it is possible to write a symmetric algorithm with fewer shared states.

In comparison, a spinlock using unconditional exchange requires just one variable with 2 states, for an arbitrary number of processes that need not be known in advance.

The complexity difference also reflects roughly in the code itself. An example implementation of Burns' simplified Dijkstra algorithm is shown below.

Listing 5.1: Burns' simplified Dijkstra algorithm.

```
1  template<size_t N>
2  class burns_dijkstra_mutex {
3   private:
4    volatile size_t turn;
5    volatile bool flag[N];
6
7   public:
8    enum { size = N };
9
10   burns_dijkstra_mutex() throw()
11     : turn(0), flag() {}
12
13   void acquire(size_t i) volatile throw() {
14   l0:
15     flag[i] = true;
16     turn = i;
```

```
17
18    l1:
19      if (turn != i) {
20        flag[i] = false;
21        for (size_t j = 0; j < N; ++j)
22          if (j != i && flag[j]) goto l1;
23        goto l0;
24      } else {
25        flag[i] = true;
26      l2:
27        if (turn != i) goto l1;
28        for (size_t j = 0; j < N; ++j)
29          if (j != i && flag[j]) goto l2;
30      }
31    }
32
33    void release(size_t i) volatile throw() {
34      flag[i] = false;
35    }
36  };
```

The methods `acquire()` and `release()` take an argument that is the process identifier number of the caller. The identifier has to be unique, so there must have been some way to pre-assign numeric identifiers to processes before they run. The identifier is used for each process to signal its intent to enter critical section using a boolean array, so the identifier has to be less than the number of processes. A process only enters the critical section if it maintains its turn and that no other processes express the intent to enter. A process that fails to maintain its turn would withdraw the intent, allowing the process indicated by the turn to proceed into the critical section.

An example spin lock implementation would look like this.

Listing 5.2: Spinlock.

```
1  class spinlock {
2   private:
3    volatile bool state;
4
5   public:
```

```
6   spinlock() throw()
7     : state(true) {}
8
9   void acquire() volatile throw() {
10    while (true) {
11      if (atomic_exchange_acq(&state, false))
12        break;
13    }
14  }
15
16  void release() volatile throw() {
17    atomic_exchange_rel(&state, true);
18  }
19 };
```

The methods `acquire()` and `release()` no longer require process identifier number as an argument. This spin lock could support an unbounded number of processes with just two states: "true" indicates availability of lock, and "false" indicates the lock is unavailable. The spinlock repeatedly tries to acquire the lock by writing false to the state, but if true is returned from the old state, that means it has swapped out the "available" bit of the lock, so it should return. The release function simply writes the true value back. Here, an assignment would work just as well, but observe there is a symmetry in the acquire and release; that is, if true represents the linear resource of the lock, then it can be acquired and released both using the same atomic exchange operation.

The code in this chapter uses atomic macros defined in `<atomic.h>` from GNU C Library (GLIBC) [1], which provides machine-specific definitions of the conditional and unconditional exchange operations with a machine-independent interface.

- The `atomic_exchange_*(mem, newval)` macros unconditionally write `newval` to `mem`, which is the memory address of the shared variable, and the macro evaluates to the old value of the shared variable.

- The `atomic_compare_and_exchange_bool_*(mem, newval, oldval)` macros conditionally write `newval` to the shared variable at memory address `mem` if the current value of the variable is `oldval`. The macro evaluates to 0 if the condition holds and the exchange was made, and to 1 otherwise.

Each macro has two variants, `*_acq` and `*_rel`, and they differ in where the memory barrier is placed: `_acq` places memory barrier after the exchange, and `_rel` places memory barrier before the exchange. Memory barrier is sometimes necessary for hardware that normally reorders memory reads and writes to maximize processor pipeline. Memory barrier prevents the reordering from taking place across the barrier, and in the case of spin locks, ensures that memory access within the critical section does not interleave with memory access in the remaining section. The distinction of where the memory barrier is placed is less important for a non-blocking data structure where the data structure can be safely manipulated simultaneously by all concurrent processes, without requiring the entering of critical section.

If a machine does not have a primitive for unconditional exchange, which is rare, it can be emulated using compare-and-swap as follows.

Listing 5.3: Emulating unconditional exchange using compare-and-swap.

```
1 template<typename Tp>
2 Tp atomic_exchange(Tp *mem, Tp newval) {
3   Tp oldval;
4   do {
5     oldval = *mem;
6   } while(atomic_compare_and_exchange_bool(mem, newval, oldval) != 0);
7   return oldval;
8 }
```

Although this emulation is inefficient compared to the hardware primitive, this illustrates how compare-and-swap is used when there is already a copy of some old, expected value of the memory location. This has ramification if we were to impose

linearity on values to be conditionally exchanged.

5.3 Linear Atomic Value

A linear atomic value can be seen as a generalization of the spinlock state. The state now carries a value whose linear ownership can be exchanged from one process to another, and it may be operated upon using both unconditional and conditional exchanges. The spinlock can be seen as a special case of linear atomic value that is a boolean, where boolean "true" indicates availability of some resource, and boolean "false" is the nil value that can be arbitrarily introduced and eliminated.

While unconditionally exchanging linear values is straightforward, there is an immediate challenge formulating conditional exchange for linear values. Recall that conditional exchange requires that we already have a copy of the expected old value at the memory location. This copy obviously is an alias of the value in memory which can only be obtained by controlled sharing. To prevent abuse, this value could have a type that is only useful for the purpose of compare-and-swap. For example, if the linear value is a linear pointer to some object, then the expected value would simply be a `void *` so that it may not be dereferenced by accident.

The `linear_atomic_base` class is an add-on to a `linear_base` that supports atomic conditional and unconditional exchange operations. Some of its methods are decorated `volatile` only to signify the fact that these methods assume that the linear atomic value could be changed by a different thread of computation, so the compiler does not optimize away memory read. It is important to note that the `volatile` decoration does not make the method or the memory access therein atomic.

Listing 5.4: Linear atomic base.

```
1 template<
2   class Self /* implements has_value<?v> */,
```

```
3     class Traits, typename Expect>
4   class linear_atomic_base {
5    private:
6     SELF_TYPE_DECL(Self);
7     typedef typename Traits::value_type value_type;
```

This class provides value access functions `expect()`, `reset()` and `release()` but does not implement the actual storage of value. The value storage is assumed to be provided by the `value()` method from subclass `Self` (used as the curiously recurring template pattern [32]) which returns the lvalue reference to the value storage. The volatile `expect()` method returns a non-linear alias to the linear atomic value. A volatile `release()` allows elimination of a linear value atomically.

```
8    public:
9     typedef Expect expect_type;
10
11    expect_type expect() const volatile throw() {
12      return static_cast<expect_type>(self()->value());
13    }
14
15    value_type release() volatile throw() {
16      return atomic_exchange_rel(&self()->value(), Traits::nil);
17    }
```

The conditional exchange operation `cswap()` exchanges the linear atomic value with another linear atomic value if the current value is the same as the expected value. It returns `true` if the exchange happened, and `false` if not. The unconditional exchange operation `swap()` simply exchanges two linear atomic values. Both operations are made atomic using the GLIBC macros.

```
18    bool cswap(self_type& that, expect_type expect) volatile throw() {
19      value_type old_val = static_cast<value_type>(expect);
20      if (atomic_compare_and_exchange_bool_rel(
21              &self()->value(), that.value(), old_val) == 0) {
22        that.value() = old_val;
23        return true;
24      }
25      return false;
```

```
  }

  void swap(self_type& that) volatile throw() {
    value_type old_val = atomic_exchange_rel(&self()->value(), that.value());
    that.value() = old_val;
  }
```

And this concludes the `linear_atomic_base` class.

```
};
```

As an example, `linear_atomic_ptr` can be built from `linear_ptr` and `linear_atomic_base` as follows. It uses `void *` for the type of expected value so that an expected value alias could not be dereferenced to access the object.

Listing 5.5: Linear atomic pointer.

```
template<typename Tp, class Traits = value_traits<Tp *> >
class linear_atomic_ptr /* implements atomic_pointer<Tp> */
  : public linear_ptr<Tp, Traits>,
    public linear_atomic_base<linear_atomic_ptr<Tp, Traits>, Traits, void *> {
 public:
  // Value constructor, reference constructor, move constructor, value
  // assignment, reference assignment.
};
```

For brevity, the required constructors and assignment operator overloading are omitted. This class must also inherit the `reset()` and `release()` methods from both base classes in order to avoid ambiguity.

A `linear_atomic_chroma_ptr` can be built similarly, but using `uintptr_t` for the expected value.

Listing 5.6: Linear atomic chromatic pointer.

```
template<typename Tp, typename Cp, unsigned int bits = 2u>
class linear_atomic_chroma_ptr /* implements atomic_pointer<Tp> */
  : public linear_chroma_ptr<Tp, Cp, bits>,
    public linear_atomic_base<linear_atomic_chroma_ptr<Tp, Cp, bits>,
                              value_traits<uintptr_t>, uintptr_t> {
 public:
  // Value constructor, reference constructor, move constructor, value
```

```
8   // assignment, reference assignment.
9 };
```

Linear atomic value can be immediately used as a spinlock. It is easy to see that the spinlock `acquire()` function in Listing 5.2 can be written in terms of `swap()`. It would attempt to swap the lock with a `nil` value and check if a non-`nil` value is swapped out, which acquires the lock. The release() function would `swap()` the linear resource back into the lock which we know would be `nil`. This mechanism can already be used to exchange values between concurrent threads as a single-slot queue.

5.4 Linear Slots

A linear slots abstraction is a generalization that allows exchange of several linear atomic values, much like how semaphore generalizes mutual exclusion. Semaphore, also conceptualized by Dijkstra, is a counter that admits at most a certain number of processes as determined by the number of available resources. The counter starts at a finite positive number. A process acquires one resource by atomically decreasing the counter, and releases the resource by atomically increasing the counter. When the counter reaches zero, no more processes may be admitted.

Linear slots is an analogy to the semaphore using an array of linear atomic values. It gives a view of the slots as a circular array. Two operations are provided on possibly infinite index i: a put(x, i) would only store x to slot i if slot i is vacant, and get(i) would attempt to swap out the value at slot i with nil. However, the circular array only has a finite capacity, so the slots may only hold a finite number of resources even though the index is potentially infinite.

Listing 5.7: Linear slots.

```
1 template<class Value /* implements atomic_value<?> */, size_t n>
2 class linear_slots {
```

```
 private:
  STATIC_ASSERT(IS_POW2(n));

 public:
  typedef Value value_type;
  enum { capacity = n };

  bool put(value_type& x, size_t i) volatile throw() {
    if (!x) return true;
    value_type nil;
    return values_[i % n].cswap(x, nil.expect());
  }

  value_type get(size_t i) volatile throw() {
    value_type x;
    values_[i % n].swap(x);
    return x;
  }

 private:
  value_type values_[n];
};
```

The static assertion checks at compile time that array size n is a power of 2, which makes it possible for the array index to seamlessly wrap-around at the same time integer overflow happens.

A process that wishes to acquire a resource from the linear slots would iterate through all i and try a get(i). A process that wishes to release resource x into the slots would iterate through all i and try a put(x, i) until it succeeds.

Listing 5.8: Slots semaphore.

```
template<class Slots>
class slots_semaphore {
 public:
  typedef Slots slots_type;
  typedef typename slots_type::value_type value_type;

  value_type acquire() volatile throw() {
    value_type x;
```

```
9       for (size_t i = 0; ; ++i) {
10        x = slots_.get(i);
11        if (x) break;
12      }
13      return x;
14    }
15
16    void release(value_type x) volatile throw() {
17      for (size_t i = 0; ; ++i) {
18        if (slots_.put(x, i))
19          break;
20      }
21    }
22
23   private:
24    slots_type slots_;
25  };
```

Slots differ from a semaphore in the sense that slots typically start empty and becomes populated by a producer. The fixed capacity of the slots imposes a limit on the number of resources that can simultaneously be passed from all producers to all consumers, but it does not restrict the total number of resources that could pass through the slots. This is already a primitive form of a multiple-producer multiple-consumer queue except there is no ordering guarantee. A more sophisticated implementation would randomize the index to begin the put() and get() trial, so that memory access contention is distributed evenly across the slots.

5.5 Double-Ended Queue

A queue is a collection of objects where items removed from the queue are ordered the same way they were inserted, in first-in first-out (FIFO) order. A double ended queue provides one additional operation, pop(), which removes the item just inserted, in last-in first-out (LIFO) order. This data structure is useful for work stealing where

the deque is used mostly as a call-stack (LIFO) by a worker, but work may be removed from the bottom of the stack (FIFO) by a thief which minimizes interference with the worker. Also, by assuming that the deque is single-producer (the worker) and multiple-consumer (thieves), the asymmetry allows the push() and pop() to be vastly simplified compared to remove() which must ensure that thieves do not step on each other's toes.

