Defining Scalable High Performance Programming with DEF

by

William Mitchell Leiserson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

Signature redacted
Author ....
Department of Electrical Engineering and Computer Science
September 11, 2019

Signature redacted
Certified by ....
Nir Shavit
Professor of Computer Science
Thesis Supervisor

Signature redacted
Accepted by ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Defining Scalable High Performance Programming with DEF

by

William Mitchell Leiserson

Submitted to the Department of Electrical Engineering and Computer Science
on September 11, 2019, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Performance engineering is performed in languages that are close to the machine, especially C and C++, but these languages have little native support for concurrency. We're deep into the multicore era of computer hardware, however, meaning that scalability is dependent upon concurrent data structures. Contrast this with modern systems languages, like Go, that provide support for concurrency but incur invisible, sometimes unavoidable, overheads on basic operations. Many applications, particularly in scientific computing, require something in between. In this thesis, I present DEF, a language that's close to the machine for the sake of performance engineering, but which also has features that provide support for concurrency. These features are designed with costs that don't impede code that doesn't use them, and preserve the flexibility enjoyed by C programmers in organizing memory layout and operations. DEF occupies the excluded middle between the two categories of languages and is suitable for high performance, scalable applications.

Thesis Supervisor: Nir Shavit
Title: Professor of Computer Science

Acknowledgments

This thesis owes a great deal to my advisor, Nir Shavit, who's ability to frame problems and craft narratives is unparalleled (so to speak). Nir has been extremely encouraging, and I'm grateful for his guidance. I also want to thank my other committee members, Saman Amarasinghe and Maurice Herlihy, whose breadth of knowledge and careful attention to detail made this thesis a stronger (and better sourced) document. And, of course, I wouldn't have been in the Ph.D. program, let alone have completed it, if not for the support and encouragement of my wife, Erin. She has consistently motivated me through the easy times and the hard. Last, as a proud Papa, thanks to Walter, Freddy, and William, whose smiles and hugs gave me endless joy as I worked.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Performance Engineering versus Scalability

Inasmuch as a new class of *systems languages* that use garbage collection (GC) have emerged at the forefront of high performance programming, the C and C++ programming languages, which lack GC, maintain a position of prominence. Part of this, naturally, is due to the sheer volume of legacy code. Applications and operating systems are implemented in C, and rewriting them in a higher level language may be impractical, undesirable, or even impossible. But new applications continue to be written in these languages. Of the popular languages, today, C and C++ are unique because they permit *performance engineering*, the ability to leverage knowledge of the specific hardware to optimize source code. Performance engineering is a practice that's only available to programmers of languages that are *close to the machine*, with data types and operations that intuitively and directly correspond to hardware primitives.[81]

A language that's close to the machine doesn't merely run fast in the common case – it provides the programmer with the tools necessary to organize data layout in memory, taking into account alignment, packing, and location on the stack or on the heap. It also doesn't insert hidden operations such as array bounds-checking, pointer liveness checking, or checking a divisor for zero. In higher level languages all of these features are provided for the safety of the programmer, but a language that's close to the machine eschews them in order to provide the programmer with the tools necessary to maximize performance.

Paradoxically, however, it's runtime support and the abstraction of a language that contribute most significantly to scalability through concurrency. Two broad categories of concurrency are relevant: concurrent data structures and task distribution. As regards concurrent data structures, when multiple threads traverse a concurrent object, if one of them removes a node it must regard the other threads as *invisible readers* that may be holding references to the node it's trying to remove. In the presence of a garbage collector, every reachable object is tracked and the thread may simply drop its reference to the removed node. Not so in the absence of a garbage collector. Merely dropping the reference leaks memory and would eventually cause the program to crash with enough use. However, garbage collection represents an unacceptable performance cost for certain applications, and it's intentionally disincluded from languages like C, C++, and Rust. The naïve solution in one of these languages is to lock the data structure, but this is slow because it causes threads to queue up on the lock, and even reader/writer locks introduce contention on read-heavy workloads.

To the second point, task distribution is a commonly achieved through language or library extensions like Cilk, OpenMP, TBB, and others.[10, 13, 78] Even here, however, there is a scalability problem in the way of contention created by communication between threads. Concurrent memory reclamation hints at this problem, as solutions for low level languages largely run into this space. But the degree of contention in a shared object is typically the make-or-break property in any practical test; memory reclamation notwithstanding. How ever well the load balancer distributes tasks, if they thrash in the cache, their "communication" is a performance bottleneck. Modern commercial CPUs include hardware transactional memory, which is an exceedingly low-contention feature where updates to commonly read memory are rare. A language may even employ hybrid transactional memory to supplement hardware transactions when the latter don't provide a forward progress guarantee. This includes invisible counters, branches, and instructions (in the slow path). But, whereas a high level language may include synchronization abstractions that are (or could be) implemented using transactions, C and C++ support nothing more than intrinsics for the hardware. Roll your own hybrid transactions. Every time.

To sum up: the problem for a low level language is two-fold: memory reclamation and contention, and the high level language can easily address both. Memory reclamation is achieved through GC, and contention can often be addressed with transactions. A programmer who's willing to accept the cost of these features has a general solution to concurrency, but these costs are antithetical to the design philosophies of contemporary close to the machine languages.

It is the space in which performance engineering and concurrency meet that I'm primarily concerned. Languages exist on a spectrum between safety and ease of use (high level), and closeness to the machine (low level). The distinctions are quantized in that the difference between the levels comes down to particular features that appear in runtimes or in the compiled code. But in practice there's an excluded middle. Even the aforementioned modern systems languages tend not to provide programmers with fine-grained control over, e.g., memory layout, instead making a best effort to be fast in the common case. They are highly successful, as evidenced by the widespread adoption of languages like Go, but applications in which performance engineering is required could never make the transition.

Contrast this with concurrent data structures implemented in C or C++: the mere problem of invisible readers has generated an enormous amount of research with numerous memory reclamation techniques. Even the literature presenting the concurrent data structures, themselves, tends to paper over the memory reclamation question. Implementing such a structure in C or C++ directly from a published research paper is non-trivial for this reason, alone. Not that it isn't done – many applications use concurrent data structures – merely, the languages provide no native support for it. Every implementation reinvents the wheel of memory reclamation, and programming hybrid transactions to deal with contention is a perilous undertaking, not only difficult to get right, but it's also difficult to debug.

The problem of scalability in low level languages compounds as new features, like hardware transactional memory, begin to appear in commercial CPUs. Again, reimplementing hybrid transactions for each use is a fraught exercise. And as new and faster techniques for software transactions are developed, source code requires reimplementation rather than a simple recompile. It's worth observing, here, that the code transformations for hybrid transactions are fairly rote; merely, it's easy to make a mistake when done by hand, and acts as the quintessential example of duplicated code (for the two or more paths) that make such code difficult to maintain at a correctness level.

1.2 Performance Engineering *and* Scalability

C and C++ aren't bad languages for making scalability difficult any more than Go is a bad language for making performance engineering impossible. These are design decisions. The languages are well suited for the purposes for which they were designed. But none of these were designed for implementing performance engineered concurrent data structures.

DEF, introduced in this thesis, is a new programming language with high level abstractions for concurrency-related features, but it also allows programmers to performance engineer their code. Runtimes and abstractions impose unavoidable overheads only insofar as they're explicitly used by applications, and don't impose hidden costs or impede a programmer's ability to performance engineer their code. The goal of DEF is to provide high level support for concurrency, particularly in the implementation of concurrent data structures, in applications that are otherwise geared towards maximum performance on each CPU core.

DEF provides native support for *on demand automatic memory reclamation* for concurrent data structures by way of the `retire` keyword that stands in where `free` would in a serial application. This is an approach to concurrent memory reclamation that provides some of the benefits of GC without requiring an entire application to pay the cost of generalized GC. Moreover, the cost of the runtime handing retired memory is borne by an application only to the degree it's actually used by the application's concurrent data structures.

Atomic operations are also natively supported, including atomic blocks of code implemented as hybrid transactions. As in any low level language, primitives are available, and nothing prevents a user from specifying specific Read-Modify-Write (RMW) instructions through intrinsics or manually implementing hybrid transactions using hardware transaction intrinsics. But the syntax is provided for these atomics in a way that keeps the code simple, and hybrid transactions as correct as the hardware fast path written by the programmer.

This thesis includes sample source code and micro-benchmark results for various concurrent data structures implemented in DEF as compared with C counterparts. As such, some space is also devoted to the DEF macro language, Interpreted Structural Macros (ISM), which was crucial for implementing the benchmarks and duplicating code with minor variations in ways that are difficult to accomplish in C or C++. DEFISMs are Lisp-like and permit various DEF constructs to be treated as data – not merely text to be substituted. They also operate at many levels of the parser, not just at the function or data structure level as C++ templates do.

Last, acknowledgment must to be given to the fact that a tremendous amount of legacy code is already written in C and C++, and it's unreasonable to expect it to be rewritten. DEF, therefore, is designed with the goal of transparent C integration, such that a module written in DEF can be dropped seemlessly into a C application. C types and functions defined in header files are available to DEF modules, and DEF types and functions can be made available to C modules as easily as they are to other DEF modules. The same utility, `defghi`, that generates DEF interface files (`.defi`) from DEF source files (`.def`) can generate C header files. This was helpful not only in the implementation of the micro-benchmarks, but also acts as a principled approach to language integration in general, as most languages work hard to integrate with C. DEF is available, not merely as a tool for writing fast concurrent data structures for itself, nor even just C and C++ as well, but for any language that requires performance engineered concurrent data structures.