Cilk is a parallel computing language whose runtime system uses this asymmetric deque [52]. An asymmetric deque is the foundation of Cilk's work-first principle which minimizes overhead along the critical path of the worker's stack-like computation, but allows more synchronization overhead to occur outside of the critical path when work is stolen. Cilk implements a semi-locked deque where push() requires no mutual exclusion locking, pop() only acquires the lock when the deque is about to become empty—which means it may enter a contention with the thief—and remove() aways requires locking. Mutual exclusion ensures that only one thread could manipulate the bottom of the deque while other threads must block.

Here a non-blocking asymmetric deque is presented which requires no mutual exclusion in any of its operations, based on the design of an array-based non-blocking queue by Robert Colvin and Lindsay Groves [31], which is itself a formally verified and corrected version of an earlier design by Shann, Huang and Chen [103]. The earlier queue designs were symmetric as they assume multiple producer and multiple consumer, and they do not support stack-like operation.

Furthermore, in order to work-around the ABA problem commonly encountered by concurrent algorithms using compare-and-swap, the earlier designs require a special compare-and-swap operation on a word with a version counter. Recall that compare-and-swap conditionally exchanges the value of a memory location if the memory location holds some expected value. ABA problem happens when the memory location is

changed from A to B and back to A, so the exchange would succeed even though the memory location had been modified twice. An algorithm generally could not rely on compare-and-swap for modification detection unless it uses a monotonically increasing version counter along with compare-and-swap to detect changes. Unfortunately, not many hardware supports this special compare-and-swap. The non-blocking asymmetric deque presented here uses only standard atomic conditional and unconditional exchange, which makes a practical implementation possible.

Without version counter, compare-and-swap needs to be tied-in more strongly to the invariants of the data structure so that invariant violation implies that the exchange would fail. This is a key insight pivotal to the correctness proof of any non-blocking lock-free wait-free algorithms using compare-and-swap.

5.5.1 Structural Invariants

The deque consists of a cyclic array of linear objects *values* of size N, and two unsigned integers *front* and *rear*, which are indices into the array. In terms of stack-like operations of the deque, *front* is the top and *rear* is the bottom. The actual position in *values* is computed by taking modulo N; however, integer wrap-around is allowed to happen in practice, provided that:

- Array size N is in power of 2, so integer wrap-around does not break the continuity of the array indexing.

- Given two unsigned integers a and b, the inequality $a < b$ is written as $(a-b) < 0$ where the quantity $(a-b)$ is interpreted as a signed integer in two's complement. This "rolling inequality" works as long as b is never ahead of a by more than half the maximum value expressible by the integer. This condition holds when comparing *front* and *rear* because they are never further apart than the array

size, which is much smaller than half of the maximum integer value.

- A thread is never "stuck" long enough to allow integer wrap-around to cause ABA problem. This assumption about the relative computing speeds of the threads would be frowned upon by Dijkstra, but sufficiently large integer size makes speed differences unlikely to cause problems.

In addition, the following properties about *front*, *rear*, and *values* hold.

- It is always the case that $0 \leq front - rear \leq N$. When $front - rear = N$, the deque is full. When $rear = front$, the deque is empty. However, the deque may or may not be empty when $rear < front$.

- When $rear < front$, we allow $values[rear]$ to be nil when it is removed by a consumer while it has not had a chance to update *rear*, or $values[front - 1]$ to be nil when it is removed by the producer while it has not had a chance to update *front*. Otherwise for all i such that $rear < i < front - 1$, we require that $values[i]$ be a continuous streak of non-nil items.

- For all i such that $i < rear$ or $i \geq front$, it must be that $values[i]$ is nil, unless $front - rear = N$. Only the producer is allowed to put an item in $values[front]$, and it is only able to do so safely when this invariant holds.

- Only the producer is allowed to modify *front* which may increase or decrease, but *rear* may be modified by any thread and is monotonically increasing.

Notice that ABA problem could still happen with individual items in *values*. The problem is partially remedied with linear resources that have unique values, such as memory addresses of objects. Even so, an item could still be removed and somehow

put back into the same place. When ABA problem happens, items could be prematurely removed, hence are no longer removed first-in first-out. Fortunately, a strict ordering is not required when the deque is used for work stealing.

5.5.2 The Implementation

The linear deque class is parameterized by the atomic value type and a fixed size.

Listing 5.9: Linear double-ended work-stealing queue.

```
1  template<class Value /* implements atomic_value<?> */, size_t n>
2  class linear_deque {
3   private:
4    STATIC_ASSERT(IS_POW2(n));
5
6   public:
7    typedef Value value_type;
8
9    linear_deque() throw()
10     : rear_(0u), front_(0u) {}
11
12   ~linear_deque() throw() {
13     assert(rear_ == front_);
14   }
```

The push() function is only used by the producer to put a value x to the front of the deque. It returns `true` if the item is successfully pushed, and x would become nil; or `false` if the deque is full, and x would be left intact.

```
15   bool push(value_type& x) throw() {
16     size_t front = front_;
17     if (!x)
18       return true;
19     else if ((ptrdiff_t) (front - rear_) >= (ptrdiff_t) n)
20       return false;
21
22     values_[front % n].swap(x);
23     front_ = front + 1;
24
25     return true;
```

```
26    }
```

If push() determines that the deque has available space, since there is only one producer, and *rear* is monotonically increasing, the deque could only free up more space after the fullness check. If the deque has space, then push() swaps x with $values[front]$, and the value swapped out from $values[front]$ must be nil because $values[i]$ is nil for $i \geq front$. The producer is the only thread modifying *front*, so it can do so using simple memory write (which is atomic). It does not need to use compare-and-swap.

The pop() function is only used by the producer to remove a value from the front of the deque. A non-`nil` item is removed upon success, or nil might indicate that the deque is empty.

```
27    value_type pop() throw() {
28      size_t front = front_, pop_ind = (front - 1) % n;
29      value_type x;
30      values_[pop_ind].swap(x);
31
32      if (x)
33        if ((ptrdiff_t) (front - rear_) > 0)
34          front_ = front - 1;
35
36      return x;
37    }
```

If pop() swaps out a non-nil item from the "pop index" which is $front - 1$, then it would decrement *front* provided that the resulting *front* still satisfies $front \geq rear$. If it encounters a nil value, then it assumes that the deque has become empty because of the continuity invariant of non-nil items.

The remove() function is used by any number of consumers or the producer to remove an item from the rear of the deque. Because of this, the algorithm must allow any of *front*, *rear*, and $values[rear]$ be changed by another thread at any time. The general strategy is to create a snapshot of these variables to work with. It

would attempt to swap the value out of *values*[*rear*] which may fail if another thread succeeded doing so before the current thread, in which case it must still make progress by incrementing *rear* to skip over the position at *values*[*rear*] which is now nil. This is how remove() achieves non-blocking. Note that remove() will work correctly even if the invariant requiring non-nil items to be continuous is broken. If the deque is somehow corrupted this way, for example due to ABA problem as mentioned before, remove() would still be able to exhaust the items in the deque while pop() would consider the deque to be empty.

```
38   value_type remove() volatile throw() {
39     typedef typename value_type::expect_type expect_type;
40     size_t rear, front, rear_ind;
41     value_type x;
42     expect_type expect;
43
44   retry:
45     rear = rear_, front = front_;
46     rear_ind = rear % n;
47     expect = values_[rear_ind].expect();
48
49     if ((ptrdiff_t) (front - rear) <= 0)
50       return value_type();
51
52     if (rear != rear_)
53       goto retry;
54
55     if (expect != x.expect() /* nil */) {
56       if (values_[rear_ind].cswap(x, expect)) {
57         atomic_compare_and_exchange_val_rel(&rear_, rear + 1, rear);
58         return x;
59       }
60     } else {
61       if ((ptrdiff_t) (front - rear) > 1)
62         if (values_[rear_ind].expect() == x.expect() /* nil */)
63           if (front == front_)
64             atomic_compare_and_exchange_val_rel(&rear_, rear + 1, rear);
65     }
66     goto retry;
```

```
67   }
```

The algorithm first takes a moving snapshot of the deque (lines 45–47) in which *rear*, *front*, and *values*[*rear*] are sampled at different times, and the variables may not actually simultaneously hold these values in the snapshot. The accuracy of the snapshot must be ascertained. For example, if the algorithm detects that *rear* has been modified (line 52), this means the expected value of the item to be removed will have changed as well, so it only makes sense to retry.

If the item to be removed is not nil (line 55), then the algorithm attempts to remove it using compare-and-swap (line 56). If the item is successfully exchanged out, it would try to advance rear past the removed item (line 57), but this does not have to succeed because another consumer calling remove() could have done this for us. In any case, the removed item is returned.

If the item to be removed appears to be nil, the algorithm would only advance *rear* if, after doing so, there would be at least one item between *rear* and *front* (line 61). This ensures that *rear* would never advance past *front* even if pop() tries to decrement *front*. Then it needs to check whether the snapshot contains accurate values for *values*[*rear*] (line 62) and *front* (lines 63). The sandwich check ensures that both *values*[*rear*] and *front* simultaneously hold the values in the snapshot at some point of time. If so, the algorithm attempts to advance *rear* past the nil value. Even though both *rear* and *front* could be changed in the mean while, it is impossible for pop() to rewind *front* to the same place as *rear* because *values*[*rear*] is nil, which guarantees that after setting *rear* to $rear + 1$ using compare-and-swap, the invariant $rear \leq front$ is still satisfied.

The rest of the class declares the member variables.

```
68  private:
69   volatile value_type values_[n];
```

```
70    volatile size_t rear_, front_;
71  };
```

And this completes the linear deque.

5.5.3 Discussion

The asymmetric deque algorithm presented here resembles linear slots with the additional *front* and *rear* variables. However, while slots manipulation is still blocking like spin-locks, the push() and pop() operations of the deque are $O(1)$ and non-blocking; remove() is non-blocking, and in the worst case would make progress in $O(P)$ attempts where P is the number of threads trying to remove from the deque.

Furthermore, the push() and pop() operations for the producer are as fast as a single-threaded stack. This allows the producer to use the deque as a call stack with no synchronization overhead. All the synchronization complexity is delegated to remove() which is used by a thief that is only stealing work because it is already idle, so the runtime system wastes no time in the critical path of the computation.

Although the non-blocking deque has negligible synchronization overhead, it is certainly not contention-free. In a typical parallel system with a number of workers, there would be the same number of deques statically assigned one-to-one to the workers. Each deque would occupy its own cache line, and each worker would use its private deque as a stack. Normally, push() and pop() will not cause activity on the memory bus at all. If any worker becomes idle, it would steal work from its peer in a round-robin fashion. Note that round-robin is equivalent to randomizing which deque to steal work from because each worker all round-robin at different rates and would become out of phase. This is sufficient to distribute memory contention over all deques.

The deque would also be relatively small-sized. If the producer finds that the

deque is full, it could simply compute the work right away without placing it onto the deque. The deque is only used to make work available to potential thieves. In a typical divide-and-conquer workload, the work towards the rear (bottom) of the deque tends to be larger sub-problems than the work towards the front (top), so larger sub-problem tends to be stolen while smaller sub-problem tends to be computed by the same worker. This allows the work-stealing to preserve the locality of reference and uses the memory hierarchy the most efficiently. This is a property that linear slots would not have because slots put() and get() guarantee no ordering.

5.5.4 Related Work

It is worth noting that while non-blocking work-stealing deque has been widely studied in the past, most of them are not practical, either due to the assumption of the availability of double- or multi-word compare-and-swap, or due to assumption about memory management to overcome ABA problem. Multi-word compare-and-swap is possible to achieve in software using single-word compare-and-swap [63], but a single operation takes several hundreds of instruction cycles[1] and also requires special memory management to guarantee fresh "descriptors" for each operation.

The deque by Arora, Blumofe and Plaxton [8] assumes the availability of compare-and-swap operation with versioning. The deque uses an infinite array with no size constraint, which is impractical, but it is easy to adapt the deque to a bounded cyclic array using a modulo index as well as comparing emptiness and fullness using a rolling inequality, as described in our structural invariant. However, most importantly, their algorithm has a race condition.

Here the algorithm is scrutinized more closely. Their notion of top and bottom

[1]The ballpark figure is 2-4 microseconds per operation in an era where CPUs are rated at 500MHz to 1.5GHz.

is opposite to ours. Their "bottom" corresponds to our "front." The pushBottom() function is used by the single producer to add more work.

```
1   void pushBottom(Thread *thr)
2   {
3     localBot = bot;
4     deq[localBot] = thr;
5     localBot++;
6     bot = localBot;
7   }
```

Aside from the fact that it does not respect a bounded queue size which is easy to fix, this function is straightforward.

The "top" corresponds to our "rear." The popTop() function is used by thieves to steal work.

```
8   Thread *popTop()
9   {
10    oldAge = age;
11    localBot = bot;
12    if (localBot <= oldAge.top)
13      return NULL;
14    thr = deq[oldAge.top];
15    newAge = oldAge;
16    newAge.top++;
17    cas(age, oldAge, newAge);
18    if (oldAge == newAge)
19      return thr;
20    return ABORT;
21  }
```

Their compare-and-swap modifies `newAge` to the value of `oldAge` if the operation succeeds, so the condition at line 18 is true if the operation succeeds. Although the function may ABORT, it suffices to retry.