1.3 Research Contributions in This Thesis

Ample acknowledgement is due to my coauthors, and certainly to Nir Shavit, my advisor, in the undertaking of the work presented in this document. But the following is an itemized account of my specific work and the contribution it makes to the field. To the best of my knowledge, these contributions are new or meaningful improvements over existing work.

- **ThreadScan**: I wrote the ThreadScan library, in its entirety, and benchmarked it. ThreadScan is the first automatic/semi-automatic approach to memory reclamation for C, apart from general-purpose garbage collection. As with a garbage collector, it imposes no burden on data structure traversals, no additional per-node memory requirements in the way that, e.g., reference counting does, and (because of its limited scope) performs at the rate of a "leaky" implementation. Versus reference counters or hazard pointers, which burden traversals, ThreadScan is light-weight and low contention for read-heavy workloads. Epoch-based mechanisms, which impose strict rules on pointer validity, don't allow the pointer freedom ThreadScan provides. As against any of these, ThreadScan is automatic or semi-automatic (depending on how it's used).

- **Forkscan**: Like ThreadScan, I did the complete implementation of Forkscan and benchmarked it. Forkscan is an incremental improvement on ThreadScan in that it maintains the theoretical *lock free* property of a data structure that uses it.[1] Moreover, it solves the general problem of references that escape the stack. Forkscan provides the same guarantees as a conservative garbage collector, but unlike the latter, tracks only objects flagged by the programmer. Again, the limited set of objects being tracked creates a multitude of opportunities for optimizations unavailable to general purpose garbage collectors. Besides ThreadScan and Forkscan, I am not aware of any prior work with this set of properties.

[1]Caveat: From a systems perspective, it's important to note that Forkscan contains a lock, but Forkscan's lock is not practically different from an allocator's lock – a lock that is generally accepted as not infringing on the theoretical lock freedom of a data structure.

- **The DEF Programming Language**: I designed and implemented the DEF language and compiler, as well as the `defghi` utility. I also implemented some of the concurrent data structures used in the benchmarks. DEF is the only language, of which I am aware, that exists at the intersection of languages that are close to the machine (and, therefore, enable performance engineering of code), and languages with high level native support for the implementation of concurrent data structures. Existing in this intersection means integration of concurrent memory reclamation through a *retire* interface into a close to the machine language, a technical barrier, and the inclusion of native atomic blocks into the same, a philosophical barrier. Since these features are inherently high level, C has neither of them and, as a matter of its design goals, never will. The Go language, designed for high performance concurrent programming, is not close to the machine and therefore does not permit performance engineering, by design. It also has a garbage collector, which is fast but unacceptable for close to the machine programmers. Rust, the other language typically cited in these contexts, is designed for safe concurrent computation with no overhead, but is not designed for implementing concurrent data structures. Its safety features, on the contrary, prohibit sharing pointers to objects among threads.

1.4 Structure of This Thesis

The chapter on Background is provided as justification of the need for a language in this space, including relative performance characteristics of C and Go. Go is treated as the architypal modern systems language designed for performance, since it advertises comparable speed to C in similar applications, but which trades safety for flexibility in performance engineering. Additionally, since a good deal of the work that went into the development of DEF was bound up in addressing the question of concurrent memory reclamation, space is allotted to explanations of conventional approaches to this problem.

After that, in the Retire chapter, the semantics of on demand automatic memory reclamation are laid out, and my work on ThreadScan and Forkscan are described. As published, neither work quite fits the required semantics for the `retire` abstraction, but Forkscan was adapted to implement that feature in DEF, so an explanation of the changes that were made (as well as some optimizations to the original implementation) is necessary.

The language, itself, is described in the DEF Overview chapter. This includes syntax, the relationship between operations, data, and hardware, and the high level features that facilitate the development of concurrent data structures. As mentioned above, it also lays out ISM and compares it to other macro languages. Since the fundamental contribution of DEF is the synthesis of high level abstractions into a language that's close to the machine, special attention is given to how they interact so as not to impose invisible (let alone unavoidable) overheads. One is particularly interested in the principle of least surprise regarding performance characteristics of a DEF application.

Examples and empirical microbenchmark results are presented in the Practical DEF chapter. DEF is designed to be as readable as, e.g., the concurrent data structures expressed in the literature, and also to provide syntactic structures that could be ported back into the literature, itself. A couple of specific data structures are developed and benchmarked, and a wider suite of benchmark results are provided to demonstrate DEF's performance and ease of development.

Last, although the language is already sufficiently developed for common use in many applications, there's more work to be done – both in the design of the language and in the implementation of the compiler. DEF takes a step out into a space that isn't well-explored, at the intersection of low level and scalable groups of languages, and opportunities abound for advancement. The Conclusions chapter describes some of these opportunities and challenges, and summarizes what I believe are the core contributions DEF makes to the field.

Chapter 2

Background

Tension exists between a language's ability to do low level manipulations of memory, and the presence of high level abstractions. To appropriate a Twain-ism, "reports of C's death have been greatly exaggerated." C hangs on, not just because of legacy code, but because of its ability to operate in a space ignored by high level languages, even systems languages. Yet it's missing crucial support for multicore programming the latter provide.

I argue this excluded middle isn't inherent, but speculate that the philosophies have generally kept them apart; low level languages don't include high level abstractions for anything because the abstractions don't give programmers tight control over sequences of instructions and memory layout. But if the low level primitives exist that give programmers control, there's no contradiction in providing the abstractions, too, unless they infringe on the former by design. Paradoxically, actually engineering scalable code is typically better handled by abstractions than individual programmers (the case for this, below). DEF is designed to do both, and there have been (and remain) clear hurdles to this kind of integration.

This chapter is primarily about the state of these two dimensions, and why both are relevant to modern software development. I want to tell a story, here, about why the state of programming languages is what it is today, why there's an excluded middle between these two philosophies, and why anybody cares about that middle. The answer to these questions provides motivation for thinking about a new kind of

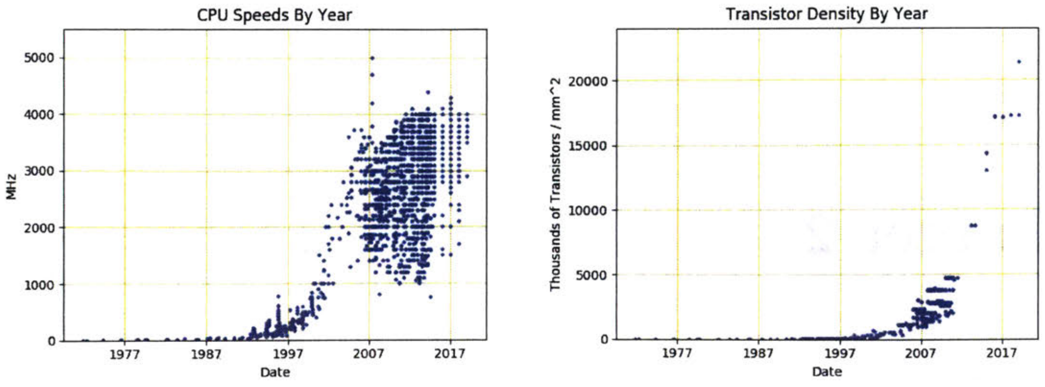


Figure 2-1: The error in conflating Moore's Law with its popular formulation. Left: Intel and AMD CPU clock speed over time. Right: Transistor density over time.

programming language – not one that tries to be everything for everybody, but merely tries to fit the excluded niche in which people care about performance engineering *and* scalability for their code. The story begins with the traditional way in which programs attained performance: "free" (from the perspective of the programmer) increases in hardware clock speed, and non-free tailoring of code to specific architectures.

2.1 The Case For Performance Engineering

For decades, *Moore's Law* was popularly understood as the idea that the clock speed of CPU's doubled every year or two. What Moore actually stated was that the number of components per integrated circuit doubled every year.[74] The erroneous conflation of the popular notion with the actual statement became apparent as the rate of increase in CPU clock speeds declined, even as transistor density maintained its trajectory (fig. 2-1; although even Moore's actual Law appears to be flagging).[1] Increasing clock speed along with reduction in transistor size led to material limitations in the silicon. In short, the silicon in which transistors are laid out has a maximum energy density before it melts.

Note, here, that clock rate is not equivalent to *performance*, even though the one could safely be used as a stand-in for the other in days of yore. Increased tran-

[1]Data up until 2014 acquired from [33]. Subsequent data (2015 and later) was gathered from [27] and the Intel website. Transistor density is estimated based on limited released data from Intel.

sistor density continued to allow for performance improvement – just not in the "free lunch" sense that software developers had come to rely on. As clock speed leveled off, developers and compiler writers, alike, had to take into consideration new and ever-increasing-length vector instructions with Single-Instruction-Multiple-Destination (SIMD) semantics. Memory hierarchies, a perennial double-edged sword in performance, similarly enjoyed more widespread focus. Knowing something about the size of a cache line, or the behavior of a branch predictor can make or break the performance of an application. Before even getting into discussion of the multicore explosion, there is an *increasing* potential to exploit architectural subtleties even on a single core.

This is the grounds for taking performance engineering seriously as a practice in the modern era. Inasmuch as modern systems languages are optimized for the common case to be competitive with low level languages, C and C++ provide a degree of control over the instructions that will be executed and the layout of memory that these languages don't. C, designed in 1972, was tasked with implementing the AT&T Unix kernel.[81] Close to the machine, as a design goal, was necessary because an OS has to be fast, and hiding implementation details presents a hazard to that goal. C++, in the early 1980's, was designed to fit the same niche, but also provide greater flexibility for application development by adding classes and other features.[87] As with C, C++ provides maximal control over instruction sequences and memory layout for end programmers, but avoids any native abstractions for which programmers don't have this kind of control.

By contrast, modern systems languages like Go, hide implementation details for common operations. In Go the location of a newly allocated object, whether on the stack or heap, is undefined.[43] As an implementation, Go will allocate an object on the stack unless it can't prove that no references escape. This balances performance with abstraction in a way that's useful to some kinds of applications. But choosing where to allocate memory is an incredibly powerful tool for handling performance: an MCS lock implementation, for example, might allocate a node on a thread's stack even though other threads *will definitely access it.*[69] Allocating it on the stack means

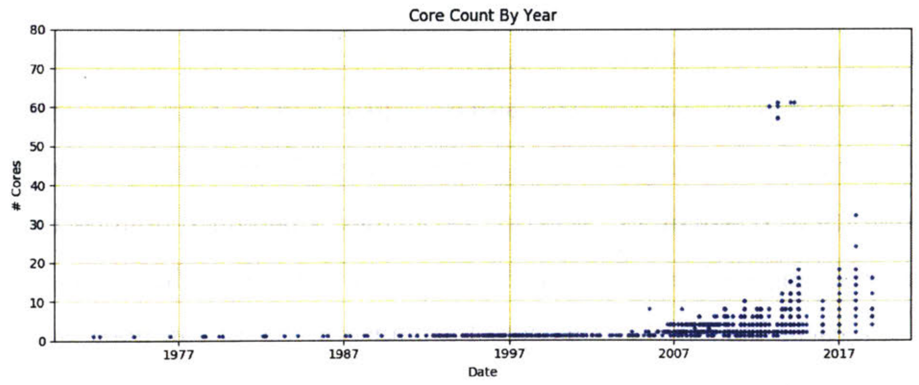


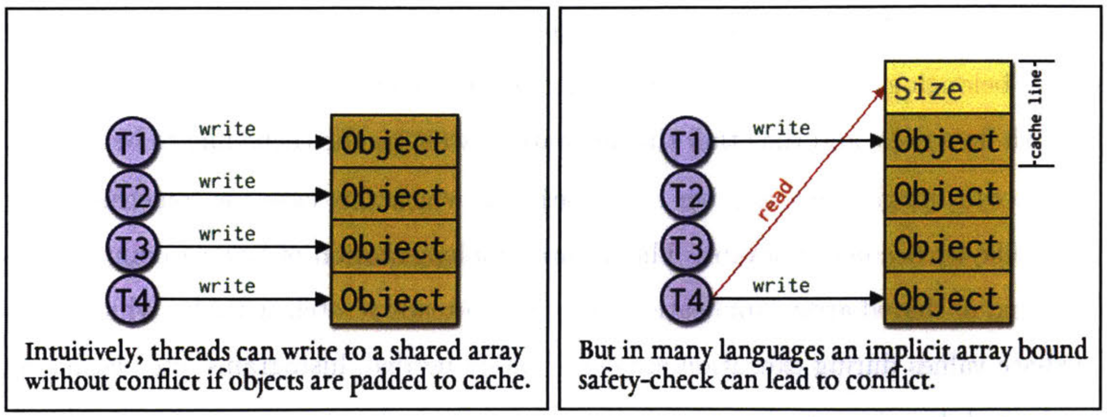
Figure 2-2: Rising core count on high-end commercial CPUs by year.

that cleanup is as simple as popping the stack frame. Go doesn't define where memory is allocated, or let programmers perform explicit pointer arithmetic, or any number of other things that are common in high performance C code.

C and C++ offer a degree of control over the generated assembly code that is missing from modern languages. Abstractions are the conventional approach to language design, leading to hidden costs that are neither avoidable, nor consistent across versions of the compiler. Clearly, low level languages not only have life left in them, but aren't going to go anywhere unless and until they're displaced by another low level language. But this story gets more interesting when multicore systems come into play, as they do since the aforementioned "free lunch" of hardware speedup has diminished.

2.2 The Multicore Revolution

Go is successful, not in spite of the abstraction it provides to programmers, but because of it. Abstract representation of common concepts lowers the bar for implementing big programs, and reduces the complexity of the code, reducing bugs. The obvious question is how one tunes the performance of a program that doesn't give the programmer access to the nitty-gritty details? But this question is obviated by the fact that the real increase in performance potential brought about by Moore's Law (the real one, not the popular one) is multicore CPUs (fig. 2-2). Scalability is



Intuitively, threads can write to a shared array without conflict if objects are padded to cache.

But in many languages an implicit array bound safety-check can lead to conflict.

Figure 2-3: The hidden cost of array bounds-checking can impede speedup unexpectedly. Here, multiple threads access their own objects in a shared array, but the bounds-checking causes cache thrashing if the array dimension is stored on the same cache line as the first object.

the name of the game in modern performance brought about, not by making individual cores faster, but by increasing the number of shared memory cores in a single computer. Of course, such systems are harder to program.

Scalability is a non-trivial problem, but well-studied, and language abstractions that facilitate scalable code are in demand. There are two ways of thinking about scalability that each warrant discussion: *parallelism* and *concurrency*. Parallelism, for the purposes of this thesis, is concerned with the maximum speedup over sequential execution that can be achieved, no matter how many cores, floating point units, or other hardware features are provided. Concurrency is a property of a program: it's the interaction of independent operations occurring simultaneously, particularly as many operations are performed on common objects by different threads.

Both of these paradigms have important implications for performance engineering, especially since the presence of other abstractions representing, e.g., safety features, can require performance trade-offs that might inhibit scalability. Consider the example of a program in which many threads are accessing a shared array, as in fig. 2-3. In this example, threads write to their own objects in the array and only look at others on rare occasions. Intuitively, provided each object has its own cache line, performance should scale with the number of cores on the system. However, a language that does array bounds checking may cause cache thrashing by placing the size of the array – a value each thread must read on every access – on the same line as the object belonging to Thread 1. As threads read the array length, they force Thread 1 to flush its writes, and the other threads have to wait for the cache line to be fetched.

We can imagine any number of scenarios in which this case (or something very much like it) becomes relevant. The natural implementation of hazard pointers uses this kind of shared array, applications may give threads scratch space for which intermediate values during execution are occasionally needed, histograms, etc. Is this an implementation bug on the part of the language? Unclear. It represents a trade-off, since putting the size on a different cache line from any of the objects means increased space consumption, even for serial applications. Can a programmer work around it? Sure. But we must acknowledge that a "work around" is precisely what this is, and that none of this is hidden (let alone subject to change) in a low level language.

We deal with the parallelism and concurrency-related abstractions each in turn to understand how, and to what degree, they impede performance engineering. For DEF's purposes, it's important to think about where unavoidable overheads become visible, particularly if a feature isn't being used. In the shared array example, the programmer may have known that no access would ever occur out of bounds, so the safety feature is unnecessary. Nevertheless, the performance cost to the application is quite high.

2.3 Parallelism

2.3.1 Theory Basics

We are here concerned with the ability to divide an algorithm in such a way as to allow parts to execute in parallel. The most intuitive parallelism is the instruction-level parallelism provided by modern CPUs in the form of vector operations. An instruction can be applied to an array of values, for example, and since none of the resulting values depend on one another there's no need to perform them sequentially. Therefore, hardware exists to process them simultaneously, even on a single core. Modern compilers are so good at exploiting this kind of parallelism, they emit vector operations for programs written in languages that have no special syntax for representing instruction-level parallelism.

More challenging for programmers is the kind of parallelism that allows instructions or blocks of code to execute in parallel on different cores, when data dependencies aren't as obvious. In particular, it isn't always clear how parallelizable an algorithm truly is. The limitation on parallelism was first clearly articulated by Amdahl, who observed that the maximum theoretical speedup was limited by the percent of code that was inherently sequential.[5] Although Amdahl intended this as a critique of parallelization, it's since been interpreted and re-expressed as a mathematical property:

$$\text{Speedup} = \frac{w_s + w_p}{w_s + \frac{w_p}{n}}$$

Where w_s is the part of the computation that's inherently sequential, w_p is the part that's parallel, and n is the number of cores across which the parallel part of the code is divided. This doesn't take into account memory hierarchies or other hardware features that might impact performance, but it's a theoretical upper-bound on what an algorithm can achieve on n cores. Note that as $n \to \infty$, the cost of the parallel part of the program approaches zero, leaving only the serial part. Thus, we can infer a maximum theoretical speedup, or total parallelism based on how much of the code is sequential:

$$\text{Parallelism} = \frac{w_s + w_p}{w_s}$$

Even if much of this serial portion is technically in parallel with everything else, it's still the limiting factor in attaining speedup. The basis for this is well-explored, and in the literature w_s is sometimes called the *critical path* or the *span* of the computation.[46, 36, 28, 11, 60]

Granularity is the path to parallelism; lots of small tasks that don't depend on each other lead to low theoretical barriers to speedup. Practical barriers to speedup are another matter. "Sequential" code, according to Amdahl's Law, is a matter of a *happens before* dependency between tasks. Parallelism, on the other hand, is the non-imposition of that relation. But as a practial matter, code can serialize itself without that explicit dependency.