The popBottom() function is used by the worker to remove work.

```
22  Thread *popBottom()
23  {
24    localBot = bot;
```

```
25      if (localBot == 0)
26        return NULL;
27      localBot--;
28      bot = localBot;
29      thr = deq[localBot];
30      oldAge = age;
31      if (localBot > oldAge.top)
32        return thr;
33      bot = 0;
34      newAge.top = 0;
35      newAge.tag = oldAge.tag + 1;
36      if (localBot == oldAge.top) {
37        cas(age, oldAge, newAge);
38        if (oldAge == newAge)
39          return thr;
40      }
41      age = newAge;
42      return NULL;
43    }
```

Most of the complexity is actually attributed to resetting top and bottom to zero when the worker is about to enter contention with a thief (line 31). This complexity does not make popBottom() slower than popTop() because it only happens in the case of contention which is rare. When this happens, the resolution of who gets the item is determined by who gets to modify the "top" index (line 37). This has a nice side-effect that when the deque becomes empty, it would fully restore its capacity by setting both top and bottom to 0; otherwise as items are added and stolen, the deque's capacity would diminish.

There is a race condition between the worker and a thief. Suppose at some point in time, `bot == oldAge.top + 1` which is necessary for the condition at line 31 to hold. The worker enters popBottom() and is suspended right before line 28. A thief enters popTop(), retrieves the same item, and is suspended right before line 17. The worker proceeds and happily returns the `thr` item that the last thief has not finished stealing. The thief proceeds and returns the same item. This causes a task to be

computed twice, which may have non-idempotent side-effects.

Hendler, Lev, Moir, and Shavit designed a non-blocking deque that can dynamically adjust its size [65]. The motivation is to overcome the problem that Arora-Blumofe-Plaxton deque loses capacity if the deque never becomes empty. The dynamic sizing is accomplished by using a doubly linked list of array fragments. It uses multi-word compare-and-swap to modify "top" and "bottom." The dynamic deque is mostly analogous to the original but is more complex, so it is not clear if the race condition that affected the original also affects the new version.

The linear deque presented in this section is the first of its kind with practical and efficient software implementation, and is simple enough to prove for correctness.

5.6 Singly Linked List

Non-blocking manipulation of singly linked list presents an interesting challenge where aliasing is required for expressing the algorithm. Consider list insertion without using linear pointer.

```
template<
  class Ptr /* implements pointer<? implements singly_linked_node<Ptr> > */>
class singly_linked_list_atomic_nonlinear {
 public:
  typedef Ptr ptr_type;

  static void insert_at(ptr_type node, volatile ptr_type& curr) throw() {
    while (true) {
      ptr_type expect = curr;
      node->next() = expect;
      if (atomic_compare_and_exchange_bool_rel(&curr, node, expect) == 0)
        break;
    }
  }
};
```

A temporary alias of `curr` is stored into the `node->next()` pointer of the node. If the compare-and-swap to modify `curr` fails, the value of `node->next()` is discarded and subsequently overwritten in the next retry.

The introduction and elimination of the alias is more clear with the linear atomic pointer version of the same function.

Listing 5.10: Concurrent singly linked list insertion.

```
template<
  class Ptr
  /* implements atomic_pointer<? implements singly_linked_node<Ptr> > */>
class singly_linked_list_atomic {
 public:
  typedef Ptr ptr_type;
  typedef typename ptr_type::expect_type expect_type;

  // Non-blocking singly linked list insertion.
  //
  static void insert_at(ptr_type node, volatile ptr_type& curr) throw() {
    assert(node);

    while (true) {
      expect_type expect = curr.expect();
      node->next().reset(expect);  // aliasing.
      if (curr.cswap(node, expect))
        break;
      (void) node->next().get();  // undo aliasing.
    }

    (void) node.get();  // undo aliasing.
  }
};
```

This code requires a new `reset()` method that takes a non-linear alias so that the expected value may be temporarily materialized as a linear value. If the compare-and-swap fails, the alias still resides in `node->next()` so that needs to be eliminated. If the compare-and-swap succeeds, the alias is the old `curr` now swapped out to `node`, so we eliminate the alias in `node`. Linearity checking forces us to eliminate the extra

alias before the next iteration of the loop or before the function returns.

For this purpose of allowing the expect value to be materialized to a linear value, we augment `linear_atomic_base<>` with the following method.

```
template<...>
class linear_atomic_base {
  ...

 public:
  void reset(expect_type expect) throw() {
    self()->value() = static_cast<value_type>(expect);
  }

  ...
};
```

This controlled aliasing also allows us to implement the corresponding `remove_at()` function naively.

```
1   static ptr_type remove_at(volatile ptr_type& curr) throw() {
2     ptr_type node;
3
4     while (true) {
5       expect_type expect = curr.expect();
6       node.reset(expect);  // aliasing.
7       if (!node) break;
8
9       if (curr.cswap(node->next(), expect))
10        break;
11      (void) node.get();  // undo aliasing.
12    }
13
14    if (node)
15      (void) node->next().get();  // undo aliasing.
16    return node;
```

The usual precautions about volatile value is taken between lines 5 to 7, which takes a snapshot of `curr` and ensures that it is not nil. The code also observes linear ownership semantics. However, this implementation is still incorrect in a subtle way. The reason is that `node->next()` is actually volatile, a fact that is not reflected on

the type. If its type were volatile, then it cannot be used with `cswap()` which requires the first argument to be non-volatile.

It is not hard to see why the first argument must be non-volatile from the definition of `cswap()` in Listing 5.4. If `that` were volatile, then between the read of `that.value()` and the assignment `that.value() = old_val`, the value could have been modified externally, and the external modification would have been lost. However, `atomic_compare_and_exchange_bool_rel()` would only protect us against the external modification of `self()->value()`, not `that.value()`.

One way to safely remove singly linked list nodes in a non-blocking manner is given by Tim Harris [62]. Assuming that memory addresses of nodes are word aligned, the idea is to use the least significant bit of the node pointers as a marker to indicate whether the *current* node that holds the pointer is to be removed, as opposed to marking the node the pointer points to. The marker is updated using compare-and-swap, requiring the next pointer to be the same. A marked node could then be *excised* by any thread that traverses the list, using another compare-and-swap, thus allowing the removal to be non-blocking.

5.7 Linear Transactional Pointer

It is also possible to use a double-word compare-and-swap to accomplish non-blocking singly-linked list node removal, as shown by Greenwald [58]. A double-word compare-and-swap atomically performs two sets of compare-and-swap, with two addresses, two old values, and two new values. However, in this particular case, we only require compare-and-swap to exchange two memory locations. A single-compare double-swap `atomic_compare_and_exchange_ptr_bool(mem1, mem2, oldval1)` could be used to correctly implement `cswap()` that allows the first argument to be volatile.

```
1  bool cswap(volatile self_type& that, expect_type expect) volatile throw() {
2    value_type old_val = static_cast<value_type>(expect);
3    return (atomic_compare_and_exchange_ptr_bool(
4                &self()->value(), &that.value(), old_val) == 0);
5  }
```

A linear transactional pointer is one that supports `cswap()` and `swap()` on two volatile memory locations transactionally.

Note that our single-compare double-swap is not to be confused with "compare double and swap" (cds) instruction on IBM 370 [89] which is a compare-and-swap that operates on two consecutive words (8 bytes on a 32-bit machine). The same instruction is known as CMPXCHG8B on x86 (32-bit), or CMPXCHG16B on amd64 (64-bit). Double-width compare-and-swap is useful to manipulate pointers with a timestamp in order to avoid the ABA problem.

For single-compare double-swap, it is unlikely that the hardware can provide the operation as a primitive because the two memory locations need not be consecutive. On Non-Uniform Memory Architecture machines, the two addresses may belong to two independent memory controllers, which complicates memory bus locking. A software approach similar to the restricted double-compare single-swap operation (RDCSS) [63] proposed by Tim Harris et al. would have to be used. In their work, RDCSS is used as a primitive to implement generic n-word compare-and-swap.

RDCSS takes two memory addresses, two old values, and one new value. If the memory addresses simultaneously hold the two expected old values, then a single new value is written to the first memory address, leaving the second memory address unmodified. Before the RDCSS operation takes place, the first address—which we will potentially modify—is replaced with a pointer to a descriptor about the operation, using a single-word compare-and-swap. The descriptor contains the operands of RDCSS. The descriptor could be told apart from a regular value as a chromatic pointer

which stores the color as the least significant bits of an otherwise aligned pointer. When another thread reads a memory location and finds a descriptor, it would attempt to make progress by finishing off the operation described there, so RDCSS is non-blocking. Completing the operation takes another single-word compare-and-swap. Completion is only possible if the destination still has the pointer to the descriptor.

The algorithm is correct as long as a new RDCSS operation implies that a different descriptor pointer will be used. Since we cannot have an infinite supply of unique descriptors, special care must be given to the memory allocation and reuse of the descriptors. A descriptor must not be reused if any thread is still attempting to complete the operation using that descriptor. Otherwise a situation may arise that, while the thread performing RDCSS binds the destination memory address with the descriptor and blocks, another thread comes along and attempts to complete the operation, but before the operation could be completed, the first thread resumes, finishes the operation, and releases the descriptor for reuse. If this descriptor is immediately reused for the same memory location, the second thread would be completing the operation using an inconsistent snapshot of the descriptor. Furthermore, since the descriptor is reused, compare-and-swap would succeed with the corrupted value.

The memory allocation technique for descriptors, as suggested by Tim Harris, uses reference counting to prevent reuse. Each thread maintains a private free list of descriptors specifically for the operations initiated by that thread. Reference counting and thread-private provisioning of descriptors ensures the required "freshness" of descriptor in order for compare-and-swap to detect modifications. It is reported that 10% of the execution time is spent on reference counting and maintaining the free list when there is no contention.

An alternative is to loosen the non-blocking requirement so that only readers are non-blocking, but writers would block unless it has exclusive write access. The readers are non-blocking because they can read the old values from the descriptor before the operation completes. Since there is at most one writer, the descriptor may be reused without reference counting, and each thread could just use its own descriptor. This also alleviates the need of memory allocating the descriptor. This is the approach the linear transactional pointer here would use to implement `cswap()` between two volatile memory locations. Both memory locations would have to be "locked" with the descriptor before the swap would happen.

Listing 5.11: Linear Transactional Pointer.

```
1  template<typename Tp>
2  class linear_trans_ptr /* implements trans_pointer<Tp> */
3    : public linear_base<linear_trans_ptr<Tp>, value_traits<uintptr_t> > {
4   protected:
5    typedef value_traits<uintptr_t> traits_type;
```

The linear transactional pointer is similar to linear chromatic pointer in that it uses coloring to designate the descriptor. However, here the color is internal and not exposed to the user. Three color tags are defined: *zero* for a regular pointer, *one* for a descriptor at the first memory address, and *two* for a descriptor at the second memory address. The *one* and *two* tags help the reader extract the correct field from the descriptor upon access.

```
6    enum color_t { zero = 0, one = 1, two = 2, three = 3 };
7    enum { bits = 2u };
8
9    static const uintptr_t mask = (1 << bits) - 1;
10
11   static color_t color_of_int(uintptr_t x) throw() {
12     return static_cast<color_t>(x & mask);
13   }
14
15   template<typename Sp>
16   static Sp *ptr_of_int(uintptr_t x) throw() {
```

```
17      return reinterpret_cast<Sp *>(x & ~mask);
18    }
19
20    static uintptr_t
21    int_of_ptr(const volatile void *p, color_t c = color_t()) throw() {
22      uintptr_t x = reinterpret_cast<uintptr_t>(p);
23      assert((x & mask) == 0);
24      assert((c & ~mask) == 0);
25      return x | c;
26    }
```

The transaction descriptor contains the two memory addresses to be exchanged, the original values in these locations, and a checksum to ensure consistency of a snapshot. The checksum is computed by XOR'ing the other fields. This will work even if the fields of the descriptor are read out of order. If a snapshot of the descriptor is not consistent, the reader may return a bogus value when the descriptor is being reused for a different operation.

```
27    struct desc_t {
28      volatile uintptr_t *a1, *a2;  // addresses to pointers to be exchanged.
29      Tp *o1, *o2;                  // original values in these pointers.
30      uintptr_t cs;        // checksum to ensure consistency.
31    };
32    //
33    static uintptr_t desc_checksum(const volatile desc_t& d) throw() {
34      const volatile uintptr_t *p =
35        reinterpret_cast<const volatile uintptr_t *>(&d);
36      return p[0] ^ p[1] ^ p[2] ^ p[3] ^ p[4];
37    }
```

Note that replacing the checksum with a monotonically increasing timestamp does not guarantee snapshot consistency, since the fields might have been in the process of updating before the timestamp reflects the change.

A reader would use the following `access()` function to read the value from a memory location, which might be a regular value or a descriptor. In the case of a descriptor, the reader discerns which field from the descriptor to return according to

the color.

```
38  Tp *access() const volatile throw() {
39    while (true) {
40      uintptr_t x = this->value();
41      assert(x != traits_type::invalid);
42
43      color_t c = color_of_int(x);
44      void *p = ptr_of_int<void>(x);
45
46      if (c == zero)
47        return static_cast<Tp *>(p);
48
49      volatile desc_t& dv = *static_cast<desc_t *>(p);
50      desc_t d = const_cast<desc_t&>(dv);
51      if (x != this->value() || desc_checksum(d) != 0)  // cs is garbage.
52        continue;
53
54      if (c == one) {
55        return d.o1;
56      } else if (c == two) {
57        return d.o2;
58      } else {
59        abort();
60      }
61    }
62  }
```

On the other hand, if the transactional pointer itself is non-volatile (which means it is not shared with any other threads), then the reader could use the simpler method.

```
63  Tp *access() const throw() {
64    assert(this->value() != traits_type::invalid);
65    return ptr_of_int<Tp>(this->value());
66  }
```

The public accessors `expect()` and the non-volatile `reset()` are simple.

```
67  private:
68   typedef linear_base<linear_trans_ptr<Tp>, traits_type> super_type;
69
70  public:
71   typedef Tp element_type;
```

```
72   typedef void *expect_type;
73
74   expect_type expect() volatile throw() {
75     return static_cast<expect_type>(this->access());
76   }
77
78   void reset(expect_type expect) throw() {
79     this->value() = int_of_ptr(expect);
80   }
```

The volatile `release()` method cannot simply swap out the value like before. It has to observe the writer lock, which is held by another thread whenever the color is non-*zero*. If the writer lock is not observed, the current thread could still swap out the linear pointer atomically with nil and release it, but the other writer in the middle of a swap might magically restore the linear pointer as remembered before the release, causing the linear pointer to reappear, which would be incorrect. However, unlike a writer swapping between two memory locations, the release writer here completes the operation without acquiring a writer lock.

```
81   Tp *release() volatile throw() {
82     while (true) {
83       uintptr_t old = this->value();
84       if (color_of_int(old) != zero)
85         continue;
86       if (atomic_compare_and_exchange_bool_acq(
87               &this->value(), traits_type::nil, old) == 0)
88         return ptr_of_int<Tp>(old);
89     }
90   }
```

The `swap()` method begins by creating a descriptor that retains the old values of the memory locations being swapped so that other readers may access the old values in a non-blocking fashion, attempts to lock these locations with a colored pointer to the descriptor, and finally commits the change if both memory locations are locked. If one lock succeeds and the other one fails, the first lock is reverted if it has not

been changed. This avoids deadlock if two writers swap the memory locations in the opposite order. There is a theoretical possibility of livelock if these two threads simultaneously acquire the first lock, fail to acquire the second lock, and release the first lock. However, this requires the two threads to be perfectly synchronized, an occurrence that does not happen in practice. A race would typically be the tie-breaker.

```
91  swap(volatile linear_trans_ptr& that) volatile throw() {
92    desc_t d;
93    d.a1 = &this->value(), d.a2 = &that.value();
94
95  retry:
96    d.o1 = this->access(), d.o2 = that.access();
97    d.cs = 0; d.cs = desc_checksum(d);
98    assert(desc_checksum(d) == 0);
99
100   // This ensures that whichever thread that traverses to our
101   // descriptor from d.a1 gets the correct value for d.o1.
102   if (atomic_compare_and_exchange_bool_acq(
103           d.a1, int_of_ptr(&d, one), int_of_ptr(d.o1)) != 0)
104     goto retry;
105
106   // This ensures that whichever thread that traverses to our
107   // descriptor from d.a2 gets the correct value for d.o2.
108   if (atomic_compare_and_exchange_bool_acq(
109           d.a2, int_of_ptr(&d, two), int_of_ptr(d.o2)) != 0) {
110     // Could not acquire a2; revert a1.
111     atomic_compare_and_exchange_val_acq(
112         d.a1, int_of_ptr(d.o1), int_of_ptr(&d, one));
113     goto retry;
114   }
115
116   // Since access() does not modify *a1 nor *a2, this means by now
117   // we have exclusive access to both memory locations.  Readers will
118   // make progress while writers will block.
119   *d.a2 = int_of_ptr(d.o1);
120   *d.a1 = int_of_ptr(d.o2);
121 }
```

If livelock becomes an issue, it can be trivially avoided by adding the following preamble to `swap()` which sorts `this` and `that` by their memory addresses. That works be-

cause unconditional exchange is symmetric, i.e. `p.swap(q)` is the same as `q.swap(p)`.

```
if (&this->value() > &that.value()) {
  that.swap(*this);
  return;
}

// rest of swap() ...
```

For compare-and-swap `cswap()`, the method differs by using `expect` to lock the first memory address. Note, however, that livelock avoidance is not as straightforward because `cswap()` is not symmetric, unlike `swap()`. The expect value is pinned to the first memory location, so `p.cswap(q, expect)` is not the same as `q.cswap(p, expect)`. To prevent livelock, two different versions of `cswap()` would have to be implemented, one that acquires `this` before `that`, the other one acquires `that` before `this`. The actual `cswap()` would dispatch between the two versions depending on the two memory addresses. Here only one version is presented.

```
122  bool
123  cswap(volatile linear_trans_ptr& that, expect_type expect) volatile throw() {
124    assert(color_of_int(int_of_ptr(expect)) == zero);
125
126    desc_t d;
127    d.a1 = &this->value(), d.a2 = &that.value();
128
129  retry:
130    d.o1 = this->access(), d.o2 = that.access();
131    d.cs = 0; d.cs = desc_checksum(d);
132    assert(desc_checksum(d) == 0);
133
134    if (d.o1 != expect)
135      return false;
136    if (atomic_compare_and_exchange_bool_acq(
137          d.a1, int_of_ptr(&d, one), int_of_ptr(expect)) != 0)
138      return false;
139    if (atomic_compare_and_exchange_bool_acq(
140          d.a2, int_of_ptr(&d, two), int_of_ptr(d.o2)) != 0) {
141      // Could not acquire a2; revert a1.
142      atomic_compare_and_exchange_val_acq(
```

```
143             d.a1, int_of_ptr(d.o1), int_of_ptr(&d, one));
144         goto retry;
145       }
146       *d.a2 = int_of_ptr(d.o1);
147       *d.a1 = int_of_ptr(d.o2);
148       return true;
149     }
```

The value constructor, reference constructor, move constructor, value assignment, reference assignment, and other accessors and mutators including the dereference operator, are similar to `linear_chroma_ptr<>` and omitted here for brevity.

```
150 };
```

And this concludes the linear transactional pointer.

5.8 Towards Theorem Proving

In a previous section, the concurrent singly linked list example in Listing 5.10 motivated the use of controlled aliasing by creating a linear pointer from an expect value. The aliasing is not due to the fact that we intend to duplicate the ownership of the object, but rather because there needs to be a way to deal with two different outcomes, one where the compare-and-swap succeeds, and one where the compare-and-swap fails. We shall see that in $\mathcal{ATS}$, a programming language with theorem proving that has both classical and linear propositions, this aliasing is not necessary.

Here is a brief introduction to concepts in $\mathcal{ATS}$, assuming that the reader is familiar with an ML-styled language.

- A *proof term* is part of a program expression that are type checked with the rest of the program, but are stripped before translating the program to machine code.

- A *view* is the type of a linear proof term. A *prop* is the type of a classical proof term. These are analogous to the way a *type* characterizes a program expression.

- The `dataview` syntax defines the sum-type constructors for linear proof terms, in the same way `datatype` defines the sum-type constructors for program objects.

- The view `t @ l` (the "`@`" symbol is an infix type operator) is a type of a linear proof term that says there is a value of type `t` at some address `l`. There is no algebraic constructor for such linear proof, but the proof term is generally obtained from somewhere else, e.g. a memory allocator which is an external function.

- A `t@ype` is a type that has a specific size. C primitive types like `char`, `int`, `long`, `float`, and `double` are all represented as their own respective `t@ype` in ATS.

With these simple concepts, it is possible to formulate the type of some external function that performs compare-and-swap—hopefully atomically, although the type does not reflect nor enforce that.

```
1 dataview cswap_result (bool, v1: view+, v2: view+) =
2   | cswap_succ (true, v1, v2) of v1
3   | cswap_fail (false, v1, v2) of v2
4
5 fun {t1, t2: t@ype | sizeof t1 == sizeof t2}
6   cswap {l: addr} (pf: t1 @ l | mem: ptr l, newval: t2, oldval: t1) :<>
7     [b: bool] (cswap_result (b, t2 @ l, t1 @ l) | bool b)
```

The type for the `cswap` function describes the sequential semantics of compare-and-swap. Given a memory location `l` which currently holds some value of type `t1` (i.e.

`t1 @ l`) the objective is to replace the value of type `t1` with a value of type `t2`. If the operation is successful, then we have `t2 @ l`, otherwise we still have `t1 @ l`.

The reason cswap exchanges two types and not just two values of the same type is because it allows the use of dependent types in $\mathcal{ATS}$ to track in a more refined way how values are swapped. With dependent types, the integers `0` and `1` have the type `int 0` and `int 1` respectively. An unknown integer has the type $\exists n.\text{int}\ n$, or written as `[n: int] int n` (the first `int` inside the square brackets actually refers to an integer *sort* which serves the purpose of characterizing the index to the `int` type, and they happen to have the same name). The formulation of swapping between `t1 @ l` and `t2 @ l` expresses precisely how the dependent-typed values are exchanged. Of course, the prerequisite is that `t1` and `t2` must have the same size in memory.

The outcome of the conditional swap is indicated by its return value, which is of the type $\exists b.\text{bool}\ b$, or written as `[b: bool] bool b`. Moreover, the index b is used to choose one of the two linear proof terms of the `cswap_result(b, t2 @ l, t1 @ l)` view: `cswap_succ` which gives us `t2 @ l` in case `b` is true, and `cswap_fail` which gives us `t1 @ l` in case `b` is false.

Here is how a singly linked list may be defined in $\mathcal{ATS}$.

```
8  viewtypedef node_struct (t: viewt@ype, next: addr) = @{ x = t, next = ptr next }
9
10 dataview list (t: viewt@ype+, int (* len *), addr (* first *)) =
11   | list_nil (t, 0, null) of ()
12   | {len: nat} {first: agz} {next: addr}
13     list_cons (t, len + 1, first) of
14       (node_struct (t, next) @ first, list (t, len, next))
15
16 viewdef list (t: viewt@ype, first: addr) = [len: nat] list (t, len, first)
17 viewdef list (t: viewt@ype) = [first: addr] list (t, first)
```

We begin with a node structure that has two members, `x` which holds the item (a viewtype is a type that may carry a linear view), and `next` which holds a pointer

to the memory location of the next node. The linear view `list (t, len, first)` describes the logical structure of a singly linked list. A linear proof of that view is built from the base case `list_nil` which is the empty list of length zero represented by the `null` pointer, and inductively using `list_cons` which combines the rest of the list `list (t, len, next)` with a first node `node_struct(t, next)` whose next pointer points to the rest of the list. The node is located at address `first`, and the resulting list `list (t, len + 1, first)` is one item longer. Here `{first: agz}` is a shorthand for `{first: addr | first > null}`, i.e. a memory address that is greater than zero. The shorthands `list (t, first)` and `list t` allow us to use the list type without caring about list length, or without caring about both the list length and where the first node of the list is.

Recall that `insert_at` in C++ has the following declaration.

```
static void insert_at(ptr_type node, volatile ptr_type& curr) throw();
```

In $\mathcal{ATS}$, it could be given the following type.

```
18 fun {t: viewt@ype} insert_at {node: agz} (
19   node_pf: list (t, 1, node) | node: ptr node,
20   curr_ref: ref @(list t | ptr)) :<!ntm,!ref> void
```

where a node is described by a linear view `list (t, 1, node)` testifying the existence of a single-item list at location `node`, accompanying a pointer `ptr node` to such list. The current cursor for insertion becomes `ref @(list t | ptr)` which is a reference to some list pointer that carries a proof of the list. The volatile qualifier is lost in translation. The `insert_at` function has two side-effects: `!ntm` for non-termination, and `!ref` for modifying shared reference.

In $\mathcal{ATS}$, `ref` is really an abstract type parameterized by `t` which is the `viewt@ype` of the target item. Dereferencing is accomplished by casting `ref` to a pointer and a "view box" that stores a linear proof testifying the item's existence. A view box is a classical proposition carrying a linear resource, and it is similar to the modality

operator in linear logic that allows controlled sharing. When a view box is opened in a lexical scope, the linear proof inside can be temporarily borrowed, but another linear proof of the same view must be put back before the scope ends.

```
21 abstype ref (t: viewt@ype)
22 absprop vbox (v: view)
23
24 castfn deref {t: viewt@ype} (r: ref t) :<> [l: agz] (vbox (t @ l) | ptr l)
```

Unfortunately, `deref` is not useful in the case where we want to implement a non-blocking version of insertion. When applying `deref` on a value of type `ref @(list t | ptr)`, it returns a view box with the view `@(list t | ptr) @ l` and a pointer `ptr l`, which means that both the proof of `list t` (which takes no size) and the list pointer `ptr` are both located at memory address `l`. The reason that `deref` is not useful is because, in this setting, both the proof `list t` and the list pointer `ptr` must be stored and retrieved together, via `ptr l`.