Mutual exclusion, for example, can limit speedup by forcing threads to wait for each other, even if the tasks they want to perform have no predetermined order – the operations may not even actually conflict, but use of the mutex is too coarse or the operations couldn't have been predetermined not to conflict. The obvious example is memory management: even if knowing when to deallocate memory is a trivial matter that requires no inter-thread communication, a bad allocator might undo all of the careful division of tasks that went into development of the algorithm. More broadly, access to shared resources and the requisite communication (implicit and explicit) can reduce the scalability of an algorithm.

Time spent holding a lock is factored into consideration of the parallelism of an algorithm by treating all such critical sections with a particular lock in common as inherently sequential. Naturally, the less time spent holding locks, the shorter those sequences. But with the advent of commercially available hardware that supports transactions, the possibility of lock elision arises.

Transactional memory makes it possible to maximize the granularity of an algorithm in spite of shared (and updated) resources. Tasks that don't *actually* conflict don't need to be executed sequentially. More complete discussion of this is presented in the section on Concurrency, but it's important to recognize that even insofar as parallelism primitives have been implemented in low level languages, the matter of hybrid transactions remains squarely within the purview of high level languages. Therefore, so does the general realization of maximum granularity.

Compound this with the problem of concurrent memory reclamation, wherein many conventional methods a thread has of alerting its peers that it's looking at a particular node require synchronization. Fig. 2-4 shows removal of a node from a linked list. The removing thread uses a compare-and-set (CAS) RMW operation to swing A's next pointer from B to C, but whether an invisible reader is looking at B is unknown. Therefore, B can't be free'd, as it would be in a serial data structure, until

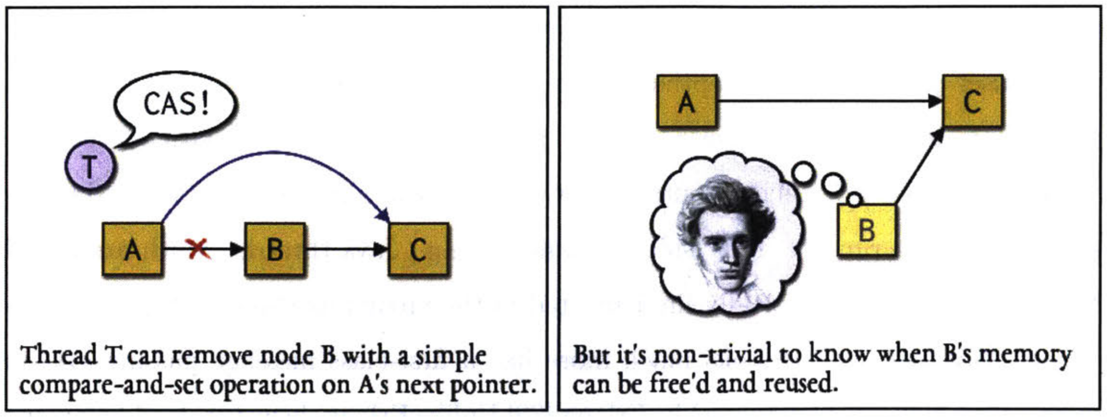

CAS!
T
A
B
C
Thread T can remove node B with a simple compare-and-set operation on A's next pointer.
A
C
B
But it's non-trivial to know when B's memory can be free'd and reused.

Figure 2-4: Node B is removed from a lock free linked list. Left: A's next pointer is swung from B to C. Right: B can't be free'd, so its fate is left to the implementation.

it can be proved that no thread holds a reference to B anymore. This is the problem of *invisible readers*, threads that read but don't write in the global state, and are therefore invisible to other threads. Again, locks are a powerful tool, but they have a tendency to serialize an algorithm.

Atomic operations are covered in sec. 2.4, and concurrent memory reclamation is discussed in greater detail in section 2.5, but a more complete analysis of the state of parallelism is beyond the scope of this thesis. The takeaway is that programmers are concerned with the ability to express parallelism within their code to achieve speedup, and it's crucial that languages provide enabling tools. If parallel units of work are necessarily coarse by virtue of the way they act on data, it doesn't matter how easy it is to express them.

Nevertheless, expressing parallelism is its own can of worms that requires discussion. Actually approaching the theoretical speedup for the given work is often a matter of scheduling the work efficiency, and there are two main models used by programming languages: the threading model, and the fork-join model.

2.3.2 Threaded Parallelism

Many parallel languages and extensions tend to encourage thinking about their parallelism-related features as threads. In general, these features aren't actually as primitive as, e.g., pthreads, Windows threads, or even Java threads, and they provide services that make it easier to distribute work across cores.

Go uses *goroutines* – conceptually cheap threads implemented as mini-stacks that the Go scheduler multiplexes over a thread pool – that communicate with one another through channels.[43] Each goroutine is a work unit, and a mini-stack is created in the process for it to run on. This isn't as coarse as, e.g., Java threads, which are literal heavy-weight threads typically implemented in the virtual machine on top of system threads,[45] and instead looks much more like a first-class message passing system from a very high level language like Erlang.[89] Unlike Erlang, however, load balancing is performed by the runtime, and every goroutine is running in the same address space and has access to global variables. The routines, themselves, are multiplexed across the threads like Windows fibers,[23] except the scheduling is done by Go.

2.3.3 Fork-Join Parallelism

An alternative model is to identify work that can be performed in parallel by conceptually forking some code, and then identifying a join point that defines a *prior-to* relationship between the parallel work and what comes after. Cilk, a language extension to C/C++ (but has also been applied to Java as JCilk[32]), expresses parallelism using `spawn`, which identifies a function that may be executed in parallel with its continuation, and `sync`, which acts as a barrier until all parallel work in the function has completed.[10] Work is distributed by the runtime according to a work-stealing scheduler in which a pool of *workers*, threads that execute Cilk-augmented code, create work and push it onto one end of a deque to be popped off the other end (stolen) by other workers that have run out of work to do.[12]

As originally implemented, Cilk used a cactus stack in which function frames were allocated from the heap, much like the mini-stacks used by goroutines; even more expensive, since mini-stacks are only allocated when a goroutine is spawned, and every function instantiation was heap allocated in Cilk. But with the relative infrequency of steals, subsequent Cilk runtimes were implemented by allocating full stacks at the point of steal, rather than individual frames at the point of spawn or call.[22] This means that spawns are cheap – only a few instructions with no allocations – and the slow path only comes in the infrequent case.

Similarly, OpenMP is a popular extension available in C/C++ (among other languages) that provides a programmer with compiler directives and library functions that allow a programmer to identify regions of code that can be executed in parallel.[13] Traditionally, OpenMP required programmers to identify a block and manually distribute the work among threads using a thread ID – a system somewhat less advanced and convenient than a goroutine. As the system evolved, however, OpenMP added a more dynamic scheduler that allowed programmers to identify blocks that could, but were not required to, execute in parallel (much like Cilk) and may (depending on implementation) even use work stealing.

Threading Building Blocks (TBB) uses this same technique in library form for C++.[78] As a library, no language extension is required and facilities have been created for expressing advanced parallel techniques like pipelining.

The difference between the two conceptual models is important. The former requires programmers to think about threads, no matter how light-weight, if not the actual scheduling of work. The latter merely identifies computations that may execute in parallel. Neither can be called "close to the machine," but the abstractions actually provide the flexibility to expressing parallelism without imposing guaranteed space, cycle, and scheduling overhead versus the threaded model.

The fork-join model (as implemented in Cilk and TBB) is also more rigidly structured, computationally, allowing the existence of tools like provably good race detectors,[38, 59] scalability analyzers that compute work and span, objectively,[50] and parallelism profilers.[82] In terms of parallelism, at any rate, the problem has been addressed in satisfactory ways that are applicable to low level languages (or at least their extensions).

The cost to a program that doesn't need the feature is negligible since even if the runtime is present, a conventional implementation will park the threads in the thread pool until there's work to do – whether in fork-join parallelism or threaded parallelism. The wide availability and use of OpenMP, TBB, and Cilk is a testament to this. At some level, independent of the readability or expressibility of the syntax (which varies from system to system), the problem of parallelism in low level languages has achieved a measure of success. In spite of the volume of work done on concurrency, however, the same cannot be said of it.

2.4 Concurrency

Speedup, in a concurrency-sense, of a program depends on design of the common objects being accessed by threads. Most intuitive methods for synchronization that avoid data races have performance-inhibiting side effects. For example, merely locking an object before operating on it doesn't scale if it's being touched by lots of threads simultaneously because the accesses are serialized. As a consequence, mechanisms have been devised that avoid locks as much as possible, typically in favor of RMW hardware primitives that make changes visible to other threads atomically. This atomicity is crucial, in that it prevents any thread from reading a partial state.

Concurrent data structures are a hot area of research, and concurrent versions of most popular data structures are known.[40, 49, 52, 86, 76, 70, 61, 2] To go into any of them in detail is beyond the scope of this thesis, but it suffices to say that many of them scale linearly with the number of cores in all but the most write-heavy applications. Naturally, since the data structures themselves are often complicated to implement, languages designed for scalability will implement their own concurrency libraries. Go has a synchronized map in its library, and Java's concurrency package is popular with that community and provides numerous data structures and primitives that allow programmers to implement their own.[44, 77]

It's worth noting, however, that there's very little work in the way of concurrency libraries for C or even C++. Although both languages have intrinsics for the hardware RMW operations,[25, 26] the data structures themselves are in short supply. Even Boost, which is typically ahead of the curve in terms of C++ functionality, contains exactly three concurrent data structures (a stack and two queues) in its Lockfree package.[9] This is not to say that concurrent data structures aren't popular in C and C++ applications – they are, as discussed below! Moreover, concurrent data structure benchmarks in C and C++ are popular for testing performance of one versus another.[47, 48] But there's a catch, here: the applications modify the data structures, as presented in the literature, and even though the benchmarks typically don't, they leak memory and will crash if executed for long enough. Extensive research has been performed in resolving this dilemma such that it deserves its own section: Concurrent Memory Reclamation. The point, here, is that conventional close to the machine languages lack some of the basic infrastructure of high level languages, and that makes concurrent data structure implementation difficult to generalize across applications. Such structures tend to be reimplemented, even in C++, from scratch for each use. Whereas high level languages have resolved this problem, low level languages haven't.

Sychronization and Atomics A second barrier to implementation in low level languages is the lack of a symbolic approach to atomicity employed by high level languages. Languages that represent complex concurrency features symbolically, just as in parallelism, avoid reinvention of the wheel on the part of programmers and allow the implementation of those features to improve along with the newest theory.

Java provides the `synchronized` keyword that can be applied to methods or blocks of code (See [45], sections 8.4.3.6 and 14.19). When applied it guarantees atomicity within that method or block against other synchronized operations on the object (either `this` in the case of a method, or a specified object in the block case). As of the writing of this thesis, synchronization is achieved through per-object monitors. Any programmer who uses it needs to be aware of the possibility of deadlock, wherein one thread tries to acquire the monitor locks on two objects, and another thread tries to acquire them in the opposite order. The semantics specify a particular implementation that imposes these constraints.

But that implementation could change and be replaced by, say, transactions, and the only change in semantics would be to release the programmer from the burden of ordering the objects. Such a change would be a simple case of lock elision, and it would improve performance in cases where simple read, or mostly read, operations were allowed to overlap. Moreover, when primitive hardware transactions don't guarantee forward progress, synchronized code could be implemented with hybrid transactions. And, again, as the theory around software transactions advances, so too could the implementation of `synchronized`.

Clearly, it would be contradictory for a low level language not to provide the primitives for implementing either a monitor/mutex or intrinsics for hardware transactions. Nevertheless, implementing hybrid transactions by hand is both tedious and hard to get right, and it makes code difficult to reason about and maintain. Moreover, the presence of such a feature wouldn't adversely impact the performance of any application that didn't explicitly use it, meaning low level languages don't benefit from its absence. On the contrary, requiring programmers to use the primitives for such a feature requires reinventing the wheel for every case, making it a barrier to implementing concurrent data structures in C and C++.

2.5 Concurrent Memory Reclamation

To recap: Concurrent data structures exist in high performance applications and benchmarks written in low level languages... but they're modified from what's presented in the literature in the applications, and typically leak memory in the microbenchmarks.[47, 48] The dilemma of modifying data structures versus leaking memory in low level languages is, at its core, the aforementioned problem of invisible readers. This is a non-issue in high level languages, which clean up their memory through garbage collection (GC). Detecting that no thread holds a reference to an object is just another day at the job for GC, and high level languages (especially high level systems languages) solve this problem trivially.

In low level programming languages, the issue is decidedly non-trivial. The conventional solutions can be classified in the following groups: Conservative GC[34, 72], epoch-based[67, 66], reference counting[35, 42], pointer-based[71, 80, 17], and Rust[64]. Each of these are described in turn, followed by a section on StackTrack[1] which makes a significant step towards a principle that's relevant to the research presented in this thesis. But I'd be remiss not to observe that the proliferation of solutions draws attention to the complexity of the problem. Unsurprisingly, concurrent data structures, as presented in the literature, typically drop pointers to removed objects and leave memory cleanup to the implementation – they avoid the matter altogether. This works well enough for high level languages that use GC, but we'll see that even GC isn't without its pitfalls; points that make it unpopular in low level languages. To speak precisely about this problem, let us establish a few definitions.

Consider a set of n threads, communicating through shared memory via primitive memory access operations: These operations are applied to *shared objects*, where each object occupies a set of (shared) memory locations. A *node* is a set of memory locations that can be viewed as a single logical entity by a thread. A node can be in one of several states [71]:

1. *Allocated.* The node has been allocated, but not yet inserted in the object.
2. *Reachable.* The node is reachable by following valid pointers from shared objects.
3. *Removed.* The node is no longer *reachable*, but may still be accessed by some thread.
4. *Retired.* The node is removed, and cannot be accessed by any thread, but it has not yet been freed.
5. *Free.* The node's memory is available for allocation.

Concurrent memory reclamation is defined as follows [51, 71]. We are given a node (or a set of nodes) which are *removed*, and we are asked to move them first to state *retired*, then to state *free.* Once in state *retired*, nodes are no longer accessible by any thread besides the reclaimer, and therefore cannot lead to access violations. The key step in the problem is deciding when a node can be *retired*, i.e., when it is no longer accessible by concurrent threads.

The properties concerning developers are intuitively all related to performance, but although such techniques are available, the difficulty in implementing or deploying them is a hurdle to the use of the concurrent data structures, at all. Certainly, the lack of libraries is a testament to this. I identify a few core properties to think about in the evaluation of any reclamation technique:

- **Automatic:** A technique is said to be automatic if the compiler can do the heavy-lifting in terms of tracking references and distinguishing between removed and retired objects. To satisfy this property it may be that a pointer needs to be labeled or a lexical scope needs to be cited, but a concurrent data structure can be implemented as presented in the literature without invasive changes or the creation of contracts with the structure's users.

- **Cheap Reads:** Since the problem is motivated by the presence of invisible readers, some techniques render readers visible by publishing what they're looking at. A technique with cheap reads is one that doesn't burden its reads with writes or memory fences.

- **Low Latency:** An application may be sensitive to latency imposed on threads, particularly threads not directly involved with updating the concurrent data structure. Threads involved in UI, networking, or drawing may preclude techniques that will cause noticable delays. Low latency typically means that periodic pauses (if they exist) can either be delayed/avoided, or are in the low millisecond or even down into the microsecond range. Some applications require even less latency.

- **Scales:** Most techniques will scale linearly up to a handful of threads, but crossing core or socket boundaries, or even simply growing into the dozens of threads may lead to extreme contention that causes performance to level off or even regress with thread count. This property has more to do with relative scalability between techniques than with objective measurements.

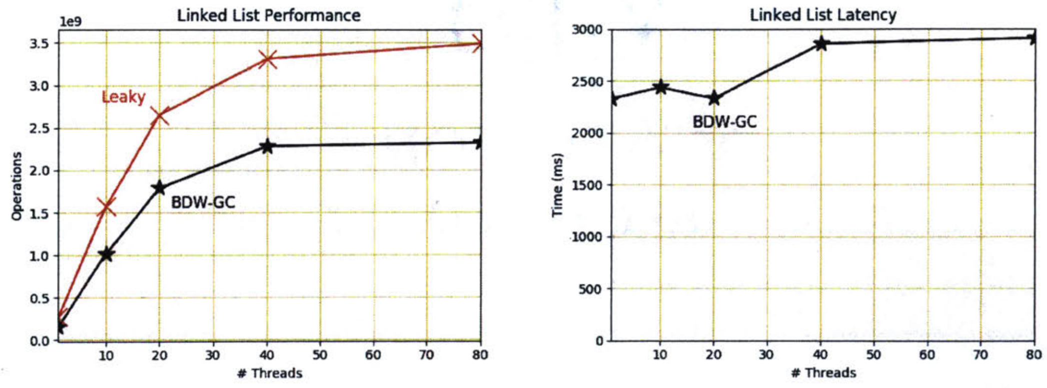


Figure 2-5: Garbage collection performance impact. Left: BDW scalability on a lock free linked list with respect to core count. Right: BDW average stop-the-world pause times on those executions.

2.5.1 Conservative Garbage Collection

Microsoft's DotNet framework contains a garbage collector, and Boehm, Demers, and Weiser developed BDW as cross-platform GC for C and C++.[72, 34] In high level languages, GC is integrated with the type system and knows what values are pointers. This is impossible in low level languages since pointers can be stored anywhere, and certain operations (like XOR) can obscure them. The implication is that during the root-finding phase of collection, wherein GC identifies pointers on the stack and in well-known global memory locations, the collector can't be sure if what it's looking at is a reference to memory that it cares about. The false positive case – a value, for example a floating point number, that looks like a pointer is misidentified as a pointer – happens, but is rare and doesn't lead to significant overhead in general.[15] The false negative case – when a pointer isn't recognized as such – is far more dangerous because it can lead to memory corruption.

Conservative collectors don't worry about false positives, and simply take the hit to memory consumption. Thus, conservative: anything that looks like a pointer is treated as a pointer. On the other hand, part of the collector's contract is that pointer manipulation is restricted. Typically, they allow for the bottom couple of bits to be overloaded, as some data structure implementations tend to do (e.g., node color in RB-Trees), but the rest of the bits are sacrosanct.