After retrieving and unpacking `@(list t | ptr)`, the proof for `list t` is used as part of `list_cons`, and the value of `ptr` is assigned to the next pointer of the node. If the compare-and-swap fails, the view box now requires us to fold back the `list t` proof while storing the same value of `ptr` back to the reference. We must not do that because the current pointer to the volatile list must have been changed to some other value, which is the reason why the swap failed in the first place!

To allow the proof to be put back into the view box without altering the pointer, it is necessary to treat `list t` and `(ptr @ l | ptr l)` separately. Also, in order to actually build the new list, we need a more precise relation between `list t` and `ptr`, namely that the pointer actually points to the list. Here is a first try.

```
25 castfn list_deref {t: viewt@ype} (r: ref @(list t | ptr)) :<>
26   [first: addr] [l: agz] (vbox @(list (t, first), ptr first @ l) | ptr l)
```

The cast function `list_deref` allows us to make an ephemeral assumption (as long

as the view box remains open) that there is a proof for `list (t, first)` whose first node is at address `first`, and there is a pointer `ptr first` to this list at another location `l`. However, there is a problem. The type of the view box is too specific. Recall that in order to close the view box, the exact same view has to be put back in. The formulation here prohibits `ptr first` from ever pointing to a new location, otherwise the view `ptr first @ l` would change. If the insertion succeeds, it should become `ptr node @ l`.

The trick is to move the existential quantifiers ∃first and ∃l inside the view box. The view `list_ptr_at` defined below encapsulates the relationship between `list (t, len, first)` and `ptr first @ l`, and allows existential quantifier for `len` and `first` to be placed inside the view box, which means that list insertion can now both change the list length and modify the list pointer to a new location.

```
viewdef list_ptr_at (t: viewt@ype, len: int, first: addr, l: addr) =
  (list (t, len, first), ptr first @ l)

viewdef list_ptr_at (t: viewt@ype, l: addr) =
  [len: nat] [first: addr] list_ptr_at (t, len, first, l)
```

And finally, the fixed version of `list_deref` is given as follows.

```
castfn list_deref {t: viewt@ype} (r: ref @(list t | ptr)) :<>
  [l: agz] (vbox (list_ptr_at (t, l)) | ptr l)
```

To get a sense of what is required to implement the concurrent singly list insertion in $\mathcal{ATS}$, let us study the non-concurrent version first. The basic framework is to setup a loop as a recursive function that would retry in case of failure, though for the non-concurrent version the loop executes only once. The body of the loop consists of dereferencing the volatile list reference, unpacking the proof, assigning to pointers, constructing the proof of the new list, and placing the proof back to the view box.

```
implement {t} insert_at (node_pf | node, curr_ref) =
let
  fun loop {node: agz} (node_pf: list (t, 1, node) | node: ptr node)
```

```
4         :<!ntm,!ref> void =
5     let
6       val [l: addr] (vbox curr_pf | curr_ptr: ptr l) = list_deref curr_ref
7       val (curr_list_pf, curr_ptr_pf) = curr_pf
8       val list_cons (node_at_pf, list_nil ()) = node_pf
9     in
10      !node.next := !curr_ptr;
11      !curr_ptr := node;
12      curr_list_pf := list_cons (node_at_pf, curr_list_pf);
13      curr_pf := (curr_list_pf, curr_ptr_pf);
14    end
15  in loop (node_pf | node) end
```

It might be helpful to have an idea how each of the assignment statement affects the view. In $\mathcal{ATS}$, the prefix operator `!` in a dynamic expression is analogous to the prefix operator `*` in C/C++ which converts a pointer to an l-value.

- For `!node.next := !curr_ptr`, the view for `node_at_pf` changes from `node_struct (t, null) @ node` to `node_struct (t, curr) @ node` where `curr` is the location of the current list's first node. Note that `curr_ptr_pf` has the view `ptr curr @ l`, so `!curr_ptr` reads out a value of the type `ptr curr`.

- For `!curr_ptr := node`, the view for `curr_ptr_pf` changes from `ptr curr @ l` to `ptr node @ l`.

- For `curr_list_pf := list_cons (node_at_pf, curr_list_pf)`, the view for `curr_list_pf` changes from `list (t, len, curr)` to `list (t, len + 1, node)`.

- Finally, for `curr_pf := (curr_list_pf, curr_ptr_pf)`, the view of `curr_pf` changes from `list_ptr_at (t, len, curr, l)` to `list_ptr_at (t, len + 1, node, l)` which allows the view box `vbox curr_pf` to close.

The concurrent, non-blocking version of `insert_at` is similar, except it uses `cswap` and has to determine the outcome of compare-and-swap.

Listing 5.12: Concurrent, non-blocking singly linked list insertion in $\mathcal{ATS}$

```
implement {t} insert_at (node_pf | node, curr_ref) =
let
  fun loop {node: agz} (node_pf: list (t, 1, node) | node: ptr node)
      :<!ntm,!ref> void =
  let
    val [l: addr] (vbox curr_pf | curr_ptr: ptr l) = list_deref curr_ref
    val (curr_list_pf, curr_ptr_pf) = curr_pf
    val list_cons (node_at_pf, list_nil ()) = node_pf

    val expect = !curr_ptr
    val () = !node.next := expect

    val (cswap_pf | result) = cswap (curr_ptr_pf | curr_ptr, node, expect)
  in
    if result then let
      val cswap_succ (curr_ptr_pf (* ptr node @ l *)) = cswap_pf
    in
      curr_list_pf := list_cons {t} (node_at_pf, curr_list_pf);
      curr_pf := (curr_list_pf, curr_ptr_pf);
    end

    else let
      val cswap_fail (curr_ptr_pf (* ptr curr @ l *)) = cswap_pf
    in
      !node.next := null;
      node_pf := list_cons (node_at_pf, list_nil ());
      curr_pf := (curr_list_pf, curr_ptr_pf);
      $effmask_ref (
        loop (node_pf | node)
      );
    end
  end
in loop (node_pf | node) end
```

It is worthwhile to note that `curr_ptr_pf` in the success case and the failure case have different view. In the success case, we have a proof of the view `ptr node @ l`, so we use it to construct a proof of the new list and put the proof back to the view box. In the failure case, we still have a proof of the view `ptr curr @ l`, which forces

us to roll back the proof before we can close the view box. Failure to roll back will cause a type error; the reader could verify this by removing the three assignments in the failure case (and keep the recursive call).

At the moment, it might seem that $\mathcal{ATS}$ has all the theorem proving facility to reason about the correctness of a concurrent algorithm, but it is not the case. Notice that both the non-concurrent and the concurrent implementations of `insert_at` pass type-checking under the same set of axioms. The reason, as mentioned earlier, is that the volatile qualifier is lost in translation. Although we temporarily assume that the current list has a certain proof and certain pointer value, this assumption is not materialized until compare-and-swap tests it. A possible future direction is to not close the view box by lexical scope, but require the call to `cswap` to close it. This gives `cswap` a transactional semantic.

5.9 Conclusion

Going back to the question how much linearity can be preserved in a runtime system where linear objects are passed between concurrent threads, it is clear that primitives such as conditional and unconditional exchange do preserve linearity, and a practical, linearity-preserving double-ended queue can be built using these primitives. However, the correctness of a concurrent algorithm is much more than linear object ownership semantics. The structural invariants of the deque demonstrates that much of the correctness arguments are not related to linearity at all. In fact, a linearity preserving concurrent algorithm can still be incorrect, such as naively removing a node from a singly linked list, which is prone to the ABA problem. Using a stronger primitive such as compare-and-swap of two volatile memory locations remedies this, but the correctness of such primitive also cannot be reasoned with linearity alone.

Attempting to codify the structural invariants of a concurrent data structure using an axiomatic proof system presents a different challenge. Many such proof systems assume that objects are immutable, although object mutation can be expressed with framing. It can be understood that linear logic achieves framing using a specific interpretation of sequent calculus. Even so, object state changes are assumed to be explicitly done by the program's current control flow. However, it is not the case with concurrent algorithms, where certain objects are volatile and can change at any time by an external process. This short-coming is evident with the $\mathcal{ATS}$ formulation of singly linked list insertion where the same set of axioms will admit both the non-concurrent and concurrent versions, when only the concurrent version should have been admitted.

Chapter 6

Conclusion

This dissertation introduces a noval technique for memory management called linear pointer and shows that it is a simple, safe, and efficient way to do memory management. Its simplicity is due to the fact that it uses existing C++ language constructs and requires no other extensions to the language. Its safety is due to the fact that it is based on linear object ownership semantics which is a well-understood programming discipline. Its efficiency is due to the erasure property that runtime linear checking can be disabled and will result in a program with identical behavior and no overhead. A memory allocator is implemented using linear pointer to provide the foundation of a safe memory management. It also proves that a practical program can be built using linear pointers and achieves comparable performance to other state of the art memory allocators. Although other methods exist to ensure memory safety, linear pointer is the sweet spot that simultaneously achieves simplicity, safety and efficiency.

This work is largely motivated by the need to understand how to use linear ownership semantics to reason about volatile invariants required to express non-blocking algorithms. For future work, I propose that view box in $\mathcal{ATS}$ is a possible way to temporarily freeze volatile invariants, but we require a transactional semantics of

closing view box by the compare and swap operation rather than closing it by the lexical scope.

Appendix A

Comparing Memory Management Scalability Using Cilk and JCilk

Cilk is a parallel programming language developed at MIT Laboratory of Computer Science (now Computer Science and Artificial Intelligence Laboratory), as a culmination of over two decades of research that began with the development of Thinking Machines Corporation's Connection Machine CM-5 in 1992, designed by Charles Leiserson, Bradley C. Kuszmaul, et al [86, 79]. The CM-5 was a machine connecting on the order of 256 or more processors, using fat-tree network topology, which structures the connection like a tree with lower bandwidth connections towards the leaf processors and higher bandwidth connections towards the trunk. As many computation problems use divide and conquer strategy to break down a large problem into smaller sub-problems, the CM-5 topology is a natural fit to run such computations.

Cilk [17, 73] lends its origin from Parallel Continuation Machine model [61] developed by Michael Halbherr et al, which implemented a dialect of the C language with parallel computation in continuation passing style on CM-5. Continuation passing was first described by John Reynolds [99] and used by Gerald Sussman and Guy Steele

Jr. to implement Scheme programming language in 1975 [115, 109]. Cilk improved upon Parallel Continuation Machine by implementing a work-stealing scheduler that is proven to be able to complete a parallel execution within a constant factor of optimal time.

The language improved in usability over the years, most notably adding support for software shared-memory with relaxed *direct acyclic graph* (DAG) consistency, called inlets, which makes computation that involves global data structure easier to program. Finally, Cilk-5 [52] was an implementation that targeted hardware shared-memory architecture. A distributed version of Cilk-5 that leverages a cluster of shared-memory machines with relaxed memory consistency was presented by Keith Randall [98]. Non-distributed Cilk-5 is the basis of Cilk++ (a dialect of C++) [85] which is now developed at Intel, as well as JCilk (a dialect of Java) [36, 84, 37] which continues to be developed at MIT.

In a nutshell, Cilk introduces three main extensions to the C language: a *spawn* operation which spins off computation of a sub-problem to a lightweight process, a *sync* opertion which waits for the result of a lightweight process, and an *abort* operation for cancelling a whole branch of spawned computation, which is useful for speculative parallelism. A lightweight process is a continuation passing style closure which represents a unit of work to be distributed among multiple worker threads each running on its own processor. When a parent lightweight process spawns a child, the parent's continuation is placed on a work queue, and the computation proceeds onto the child. The work queue is a double-ended queue, which means that items can be pushed and popped from the top like a stack, but the queue also supports a deque operation that removes an item from the bottom.

In the normal case, when the child computation finishes, the child would simply return back to the parent, so the work queue behaves like a call stack. While the child

is being computed, the most ancient ancestor continuation in the queue (which could also be the child's immediate parent) may also be stolen by another idle worker. This is called work-stealing scheduling, and it is how work is distributed among workers. If we consider parallel computation as a tree, most computation is carried out in depth first traversal like a sequential program, except when work is stolen from closer to the root of the tree. The reason is that work stealing incurs synchronization overhead, and the fact that child computation is performed first means that this synchronization overhead is edged outside of the critical path of the computation, which guarantees the shortest overall running time. This is called work-first principle.

Under the hood, both Cilk and JCilk are source-to-source translators. Cilk converts a program written in the C language with spawn and sync keywords to an ANSI-C source code that uses Cilk runtime scheduling primitives for creating closures and manipulating the work queue. JCilk converts a program written in the Java language with spawn and sync keywords to Java source code extended with the goto-statement, which is then compiled into Java byte code. This Java byte code can be executed on any Java virtual machine which already has the jump instruction. Both Cilk and JCilk implement lightweight processes using continuation passing closures and work-first principle with work-stealing scheduling. Although C and Java use different compilation strategies with various levels of optimization, the difference is normalized away when we consider only scalability but not the absolute performance.

There is a subtle difference in the work queue implementation. Cilk uses the THE (tail-head-exception) protocol, which avoids the need for both thief and victim workers to acquire lock for accessing the queue in the common case. The lock is only acquired when both the thief and the victim are competing to remove the same continuation from the work queue (i.e. the queue has exactly one item). The protocol requires the use of memory barrier or sequential consistency memory model in

order for both the thief and victim to reliably detect contention, but does not incur hardware synchronization cost in the common case. Java's memory model supports atomic operations which incur hardware synchronization cost but also act as a memory barrier. This is used by JCilk instead to implement the work queue, so there may be hardware synchronization cost in the common, non-contention case.

A.1 Description of Memory Management Methods

Due to the similarity of Cilk and JCilk parallel computing model and scheduling implementation, they are a convenient dual to compare parallel program scalabity on manual memory allocation (Cilk) and garbage collection (JCilk). We are mostly interested in the speed-up when running with multiple workers compared to running a single worker. Cilk uses an internal allocator. The garbage collector used by JCilk depends on the Java virtual machine configuration. We used Java 1.7 with the Parallel collector [114] and Garbage First collector [43]. Here is a brief description of the memory management methods under the benchmark.

Cilk memory allocation For the purpose of allocating internal data structures, Cilk uses a layer of memory allocation that caches objects in per-worker free lists (also per-processor, since Cilk typically starts as many workers as there are processors). There are 9 object size classes from 2^4 through 2^{12} bytes in powers of two, and each size class has its own per-worker free list. Larger objects are obtained from the system's `malloc()`. When the worker's local free list is exhausted, the worker transfers a batch of objects from the global free list of the same size class. All global free lists are protected by one lock, but accessing the global free list should occur relatively

infrequently compared to the worker's local free list. When the global free list is exhausted, an object of the requested size is carved out of a global pool by bumping an allocation pointer. The global pool is a *region* in the sense of region-based memory management. Objects have no header that might allow the allocator to record the object size. The pool itself is a collection of 32KB pages, and the pool is extended 32KB at a time when the current page runs out of space. Ultimately, the allocator allocates the 32KB pages from `malloc()`.

It is notable that Cilk internal free function expects the caller to inform the allocator the size of the object being freed, since the allocator does not keep a record of object sizes in a header or by other means. Free simply returns the object back to the free list of the appropriate size class, but keeps at most a batch size of objects. Batch size is the same for all object sizes, configurable through a command line flag. When the free list has more than a batch size number of objects, half of the batch is transferred to the global free list. On the other hand, during allocation, if the local free list is exhausted, then half of the batch size number of objects is transferred from the global free list to the local free list. The global free list grows indefinitely, and memory used by the global pool is never returned to the system.

This allocator does not actively prevent false sharing. Although objects obtained in a batch from the global pool are contiguous, they can be dispersed to different workers during work stealing. When a worker steals objects from various other workers and subsequently frees the objects to the worker's local free list, the list now contains non-contiguous objects, which means that these objects have neighbors in the same cache line that do not belong to the same worker. Non-contiguousness of objects is propagated to the global free list when objects are released back from the local free list. The non-contiguous segment of the global free list can be allocated into a local free list again and then used by another worker.

Parallel collector The parallel collector is the parallelized version of the serial collector. The collector is a stop-the-world generational collector that performs copying for the young generation, and mark-sweep for the old generation. The only difference is that the copying and mark-sweep phases are distributed to parallel workers to speed up collection time. This collector has high throughput although the pause time can be unpredictable. This is the defualt collector for JVM.

Garbage First collector The Garbage First collector is a generational, concurrent-marking garbage collector with parallel evacuation. The collector is adaptive to an "ergonomic policy" tuned by the user to meet memory footprint and pause time, and throughput goals. The soft real-time pause time constraint is achieved by delimiting the heap into regions and copying (evacuating) one region at a time. The regions are conveniently also thread-local allocation buffers so the mutator threads can allocate memory within their own private regions in parallel. It is notable that Garbage First does not use size-class free list for allocation. Objects are allocated from the region using a bump pointer.

Scalability bottleneck is still present whenever the mutator threads are stopped. First, mutator threads are stopped to mark the roots, but further marking can be done concurrently afterwards using write barriers. Mutator threads are stopped again when marking is finished to ensure that there is no more pending marking activities before evacuation. Mutator threads are also stopped during the evacuation pause, but evacuation workload is dispatched to multiple workers in parallel, so the main bottleneck in this phase is the work scheduling implementation.

A.2 Scalability Metrics

The parallel structure of the program is the main decider of its scalability. Amdahl observed that scalability bottleneck is limited by the sequential part of the program that could not be parallelized [5]. Let $0 \leq s \leq 1$ be the fraction of the program that is inherently sequential, and P be the number of parallel processors. For the sake of argument, assume that total running time on one processor T_1 is $s+(1-s) = 1$ second. The total running time on P processors is $T_P = s + \frac{1-s}{P}$. If P tends to infinity, the total running time on infinite number of processors is $T_\infty = s$. The maximum possible speed up on infinite number of processors is $T_1/T_\infty = \frac{1}{s}$. Blumofe and Leiserson presented a graph-theoretical method [18] to derive T_1 and T_∞ when the structure of the parallel computation is given as a directed acyclic graph. They also derive the space consumption when running on a single processor S_1. The running time T_P can be compared to the running time of the sequential program T_S to quantify the benefit of parallelizing the program. Usually $T_1 > T_S$ because of the task scheduling runtime overhead.

In practice, the operating system measures running time in terms of elapsed wall-clock time τ, user time μ and system time σ, and the space consumption in terms of max resident set size (maxrss). User time is the amount the program uses for computation. System time is the amount the operating system spends on behalf of the program. In the case of computation-heavy program, the system time is typically spent in scheduling overhead or for making memory available to the program. When a program is multi-threaded, both user time and system time are the cumulative time across all threads. The elapsed wall-clock time is simply the time it takes for the program to run from start to finish. Therefore, the overall number of processors utilized is $P' = (\mu + \sigma)/\tau$.

Scalability bottleneck manifests on a shared-memory multi-processor machine in both measurable and immeasurable ways. When a thread of computation blocks waiting for another thread, the operating system does not count the blocked time towards user time because the blocked thread is not scheduled to run on any processor. However, this incurs a small amount of scheduling overhead as system time. The blocking reflects as lower utilization P' and is measurable. On the other hand, when a thread performs a busy-wait (i.e. spins to wait on the result rather than blocking), or when the memory bus is saturated (i.e. processor's execution pipeline stalls to wait for memory bus to become available), both are reflected as part of user time μ that cannot be easily told apart from the actual time spent on productive computation.

It is worthwhile to note that memory management can increase utilization without reducing the elapsed wall-clock time. This can happen when the allocator or garbage collector spawns a separate thread for memory management work while the program is running. This is the case for all concurrent garbage collectors by definition, but to one extreme it can be used to speed up single-threaded programs as well by allocating in a different thread, e.g. Herrmann and Wilsey [66].

A.3 Results

Among the many benchmarks for Cilk and JCilk, the naive Fibonacci sequence program is chosen. The program computes the following recursive function as written, except that in the parallel version, the two recursive calls are run in parallel:

$$\mathrm{Fib}(x) \quad = \quad \begin{cases} x & \text{if } x \leq 1, \\ \mathrm{Fib(x-1)} + \mathrm{Fib(x-2)} & \text{otherwise.} \end{cases}$$

Apart from the two parallel calls is one comparison, two subtractions and one addition. This program is an ideal benchmark because it does very little work outside of the runtime overhead, and it has a predictable parallel structure.

The experiment compares the following variants of the Fibonacci program.

- Fib compiled by Cilk 5.4.6 and GCC 4.3.6.
- Fib compiled by JCilk (2008 release) and Java 1.7.0_02
  - Ran with the Parallel Scavenging collector with -XX:MaxNewHeap of 8MB, 16MB, 32MB, and 64MB, as well as unspecified size.
  - Ran with the Garbage First collector with unspecified MaxNewHeap size.

Each variant is run with 1, 2, 4, 8, 16, 24, and 32 workers, as well as the sequential elison "S" which is compiled using the same GCC or Java without making Cilk calls. The difference between the sequential elison "S" and running with one worker is that the one-worker version still makes the recursive calls as if they are meant to be scheduled in parallel, whereas the sequential elison performs a straight non-Cilk function call.

The results are collected from a 32-core NUMA machine with quad 8-core 2.4GHz AMD Opteron 6136 processors and 256GB DDR3-1333 RAM, running Scientific Linux release 6.4 on the 2.6.32-358.6.2.el6.x86_64 kernel. Each run consists of 100 iterations, and the numbers reported are the means and the standard deviations of the run.

Table A.1 shows the elapsed wall-clock time, and Figure A.1 shows a plot of the speed up T_1/T_P based on elapsed wall-clock time. While Cilk is able to achieve near linear speed-up, all JCilk variants show scalability problems depending on allowed new heap sizes, with smaller heaps the least scalable and larger heaps more scalable for Parallel Scavenging collection. However, Garbage First collector experiences

Variant	S	1	2	4	8	16	24	32
Cilk	4.5	60.2	29.8	15.0	7.5	3.8	2.5	1.9
±	0.0	0.2	0.2	0.3	0.0	0.0	0.0	0.0
JCilk 8M	7.7	201.8	162.2	144.7	114.5	102.6	114.5	122.4
±	0.2	5.4	44.0	2.2	5.2	1.9	1.8	1.9
JCilk 16M	7.7	154.1	116.5	88.6	72.7	63.1	60.5	63.8
±	0.1	3.0	41.3	11.2	3.5	3.2	1.4	1.2
JCilk 32M	7.7	134.3	98.7	65.7	44.4	38.9	36.9	37.1
±	0.2	1.3	57.6	8.6	2.8	1.3	1.0	0.9
JCilk 64M	7.7	129.0	84.7	54.2	31.3	25.0	23.1	23.1
±	0.1	4.3	31.2	9.2	2.8	0.9	0.3	0.3
JCilk ∞	7.7	117.2	75.4	38.9	20.4	11.4	8.6	8.1
±	0.0	3.0	37.8	14.5	3.8	1.2	0.7	0.7
JCilk G1	7.7	123.2	72.9	43.9	27.5	22.9	24.8	29.5
±	0.0	3.4	14.7	5.1	2.3	0.6	1.0	0.5

Table A.1: Elapsed wall-clock time.

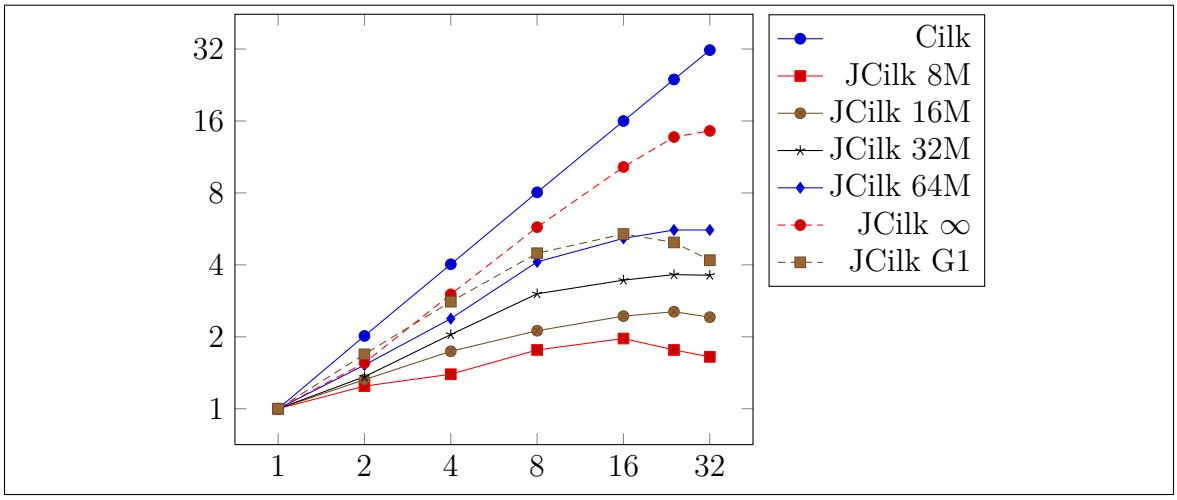


Figure A.1: Speed up T_1/T_P based on elapsed wall-clock time.

Variant	S	1	2	4	8	16	24	32
Cilk	1.68M	2.69M	2.83M	6.44M	11.8M	22.1M	27.1M	36.8M
±	6.96K	5.58K	5.52K	4.42M	3.83M	4.22M	7.12M	9.21M
JCilk 8M	119M	605M	960M	1.55G	2.52G	3.70G	4.68G	5.56G
±	4.79M	12.1M	10.9M	11.0M	21.1M	59.5M	77.1M	125M
JCilk 16M	118M	404M	552M	851M	1.33G	2.05G	2.52G	3.15G
±	5.17M	4.77M	7.72M	6.57M	10.2M	13.6M	17.2M	46.3M
JCilk 32M	119M	386M	461M	589M	849M	1.23G	1.52G	1.77G
±	4.30M	8.71M	11.6M	10.9M	10.2M	14.2M	13.8M	18.4M
JCilk 64M	120M	467M	495M	555M	684M	874M	1.01G	1.16G
±	3.87M	8.58M	9.29M	8.59M	11.1M	12.9M	11.5M	12.9M
JCilk ∞	121M	7.10G	17.2G	32.3G	39.7G	40.1G	40.1G	40.1G
±	3.51M	4.40G	8.24G	6.65G	1.35G	32.3M	17.2M	9.89M
JCilk G1	204M	7.44G	8.15G	9.57G	9.01G	10.3G	13.5G	11.5G
±	4.92M	5.52M	3.14G	5.14G	4.50G	5.78G	7.11G	6.51G

Table A.2: Maximum resident set size.

scalability problems even with unspecified new heap sizes.