This system has performance limitations, particularly as the number of cores increases, and also as heap size grows (fig. 2-5). The fundamental problem is that there are no short sweeps in modern conservative GC, and each full sweep pauses all threads and doesn't let them resume until complete. There are no partial sweeps because the collector needs to scan a snapshot of memory, and without that snapshot, threads can hide references by moving them from an unscanned region of code to a scanned one, and removing the original reference causing the collector to reclaim the memory erroneously. Transferring a reference in a high level language can be tracked, and Go boasts collection latency in the single-digit millisecond range.[53] But these tricks aren't available in low level languages.

Performance limitations, including unpredictable, uncontrollable thread latency make GC unpalatable to most C and C++ applications. Moreover, high performance programmers may choose allocators based on their specific applications since certain kinds of workflows benefit from different allocation/deallocation techniques. Many highly optimized multithreaded allocators exist[8, 41, 37, 57] but GC incorporates its own memory allocator, thereby making the decision for the programmer. In spite of the convenience of GC, it has largely been rejected by the C and C++ communities.

- **Automatic:** Yes. No modification to a data structure is required. It just works.
- **Cheap Reads:** Yes. No burden is placed on any individual operation.
- **Low Latency:** No. Integrated GC can have low latency, but no method of applying it to low level languages in a way that doesn't impose significant burden has been discovered.
- **Scales:** No. Performance tends to level off after a few threads.

2.5.2 Epoch-based

On the other end of the spectrum from GC is an epoch-based system in which there is generally no measurable overhead for reclamation, but where transfer of references to objects is severely restricted. An *epoch* is a programmer-defined sandbox in which references can flow freely, but can never escape without causing erroneous reclamation. Each thread maintains an epoch counter and increments that counter to indicate that it's about to perform an operation involving shared objects. It increments the counter again when it's done, asserting that it holds no shared references. Node removal means unlinking the node from the data structure and placing in a pool of retired objects. When the pool is full every thread's epoch counter is checked, and all threads that aren't in epochs can't possibly be holding references to it. Any thread that's in an epoch may be holding references, so the pool can be put aside until those threads' counters have been incremented at least once. A subsequent check that discovers that those threads' counters have been incremented can mark the pool "clean" and reclaim the memory.

The traditional correctness limitation of epoch schemes has been the ability of a single thread to hold up all memory reclamation by hanging inside of an epoch and never updating its counter. More modern schemes like RCU[67] and RLU[66] help to resolve this problem using a more fine-grained mechanism that tracks individual references within the coarser data structure operations. Again, the performance is very good, but special care needs to be taken not to let references to shared objects escape. Libraries that use epoch-based mechanisms to reclaim memory, for example, form a contract with programmers that the epochs must be respected.

In constrast with GC, epoch-based memory reclamation is quite popular on account of its performance, including in the Linux kernel.[68] At the cost of automation, and a measure of space blowup, application of epoch to concurrent data structures tends to perform comparably to leaky implementations. An epoch sandbox isn't a general solution, however, as references may flow freely from one thread to another in some applications. Thus, the lack of C and C++ concurrency libraries, even implemented with epochs.

- **Automatic:** No. It's applied manually because it requires a higher level semantic understanding of the code, and a library imposes a contract on a user about how references may be handled.

- **Cheap Reads:** Yes. No burdening is necessary because the runtime can infer when no outstanding references remain.

- **Low Latency:** Yes. RCU/RLU don't require threads to pause for scanning.

- **Scales:** Yes. In fact, performance is typically comparable to a leaky implementation.

2.5.3 Reference Counting

Reference counting *can* be automated, but isn't in C or C++. In such a system, each object has a counter that stores the number of references to it. Each time a thread takes a reference to it the counter is incremented. When the thread drops the reference it decrements the counter, and when the counter reaches zero the object can be reclaimed.

This is a popular and effective technique for certain kinds of data structures, but less applicable to others. Overhead is quite low when traversals are few and short, as in a lightly loaded hash map, but high when traversals are many and long, as in a linked list or graph-like data structure. In the linked list case, even insertions and removals require extensive traversal, adding a write to each read and thrashing the cache when lots of threads are operating on it.

As with epoch-based reclamation, reference counting is used in many applications. The overhead is not so great on concurrent data structures with short traversals and low contention. The counter, too, usually introduces acceptable space overhead when objects are large. Nevertheless, these points limit its applicability. A C/C++ compiler augmented with reference counting wouldn't solve the memory reclamation problem for the classes of data structures where nodes are small, traversals are long, or where counters will cause cache contention.

- **Automatic:** Yes. In some languages it is, but typically not when languages are close to the machine. The possibility certainly exists.

- **Cheap Reads:** No. Invisible readers broadcast their traversals by adding [potentially-contentious] writes to every read.
- **Low Latency:** Yes. No threads ever need to pause.
- **Scales:** No. In some data structures, yes, but when any nodes receive high traffic, cache thrashing can impede speedup.

2.5.4 Pointer-based

Two of reference counting's three problems – per-node overhead and cache contention – are addressed with the various implementations of hazard pointers.[71, 80, 17] Instead of keeping a counter per object, each thread has a list of objects it's holding references to: its *hazard pointer*. When a node is removed from a concurrent data structure, each thread's hazard pointer is scanned to see if any outstanding references remain before freeing the memory. But in general, a thread's hazard pointer is accessed almost exclusively by its owner, making contention low on highly trafficked nodes.

Astute readers will identify a potential data race, here. A thread gets a reference to a node, adds it to its hazard pointer, and then dereferences the node. What if the thread hung between getting the reference and adding it to its hazard pointer, and another thread removed the node and freed its memory? The fix is to read, update hazard pointer, memory fence, and then read a second time to verify. As with reference counters, this can noticeably increase the cost of lengthy traversals. Efforts have been made to reduce the need for memory fences,[7, 80] though the read-write-verify sequence can still undermine the progress guarantees in data structures that use them in the literature.[18]

Once again, however, pointer-based techniques are not automatic. They have to be reimplemented for each data structure and they're complicated, making them a potential vector for bugs. And, as with epoch, a library with data structures with hazard pointers creates a contract with users of that library in how to protect and transfer references to internal objects.

- **Automatic:** No. Extensive modification has to be made to code presented in the literature.
- **Cheap Reads:** No. Even in the absence of a memory fence, the read-write-verify sequence is costly.
- **Low Latency:** Yes. There is no pause.
- **Scales:** Mostly. In cases where reference counting leads to contention, hazard pointers resolve this. But it's typically decidedly slower than an epoch-based implementation.

2.5.5 Ownership-based

Rust has a language solution to memory reclamation in concurrent code.[64] It's fully automatic without requiring GC because a programmer explicitly assigns "ownership" of memory to a single function at any given time. When a reference is passed to another function, that function receives ownership and the parent can't use the reference again unless its child relinquishes ownership. Thus, if the reference escapes from the child, and the child tries to relinquish ownership to its parent, the compiler catches the semantic error.

This is, of course, very fast because the compiler can statically infer precisely when to free memory – when the memory has no owner. The limitation, here, is that the problem has been defined away. Concurrent data structures are objects to which multiple threads hold references and may write, but Rust's approach is based on atomic reference counters.[88] Certainly, a programmer may identify a block of code as `unsafe`, and pass references around freely, but now the programmer is responsible, again, for solving the problem.

Insofar as Rust's model works within an application, this is a low cost method of achieving memory reclamation in multithreaded code, and no discussion of concurrency is complete without mentioning it. However, it addresses only the sub-problem in which every thread is a reader of shared data structures, or in which reference counting is suitable. For this reason, although it's necessary to mention Rust, it doesn't make sense to evaluate the properties of memory reclamation for comparison with the other methods discussed.

2.5.6 StackTrack

Alistarh, Eugster, Herlihy, Matveev, and Shavit took a step in a new direction, returning to a compiler-supported, semi-automatic approach to memory reclamation, but without the burden of GC. The high level observation, here, is that the sub-problem of automatic concurrent memory reclamation is easier to solve than the broader problem of automatic memory reclamation as a whole. StackTrack is a model for concurrent memory reclamation that can be embedded into a compiler that leverages hardware transactions to atomically search for reclaimable nodes.[1]

A thread that wants to free a node scans the stacks of all other threads to look for references. Naturally, those threads are all constantly updating their stacks, and data is hidden in their registers. Operations are augmented, therefore, with hardware transactions to maintain a consistent view for scanning threads, and registers are dumped at the beginning of each one. Since operations may be long or contentious, there's a technique for splitting an operation that's beyond the scope of this thesis. Aborts and instrumentation, the primary causes of overhead, increase in cost linearly with the number of threads. But the takeaway is that a flagged function could receive this augmentation by the compiler, and the overhead to the application is limited exclusively to use of the concurrent data structure.

This suffers from some of the same contention problems as reference counting, in that even traversals can be burdened by the instrumentation when scans occur. But the overhead of an operation has shifted to the stack from the data structure, itself, without impeding the ability to automate it – a significant advancement. Experimental results in the paper showed that StackTrack was able to perform competitively with, or outperform, hazard pointers.