Although the inner-workings of a garbage collector is difficult to understand, other metrics can shed lights on what might be causing scalability problems. Table A.2 shows that the garbage collector took advantage of larger heap sizes and used more memory (above several orders of magnitude!). One can theorize that the collector does not have to collect as often if it has more memory to use. Indeed, Table A.3 confirms this theory, showing that as heap size increases, the number of seconds for young generation collection also decreases. This statistics was collected using -verbosegc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps flags passed to JVM. None of the runs had incurred collection of the old or permanent heaps. Not only that more threads used more memory, it also increased variability of memory usage. This can be seen across the board even with Cilk.

Table A.4 shows the CPU utilization $(\mu + \sigma)/\tau$ computed from the user time μ, system time σ, and the elapsed wall-clock time τ. This provides insight about

Variant	S	1	2	4	8	16	24	32
JCilk 8M	0.0	50.3	48.3	50.7	55.8	63.9	76.3	79.6
±	0.0	4.1	0.3	0.7	1.0	0.5	1.5	1.9
JCilk 16M	0.0	24.2	24.3	25.1	26.7	29.9	31.9	34.3
±	0.0	0.7	0.4	0.4	0.3	0.6	0.6	0.7
JCilk 32M	0.0	12.4	12.4	15.9	11.3	12.3	12.8	13.2
±	0.0	0.3	0.9	2.7	0.1	0.2	0.2	0.2
JCilk 64M	0.0	7.9	9.0	7.8	5.5	5.4	5.0	4.9
±	0.0	1.7	2.8	0.5	0.4	0.1	0.1	0.2
JCilk ∞	0.0	0.8	0.3	0.2	0.2	0.1	0.1	0.1
±	0.0	0.2	0.1	0.0	0.0	0.0	0.0	0.0
JCilk G1	0.0	2.1	2.0	1.9	1.9	1.8	1.8	2.1
±	0.0	0.1	0.1	0.2	0.2	0.1	0.2	0.3

Table A.3: Number of elapsed wall-clock seconds garbage collecting the young generation.

Variant	S	1	2	4	8	16	24	32
Cilk	1.0	1.0	2.0	4.0	8.0	16.0	23.9	31.6
±	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1
JCilk 8M	1.0	2.4	3.2	4.7	7.3	10.0	11.4	12.7
±	0.0	0.4	0.2	0.1	0.1	0.1	0.2	0.3
JCilk 16M	1.0	1.8	2.9	4.6	7.2	10.8	12.6	12.7
±	0.0	0.0	0.1	0.1	0.1	0.2	0.2	0.2
JCilk 32M	1.0	1.6	2.8	4.9	7.5	11.9	15.2	17.2
±	0.0	0.0	0.2	0.2	0.1	0.2	0.2	0.3
JCilk 64M	1.0	1.4	2.7	4.5	7.5	13.0	17.8	20.7
±	0.0	0.1	0.2	0.1	0.0	0.1	0.2	0.5
JCilk ∞	1.0	1.0	2.0	3.9	7.7	14.5	20.5	22.8
±	0.0	0.0	0.0	0.1	0.1	0.3	0.4	0.8
JCilk G1	1.0	1.1	2.1	4.1	7.6	13.9	18.6	17.2
±	0.0	0.0	0.0	0.0	0.1	0.3	0.4	0.8

Table A.4: CPU utilization $(\mu + \sigma)/\tau$.

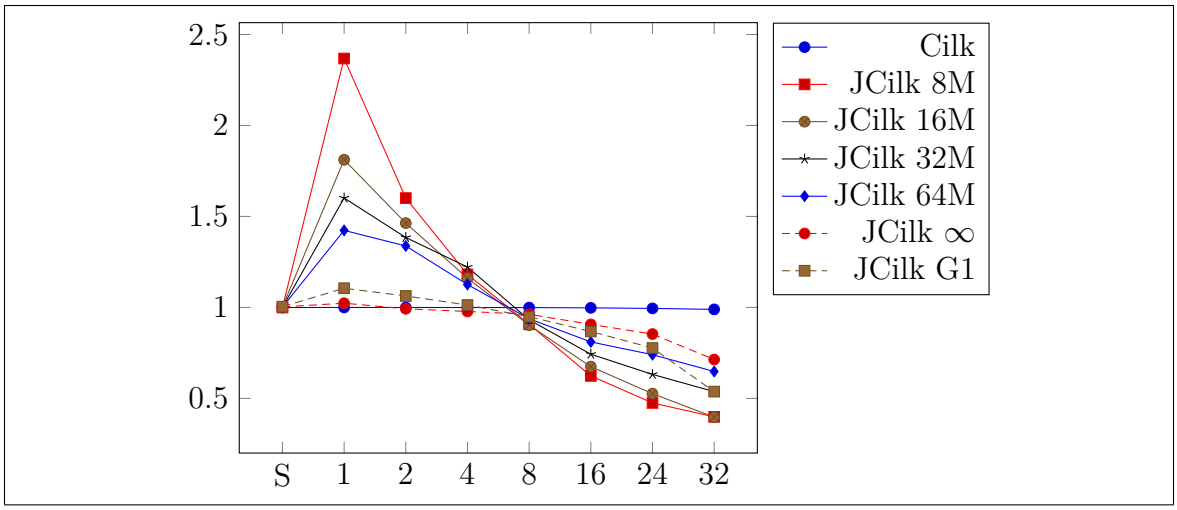


Figure A.2: Utilization normalized to the number of workers.

scalability bottleneck. While Cilk keeps all CPUs almost fully utilized, this is not the case with JCilk. Our results show that utilization is lower when the new heap size is smaller and the garbage collector has to work more. Figure A.2 shows the utilization relative to the number of workers. It shows that garbage collection could super-utilize CPUs by offloading the work to additional threads, but the advantage of offloading disappears for 6 or more workers. The lower utilization can be explained by the stop-the-world synchronization. The more threads there are, the longer is the wait for all threads to enter the stop state.

A.4 Conclusion

Cilk and JCilk are used to compare memory management scalability because of the similarity of the scheduling algorithm and programming paradigm. Although Cilk is compiled as C and JCilk is compiled as Java, both speed-up and utilization normalize away language differences such as compilation strategy and optimization. The only controlled difference is memory management.

Cilk uses an internal malloc() and free() based on thread-local free lists, where

as JCilk uses Java's built-in garbage collectors. In our JCilk runs, we varied the max new heap size of the Parallel Scavenging collector and also compared it against the Garbage First collector without a max new heap size. We chose the Fibonacci number benchmark because it performs very little work beside the runtime scheduling overhead, so it makes an ideal scalability benchmark.

Our results show that garbage collection is the scalability bottleneck. By lowering new heap size, garbage collection does more work, and it impaired both speed-up as well as utilization.

Appendix B

Linear Pointer in C++11

The work on this dissertation predated C++11 which introduced new constructs that could streamline the implementation of linear pointer. Most notably, rvalue constructor and rvalue assignment allow moving linear ownership from a temporary pointer, e.g. one that is returned from a function call. This eliminated the need to introduce an intermediary reference type as well as reference construction and assignment. C++11 also introduced atomic primitives that allow a linear pointer to feature atomic operations more easily. This appendix gives a listing of the streamlined linear atomic pointer using C++11 features.

```
1  #include <atomic>
2  #include <cassert>
3
4  template<typename Tp>
5  struct value_traits {
6    typedef Tp value_type;
7    static const Tp nil;
8    static const Tp invalid;
9  };
10
11 template<typename Tp>
12 const Tp value_traits<Tp>::nil = (Tp) 0lu;
13
```

```
14 template<typename Tp>
15 const Tp value_traits<Tp>::invalid = (Tp) ~0lu << 12;
16
17 template<
18   typename Tp,
19   class Traits = value_traits<Tp *> >
20 class linear_ptr : protected Traits {
21  protected:
22   using Traits::nil;
23   using Traits::invalid;
24
25  public:
26   typedef Tp element_type;
27   typedef Tp * pointer;
28
29   explicit linear_ptr(pointer q = nil) throw()
30     : p(q) {}
31   linear_ptr(linear_ptr& that) throw()
32     : p(that.get()) {}
33   linear_ptr(linear_ptr&& that) throw()
34     : p(that.get()) {}
35
36   ~linear_ptr() throw() {
37     assert(this->p == invalid || this->p == nil);
38   }
39
40   linear_ptr& operator=(linear_ptr& that) throw() {
41     this->reset(that.get()); return *this;
42   }
43
44   linear_ptr& operator=(linear_ptr&& that) throw() {
45     this->reset(that.get()); return *this;
46   }
47
48   operator bool() const volatile throw() {
49     assert(this->p != invalid);
50     return this->p != nil;
51   }
52
53   pointer get() throw() {
54 #if NDEBUG
55     return this->p;
```

```
56 #else
57     assert(this->p != invalid);
58     pointer q = this->p;
59     this->p = invalid;
60     return q;
61 #endif
62   }
63
64   pointer release() throw() {
65     assert(this->p != invalid);
66     pointer q = this->p;
67     this->p = nil;
68     return q;
69   }
70
71   void reset(pointer q = nil) throw() {
72     assert(this->p == invalid || this->p == nil);
73     this->p = q;
74   }
75
76   Tp& operator*() const throw() { return *this->p; }
77   Tp* operator->() const throw() { return this->p; }
78
79   // Support for atomic operations.
80
81   typedef void *expected_type;
82
83   expected_type expected() volatile throw() {
84     return static_cast<expected_type>(this->p);
85   }
86
87   void reset(expected_type expected) throw() {
88     this->reset(static_cast<pointer>(expected));
89   }
90
91   pointer release() volatile throw() {
92     pointer q = this->p.exchange(nil);
93     return q;
94   }
95
96   bool cswap(linear_ptr& that, expected_type expected)
97       volatile throw() {
```

```
98      assert(that.p != invalid);
99      pointer old = static_cast<pointer>(expected);
100     if (this->p.compare_exchange_weak(old, that.p)) {
101       that.p = old;
102       return true;
103     }
104     return false;
105   }
106
107   void swap(linear_ptr& that) volatile throw() {
108     assert(that.p != invalid);
109     pointer q = this->p.exchange(that.p);
110     that.p = q;
111   }
112
113  protected:
114   std::atomic<pointer> p;
115 };
```

List of Journal Abbreviations

ACM Association for Computing Machinery

AFIPS American Federation of Information Processing Societies

CASES Compilers, Architecture, and Synthesis for Embedded Systems (ACM)

CISIS Complex, Intelligent and Software Intensive Systems

ESA European Symposium on Algorithms

FPCA Functional Programming and Computer Architecture (ACM SIGPLAN)

ICECCS International Conference on Engineering of Complex Computer Systems (IEEE)

ICTAC International Colloquium on Theoretical Aspects of Computing (UNU-IIST)

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

ISMM International Symposium on Memory Management (ACM SIGPLAN)

ISO International Organization for Standardization

LCS Laboratory for Computer Science (MIT)

MFPS Mathematical Foundations of Programming Semantics

MIT Massachusetts Institute of Technology

PEPM Partial Evaluation and Program Manipulation (ACM SIGPLAN)

PLDI Programming Language Design and Implementation (ACM SIGPLAN)

PLOS Programming Languages and Operating Systems (ACM SIGOPS)

PLPV Programming Languages meet Program Verification

POPL Principles of Programming Languages (ACM SIGPLAN-SIGACT)

SFCS Symposium on Foundations of Computer Science (IEEE)

SIGACT Special Interest Group on Algorithms and Computation Theory (ACM)

SIGARCH Special Interest Group on Computer Architecture (ACM)

SIGOPS Special Interest Group on Operating Systems (ACM)

SIGPLAN Special Interest Group on Programming Languages (ACM)

SIGSOFT Special Interest Group on Software Engineering (ACM)

SPAA Symposium on Parallel Algorithms and Architectures (ACM)

STOC Symposium on Theory of Computing (ACM)

UNU-IIST International Institute for Software Technology of the United Nations University

USENIX The Advanced Computing Systems Association (formerly: Unix Users Group)

Bibliography

[1] GNU C library. `http://sourceware.org/git/?p=glibc.git`.

[2] Boost C++ libraries smart pointers. `http://www.boost.org/libs/smart_ptr`, March 2009.

[3] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1-2):3–57, 1993.

[4] G. M. Adel'son-Vel'skij and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk USSR*, 16(2):263–266, 1962. also available in translation as Soviet Mathematics: Doklady, published by American Mathematical Society, ISBN 0197-6788.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[6] Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 2010 international symposium on Memory management*, ISMM '10, pages 21–30, New York, NY, USA, 2010. ACM.