- **Automatic:** Semi-automatic. StackTrack requires a programmer not to store

Technique	Automatic	Cheap Reads	Low Latency	Scales
Conservative GC	Yes	Yes	No	No
Epoch-based	No	Yes	Yes	Yes
Reference counting	Yes*	No	Yes	No
Pointer-based	No	No	Yes	Mostly
StackTrack	Semi	Yes	Yes	Mostly

Table 2.1: A qualitative comparison of conventional concurrent memory reclamation techniques.

references off the stack.

- **Cheap Reads:** Yes. Although instrumentation is added to operations, it's related to the management of hardware transactions. The cost is ammortized over many reads.
- **Low Latency:** Yes. No pause is required.
- **Scales:** Mostly. StackTrack compares favorably against pointer-based techniques, but transactions burden operations as thread count grows.

2.6 Summary

For DEF's purposes, it's clear that extensive work has been done for parallelism, and development continues on fork-join oriented models. DEF leverages the fork-join instructions available in the TAPIR[83] branch of LLVM,[58] introducing Cilk-like parallel semantics, and permitting the use of the various tools.[2] The alternative is threaded parallelism which, fast though modern implementations may be, puts more than a few instructions on the critical path and passes the headache of work distribution to the programmer, and precludes the use of tools available to the Cilk-like model.

Conventional techniques of concurrent memory reclamation (summed up in table 2.1) involve trade-offs that make them unsuitable for general purpose concurrent memory reclamation. Each has its applications, but a general purpose technique

[2]LLVM and TAPIR are discussed in greater detail in the DEF Overview chapter.

would need to meet all of these criteria for a low level programming language. I contend that this is the reason libraries don't exist for C and C++. Developers must decide which criteria are essential to their applications, and which are flexible.

In my work, this was the most significant barrier to thinking about concurrency in DEF. Choosing one of these techniques over the others necessarily limits its applicability. Moreover, the "automatic" criterion is non-negotiable since DEF has to interoperate seemlessly with C, and dropping in a general purpose data structure can't impose non-API-related restrictions on a C module such as pointer-based solutions would require. But conservative GC has generally been rejected by that community, and the cost of reference counting is far too great (to say nothing of reference counting not *actually* being automatic in C). StackTrack, on the other hand, represents a compelling direction to take memory reclamation because it isn't costly, either in space or in instrumentation, and it's semi-automatic.

Chapter 3

Retire

The problem of memory reclamation is an actual theoretical barrier to any generalized concurrent data structure in conventional low level programming language. My initial research was on resolving this problem with a library solution that would pull all of the memory management complexity away from the programmer; no epochs to track, no inline instrumentation, etc. From StackTrack, it was clear that a semi-automated solution was possible that didn't excessively burden normal operations. The important thing was that there was a snapshot to scan.

Removing the instrumentation was necessary for performance, and making it fully automatic was necessary for general applicability. These two steps were taken in sequence with the ThreadScan[4] and Forkscan[3] runtime libraries. ThreadScan was designed to eliminate the burden of instrumentation, except during an actual sweep, and Forkscan expanded the scan to the whole of memory (as conservative GC would do) with minimal pause in execution.

Interestingly, these libraries have a common approach to their interface that's both intuitive and minimally invasive to data structure source code. Generalized, this is a *retire* interface, in the sense that an object isn't known to be freeable, but that it's been removed from its concurrent data structure and is ready to be freed once all remaining outstanding references have disappeared. In a serial data structure where one might free a node, in ThreadScan and Forkscan a programmer retires that node, at which point the runtime system can track it and look for references.

Going forward, retire represents an approach to concurrent memory reclamation that's *automatic on-demand.* It's automatic in the sense that a node that's been retired is tracked by the system with the same theoretical properties as conservative GC, but it's on-demand because it tracks only memory that's been retired rather than all memory that's been allocated. The advantages of this method are discussed in detail below.

3.1 ThreadScan

3.1.1 Overview

ThreadScan is a compiler-agnostic step towards automatic reclamation without a garbage collector. It's a library that supports the ThreadScan and ThreadScan+ protocols, described below.

At a high level, ThreadScan works as follows: when a thread deletes a node, it adds it to a shared delete buffer. When the buffer becomes full, the thread inserting the last node initiates a **Retire** procedure, which examines memory for references to nodes in the delete buffer, and marks the nodes which still have outstanding references. The reclaiming thread can then free nodes which are no longer referenced.

The key challenge in ThreadScan is the automatic and efficient implementation of Retire.

Figure 3-1 illustrates the key idea of ThreadScan: when initiating a Retire, the reclaiming thread sends signals to all threads accessing the data structure, asking them to scan their own stacks and registers for references to nodes in the delete buffer, and to mark nodes in the delete buffer which may still be referenced. Threads execute this procedure as part of their signal handlers. At the end of this process, each thread replies with an acknowledgment, and resumes its execution. Once all acknowledgments have been received, the thread reclaims all unmarked nodes and returns.

There are two main advantages to this design. First, ThreadScan is shielded from errors in data structure code, such as infinite loops: these will not prevent the protocol from progressing, since the handler code always has precedence over application code[55].

Second, ThreadScan offers strong progress guarantees as long as the operating system does not starve threads. In particular, notice that, since the reclaiming thread waits for acknowledgments, the reclamation mechanism could in theory be *blocking*. This is not an issue in a standard programming environment, since each participant must finish executing ThreadScan in the signal handler before returning to its code. Therefore, the only way a thread may become unresponsive is if it is starved for steps *by the operating system*. This phenomenon is highly unlikely in modern operating systems, which schedule threads fairly: for instance, the Linux kernel avoids thread starvation by using dynamic priorities[62]. At the same time, we emphasize that all other data structure operations preserve their progress properties, as ThreadScan adds a bounded number of steps to their execution.

The cost of the memory scan is amortized among threads by having each thread scan its own stack and registers, marking referenced nodes in the delete buffer. The scan is performed word-by-word, checking each chunk against pointers in the delete buffer. ThreadScan does assume that the programmer will not actively "hide" pointers to live nodes.[1]

3.1.2 ThreadScan+

The ThreadScan+ protocol provides a semi-automatic solution for the extended memory reclamation: it can detect reference exchanges between threads.

An example of reference exchange is shown in Figure 3-2, in which Thread 1 sends the address of Obj1 to Thread 2. In the first step, Thread 1 sets a shared memory location S1 to the address of Obj1 and then sends a signal (or some indication) to Thread 2 that S1 is ready to be taken. In the next step, Thread 2 receives this indication and reads the value of S1. Now Thread 2 can access Obj1 that was initially

[1]This assumption has been addressed and justified as similar to those of conservative garbage collectors[15], and is necessary for *automatic* reclamation.

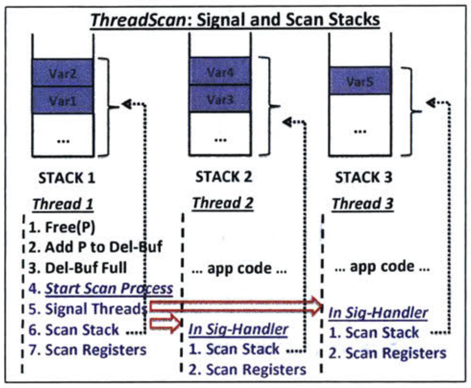


Figure 3-1: ThreadScan protocol illustration. Thread 1 calls Free(P) and this makes the delete buffer full. As a result, Thread 1 initiates a reclamation process, that sends a signal to other threads, and makes each thread to scan its own stack and registers. After all threads are done, Thread 1 traverses the delete buffer and deallocates nodes that have no outstanding reference to them.

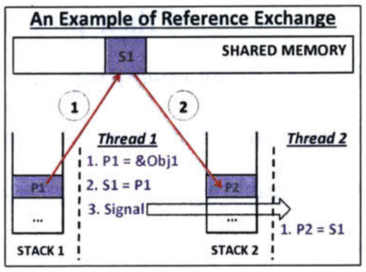


Figure 3-2: An example of reference exchange between threads via a shared memory location. In the first step, Thread 1 sets S1 to address of Obj1 and then sends a signal (or some indication) to Thread 2 that S1 is ready. In the next step, Thread 2 reads S1 and gets the address of Obj1.

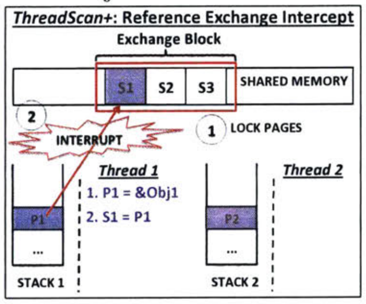


Figure 3-3: In ThreadScan+, the programmer defines a predefined exchange block, and then uses this block to execute reference exchanges. Then, the ThreadScan+ protocol locks the pages of this block before the scan process and this allows to intercept any reference exchange that occurs during the scan.

read by Thread 1. Notice, that the exchange process involves multiple steps. As a result, in between there may be a state where the value of this reference only resides in the shared variable S1, so a stack scan of Thread 1 and Thread 2 will not find this reference anywhere. Moreover, scanning the whole global memory range may also miss the exchange, since it may happen after the exchange is done, and the value of the shared variable S1 has already been reset back to zero.

The key idea in ThreadScan+ is to declare a specific shared memory block, called an *exchange block*, that the program will use for reference exchanges between threads. The operating system hardware page read-write protection mechanism will intercept writes to this block. In this way, the reclaiming thread can lock the pages of the exchange block before it signals other threads to start scanning their stacks and registers, and then, during the scan, a thread that tries to perform a reference exchange, will receive a page protection interrupt when it tries to write a reference to the exchange block. Then, after the threads have sent their acknowledgments, the reclaiming thread can finish by scanning the exchange block for additional references, while being certain it did not miss any reference exchange during the scan process.