[7] Andrew W. Appel. *Modern compiler implementation in ML: basic techniques*. Cambridge University Press, New York, NY, USA, 1997.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[9] Henry G. Baker. Lively linear lisp: "look ma, no garbage!". *SIGPLAN Notices*, 27:89–98, August 1992.

[10] Henry G. Baker. Linear logic and permutation stacks—the forth shall be first. *SIGARCH Computer Architecture News*, 22:34–43, March 1994.

[11] Henry G. Baker. A "linear logic" quicksort. *SIGPLAN Notices*, 29:13–18, February 1994.

[12] Henry G. Baker. "use-once" variables and linear objects: storage management, reflection and multi-threading. *SIGPLAN Notices*, 30:45–52, January 1995.

[13] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21:280–294, April 1978.

[14] J. E. Barnes and P. Hut. Error analysis of a tree code. *Astrophysical Journal Supplement Series*, 70:389–417, June 1989.

[15] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Notices*, 35:117–128, November 2000.

[16] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30:207–216, August 1995.

[18] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 362–371, New York, NY, USA, 1993. ACM.

[19] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[20] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[21] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[22] James Burns and Nancy A. Lynch. Mutual exclusion using invisible reads and writes. In *In Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.

[23] James E. Burns. Symmetry in systems of asynchronous processes. In *Foundations of Computer Science, 1981. SFCS '81. 22nd Annual Symposium on*, pages 169–174, oct. 1981.

[24] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13:677–678, November 1970.

[25] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. *SIGPLAN Lisp Pointers*, V:139–150, January 1992.

[26] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(02):195–244, 1996.

[27] Marshall Cline. C++ FAQ. `http://www.parashift.com/c++-faq/`, July 2004.

[28] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5:532–553, October 1983.

[29] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3:655–657, December 1960.

[30] Gregory Colvin. Exception safe smart pointers. Technical Report 94-168/N0555, C++ committee document, July 1994.

[31] R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 507–516, june 2005.

[32] James O. Coplien. Curiously recurring template patterns. In *C++ gems*, pages 135–144. SIGS Publications, Inc., New York, NY, USA, 1996.

[33] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[34] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 262–275, New York, NY, USA, 1999. ACM.

[35] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[36] John S. Danaher. The JCilk-1 runtime system. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, June 2005.

[37] John S. Danaher, I.-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in jcilk. *Science of Computer Programming*, 63:147–171, December 2006.

[38] Matthew Danish and Hongwei Xi. Operating system development with ats: work in progress. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 9–14, New York, NY, USA, 2010. ACM.

[39] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Pro-*

gramming language design and implementation, PLDI '01, pages 59–69, New York, NY, USA, 2001. ACM.

[40] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11:323–333, May 1968.

[41] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2:153–189, September 1970.

[42] Peter J. Denning. The locality principle. *Communications of the ACM*, 48:19–24, July 2005.

[43] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[44] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569–, September 1965.

[45] Edsger Dijkstra, Leslie Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly darbage collection: an exercise in cooeration. In Friedrich Bauer, E. Dijkstra, A. Ershov, M. Griffiths, C. Hoare, W. Wulf, and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 43–56. Springer Berlin / Heidelberg, 1976. 10.1007/3-540-07994-7_48.

[46] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21:966–975, November 1978.

[47] Ulrich Drepper. ELF handling for thread-local storage. `http://www.akkadia.org/drepper/tls.pdf`, December 2005.

[48] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. `http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf`, 2006.

[49] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 177–190, New York, NY, USA, 2006. ACM.

[50] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. *SIGPLAN Notices*, 37:13–24, May 2002.

[51] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12:611–612, November 1969.

[52] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Notices*, 33:212–223, May 1998.

[53] Sanjay Ghemawat and Paul Menage. TCMalloc : Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[54] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, pages 7:1–7:5, New York, NY, USA, 2011. ACM.

[55] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[56] Jean-Yves Girard. Linear logic: its syntax and semantics. In *Proceedings of the workshop on Advances in linear logic*, pages 1–42, New York, NY, USA, 1995. Cambridge University Press.

[57] James Gosling and Henry McGilton. The Java language environment: White paper. `http://java.sun.com/docs/white/langenv/`, May 1996.

[58] Michael Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, Department of Computer Science, August 1999. Technical report CS-TR-99-1624. Also available as `http://infolab.stanford.edu/pub/cstr/reports/cs/tr/99/1624/CS-TR-99-1624.pdf`.

[59] Dirk Grunwald and Benjamin Zorn. Customalloc: Efficient synthesized memory allocators. *Software: Practice and Experience*, 23(8):851–869, 1993.

[60] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:8–21, 1978.

[61] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994.

[62] Timothy Harris. A pragmatic implementation of non-blocking linked-lists. In Jennifer Welch, editor, *Distributed Computing*, volume 2180 of *Lecture*

Notes in Computer Science, pages 300–314. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45414-4_21.

[63] Timothy Harris, Keir Fraser, and Ian Pratt. A practical multi-word compare-and-swap operation. In Dahlia Malkhi, editor, *Distributed Computing*, volume 2508 of *Lecture Notes in Computer Science*, pages 265–279. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36108-1_18.

[64] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[65] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18:189–207, 2006. 10.1007/s00446-005-0144-5.

[66] E.C. Herrmann and P.A. Wilsey. Threaded dynamic memory management in many-core processors. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 931–936, feb. 2010.

[67] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Notices*, 40:313–326, October 2005.

[68] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 143–153, New York, NY, USA, 2005. ACM.

[69] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. ACM.

[70] Sören Holmström. A linear functional language. In Thomas Johnsson, Simon L Peyton Jones, and Kent Karlsson, editors, *Workshop on Implementation of Lazy Functional Languages*, Göteborg, Sweden, 1988. Chalmers tekniska högskola, Programming Methodology Group.

[71] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, September 1998. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`.

[72] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

[73] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1996. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-701.

[74] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook : the art of automatic memory management*. Chapman & Hall, London, 2011.

[75] Richard Jones and Rafael Lins. *Garbage collection : algorithms for automatic dynamic memory management*. Wiley, Chichester [u.a.], 1999.

[76] Poul-Henning Kamp. Malloc(3) revisited. `http://phk.freebsd.dk/pubs/malloc.pdf`, January 2004.

[77] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[78] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[79] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-645.

[80] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59(1-2):157 – 180, 1988.

[81] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[82] Rick Lavoie and Hongwei Xi. Garbage collector for ATS. `https://ats-lang.svn.sourceforge.net/svnroot/ats-lang/trunk/ccomp/runtime/GCATS0/`, July 2007.

[83] Rick Lavoie and Hongwei Xi. Garbage collection in ATS. `https://ats-lang.svn.sourceforge.net/svnroot/ats-lang/trunk/ccomp/runtime/GCATS1/`, June 2008.

[84] I-Ting Angelina Lee. The JCilk multithreaded language. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, August 2005.

[85] Charles Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51:244–257, 2010. 10.1007/s11227-010-0405-3.

[86] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the connection machine cm-5 (extended abstract). In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, SPAA '92, pages 272–285, New York, NY, USA, 1992. ACM.

[87] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26:419–429, June 1983.

[88] Likai Liu. Mac OS X thread local storage. `http://lifecs.likai.org/2010/05/mac-os-x-thread-local-storage.html`, May 2010.

[89] R. A. MacKinnon. Advanced function extended with tightly-coupled multiprocessing. *IBM Systems Journal*, 13(1):32–59, 1974.

[90] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Electronic Notes in Theoretical Computer Science*, 1:370–392, 1995.

[91] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer Berlin Heidelberg, 2006.

[92] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, April 1960.

[93] Marvin Minsky. A lisp garbage collector algorithm using serial secondary storage. Technical report, Cambridge, MA, USA, 1963.

[94] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[95] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[96] Oracle Inc. Java SE 6 HotSpot™ virtual machine garbage collection tuning. `http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html`. Accessed on December 28, 2011.

[97] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[98] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[99] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[100] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 84–94, New York, NY, USA, 2006. ACM.

[101] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[102] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[103] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pages 470–475, 2000.

[104] Rui Shi, Dengping Zhu, and Hongwei Xi. A modality for safe resource sharing and code reentrancy. In Ana Cavalcanti, David Deharbe, Marie-Claude Gaudel, and Jim Woodcock, editors, *Theoretical Aspects of Computing – ICTAC 2010*, volume 6255 of *Lecture Notes in Computer Science*, pages 382–396. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-14808-8_26.

[105] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In *Proceedings of the 2010 international symposium on Memory management*, ISMM '10, pages 11–20, New York, NY, USA, 2010. ACM.

[106] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, July 1985.

[107] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering*, volume 2177 of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44815-2_12.

[108] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-46425-5_24.

[109] Guy L Steele and Gerald J Sussman. Lambda: The ultimate imperative. Technical report, Cambridge, MA, USA, 1976.

[110] Bjarne Stroustrup. Exception safety: Concepts and techniques. In Alexander Romanovsky, Christophe Dony, Jørgen Knudsen, and Anand Tripathi, editors, *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in*

Computer Science, pages 60–76. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45407-1_4.

[111] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

[112] P. Stygar. LISP 2 garbage collector specification. Technical Report TM-3417/500/00, System Development Corporation, Santa Monica, California, April 1967.

[113] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 254–264, New York, NY, USA, 2012. ACM.

[114] Sun Microsystems. Memory management in the Java HotSpot virtual machine. `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`, April 2006.

[115] Gerald J Sussman and Guy L Steele. SCHEME: an interpreter for extended lambda calculus. Technical report, Cambridge, MA, USA, 1975.

[116] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17:245–265, 2004. 10.1023/B:LISP.0000029446.78563.a4.

[117] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

[118] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. *SIGPLAN Notices*, 42(1):97–108, January 2007.

[119] Harvey Tuch, Gerwin Klein, and Michael Norrish. Verification of the l4 kernel memory allocator. `http://www.ertos.nicta.com.au/research/l4.verified/document-recent.pdf`, August 2008.

[120] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1-2):231–248, 1999.

[121] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *FPCA*, pages 1–11, 1995.

[122] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT Software Engineering Notes*, 9:157–167, April 1984.

[123] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14:1–27, January 1992.

[124] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*, Amsterdam, 1990. North Holland.

[125] Philip Wadler. Is there a use for linear logic? In *PEPM*, pages 255–273, 1991.

[126] Philip Wadler. There's no substitute for linear logic. `http://homepages.inf.ed.ac.uk/wadler/topics/linear-logic.html#linearsub`, 1992.

[127] Philip Wadler. A syntax for linear logic. In Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *MFPS*, volume 802 of *Lecture Notes in Computer Science*, pages 513–529. Springer, 1993.

[128] David Walker and Greg Morrisett. Alias types for recursive data structures. In Robert Harper, editor, *Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45332-6_7.

[129] David Walker and Kevin Watkins. On regions and linear types (extended abstract). *SIGPLAN Notices*, 36:181–192, October 2001.

[130] Paul Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0017182.

[131] Paul Wilson, Mark Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baler, editor, *Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60368-9_19.

[132] Hongwei Xi. A garbage collector for multithreaded programming in ATS. `https://ats-lang.svn.sourceforge.net/svnroot/ats-lang/trunk/ccomp/runtime/GCATS2/`, October 2009.

[133] Hongwei Xi. An implementation of functional maps based on splay trees. `https://ats-lang.svn.sourceforge.net/svnroot/ats-lang/contrib/linmap_splaytree_ngc/linmap_splaytree_ngc.dats`, August 2009.

[134] Hongwei Xi. Introduction to programming in ATS. `http://www.ats-lang.org/DOCUMENT/INTPROGINATS/PDF/main.pdf`, 2010–20??

[135] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In Manuel Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*, pages 83–97. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-30557-6_8.

Vita

Likai Liu was born in Taipei, Taiwan in 1982. During his undergraduate years at Boston University from 2000 through 2004, he became involved in the programming language research group in the Department of Computer Science. He first studied under Professor Hongwei Xi in the summer of 2002, contributing to Xanadu, an imperative programming language featuring Dependent Types. He studied under Professor Assaf Kfoury in the summer of 2003, working on System I^{ω}, which is a type inference algorithm for Lambda Calculus using expansion variables and intersection types with special ω type assigned to unused program variables. Around this time, intrigued by the theoretical remification of unused program resources, he began studying Linear Logic.

In his first graduate school years at Boston University from 2004 through 2008, he worked under the iBench initiative, supervised by Professor Assaf Kfoury and Professor Azer Bestavros, and supported by NSF grants ITR ANI-0205294, ANI-0095988, ANI-9986397, and EIA-0202067. He developed the type system and type inference for Traffic, a declarative language for specifying global flows of network systems.

Since 2008, Likai began to study parallel programming on shared memory computers, lock-free wait-free algorithms, and memory management. This dissertation is a culmination of his work since 2008.

He also writes a blog titled Life of a Computer Scientist (`http://lifecs.likai.org`), where he posts a garden variety mix of ideas, micro research results, commentary on other people's research work, as well as notes on using software and configuring computer systems.