Figure 3-3 depicts the key idea of ThreadScan+, and shows how it intercepts a reference exchange.

The requirement of ThreadScan+ is to declare the exchange block, once, when the program starts, and then the program can freely use shared memory locations from this block to perform exchanges. Notice that this approach is simpler than using hazard pointers which require the programmer to declare each shared memory location as a hazard pointer, and then constantly update the tracking information for each such pointer: a meticulous and inefficient process that requires the use of memory fences and validation steps, as opposed to a one-time declaration of a memory block.

In comparison to the basic ThreadScan, the additional costs of ThreadScan+ are amortized using large delete buffer sizes, and the progress guarantee of ThreadScan+ is OS-based: guaranteed to make progress as long as the operating system has no bugs and provides a fair scheduler.

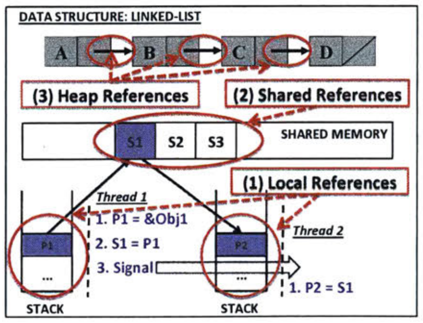


Figure 3-4: Reference types illustation for a linked-list data-structure and a reference exchange process between two threads.

3.1.3 Model

The abstract formulation of reclamation was described in section 2.5, but how do these definitions map onto a real data structure implementation? Since we wish to perform memory reclamation automatically, it is important to describe which code patterns are disallowed by the above problem definition.

In general, the assumption is that removed nodes are *unreachable*, but to understand this definition more precisely, we need to categorize the various reference types.

Reference Types. We categorize references into types to represent the specific memory access pattern of each reference. The possible types are as follows:

1. *Local Reference:* A reference that is local to a specific thread and is used only by this thread.

2. *Shared Reference:* A reference that is accessed by more than one thread (used by reference exchanges).

3. *Heap Reference:* A reference that is used to connect or "structure" objects in a data-structure (also called internal reference).

Fig. 3-4 shows how this reference type definitions apply to the example execution that was previously shown in figure 2-4. In the figure, we can see a linked-list data-structure and two threads that execute a reference exchange. The pointers that connect the nodes of a linked-list are the heap references, the local variables on the stack (and registers) of each thread are the local references, and the shared memory locations that are used for exchanges are the shared references.

Classical Memory Reclamation. In the classical memory reclamation, the following is assumed.

Assumption 1 (Reachability). *An* unreachable *node has no:*

1. Heap References. *Pointers to this node from the inside of a data-structure.*
2. Shared References. *Pointers to this node from shared memory locations that are used for exchanges.*

Practically, the above assumption limits where outstanding references to nodes in the delete buffer may be located. In particular, notice it implies that nodes may only hold local references to such nodes *in their stacks or registers.* Notice however both statements limit the programming patterns the code can employ. We will show that these assumptions are not necessary for *efficient* reclamation.

Extended Memory Reclamation. In the extended memory reclamation, that is able to detect reference exchanges, the following is assumed.

Assumption 2 (Extended Reachability). *An* unreachable *node has no:*

1. Heap References. *Pointers to this node from the inside of a data-structure.*

Both protocols we propose work under the following conservative assumptions:

Assumption 3. *It is assumed the following hold.*

1. Reference Matching. *Node references are word-aligned, and they can be matched to node pointers via comparison. Arbitrary memory words will not be matched to node addresses.*
2. Reclamation Rate. *There exists a finite bound k on the number of* reclaim *signals that a method m may receive during its execution.*

Discussion. The *heap references* assumption follows from the definition of concurrent memory reclamation [51, 71], and in fact is one of the main distinctions from garbage collection. The absence of *shared references* appears to be inherently implied by known memory reclamation schemes, although we revisit it in this paper.

Assumption 3.1 is standard for conservative garbage collection, preventing the programmer from hiding references from the scan process through arithmetic operations such as XOR. These assumptions are intuitively necessary for automatic reclamation. This doesn't change in DEF even though, in principle, compiler support is available, because the programmer has direct access to the bits that make up a pointer. Assumption 3.2 is justified by the fact that we reclaim memory infrequently, by choosing a large delete buffer size.

3.1.4 The ThreadScan Algorithm

Generic Structure. Our algorithm is based on the following blueprint: once a thread wishes to reclaim a node, it adds a pointer to this node to a delete buffer, whose size is fixed by the application. Whenever the buffer is full, the thread which inserted the last node into the buffer becomes the *reclaimer*. For simplicity, we assume that there can only be a single reclaimer at a given point in time. (In practice, this follows by choosing a buffer of appropriate size)

The chosen thread starts a ThreadScan Retire operation by signaling all other threads to help with examining references to nodes in the buffer. Thus, the ThreadScan algorithm consists of the implementation of the `retire` procedure on the reclaimer side, and of the implementation of the `scan` signal handler for all other threads accessing the data structure.

The Basic ThreadScan Algorithm

In the following, we describe the implementation of `retire` which works under Assumption 1 and Assumption 3.

The implementation ensures that, at the end of the `retire`, each node in the buffer is either *marked* or *unmarked*. Marked nodes may still have outstanding references, and cannot yet be reclaimed. Unmarked nodes are safe for reclamation, and are freed by the thread as soon as the `retire` procedure completes.

The `TS-Reclaim` procedure, whose pseudocode is given in Algorithm 1, works as follows. The reclaiming thread first sorts the delete buffer, to speed up the scan process. Next, the reclaimer signals all other participating threads to start a `TS-Scan` procedure, and executes this procedure itself. This procedure will mark all nodes with outstanding references. The reclaimer then waits for an acknowledgment from all other threads. Once it receives all thread acknowledgments, it scans the delete buffer and frees all unmarked nodes.

The `TS-scan` procedure is called by the signal handler for all participating threads. Each thread scans its stack and registers word-by-word,[2] and checks whether each chunk is a reference to a node in the delete buffer, via binary search. If a possible reference is found, the node is marked in the delete buffer, which prevents it from being deleted in this reclamation phase. At the end of the scan, the thread sends an acknowledgment to the reclaimer, and returns to its regular execution.

The Extended ThreadScan+ Algorithm

The ThreadScan+ algorithm solves the *extended* memory reclamation which works under Assumption 2 and Assumption 3.

Practically, we are given a set of nodes that have been unlinked from the data structure, i.e., there are no more heap references to these nodes from inside the data-structure, and we must reclaim a subset of these nodes, which do not have outstanding references. However, in the extended version, threads may exchange references between them, and therefore, references to these nodes may still exist in shared memory locations.

The ThreadScan+'s semi-automatic idea is to define a specific shared memory range, called an *exchange block*, that the programmer can use to perform reference exchanges between threads. In effect, those shared references only reside inside the

[2]The details of this procedure are given in the Details subsection.

Algorithm 1 ThreadScan Pseudocode.

```
function TS-RECLAIM( delete_buffer )
    sort(delete_buffer)
    for each thread $t$ do
        signal($t$, scan)
    end for

    TS-Scan(delete_buffer)

    wait for ACK from all other threads

    for each pointer $p$ in delete_buffer do
        if delete_buffer[$p$].marked == false then
            free($p$)
        end if
    end for
end function

function TS-SCAN( delete_buffer )
    for each word chunk in thread's stack and registers do
        index = binary-search(delete_buffer, chunk)
        if $index \neq -1$ then
            delete_buffer[index].marked = true
        end if
    end for
    signal(reclaimer, ACK)
end function
```

limits of the exchange block.

Figure 2 shows the pseudo-code for ThreadScan+: the reclaiming thread executes the `TS-Retire` procedure, and all threads execute the `TSP-Scan` signal handler to scan the stacks and registers.

The only difference between ThreadScan+ and the basic ThreadScan is the need to handle the exchange block. In the extended version, the reclaiming thread first locks the exchange block to write-access before signaling threads to scan the stacks and registers. Then, after all threads have been signalled and have scanned their stacks and registers, the reclaiming thread scans the exchange block for possible shared shared references and unlocks the exchange block. As a result, the ThreadScan+ algorithm cannot miss any reference exchange that could occur during the scan process, since a thread that tries to perform an exchange is going to be interrupted by a write to the exchange block.

A possible optimization is to use the design of the Linux signaling mechanism: it ensures that when the `signal()` call returns, the signaled thread has received its signal and cannot continue executing code of the application that may perform exchanges. This behavior allows the reclaimer to scan of the exchange block and unlock write-access to this block immediately after all threads are signaled.

Implementation Details

The previous section provides a detailed overview of our technique, but, to simplify the presentation, omits several important implementation details. We provide these details here.

Signaling. We use POSIX Signals[90] for inter-thread communication. A thread that receives a signal is interrupted by the OS and begins running the signal handler immediately[55]. If the thread is stalled, the signal is delivered as soon as the thread is resumed. If the thread is running, it is interrupted and the signal is delivered at the completion of a system call.

Algorithm 2 ThreadScan+ Pseudocode. The only changes from ThreadScan are lines: 3, 9, 10, and 18 - 24.

```
 1: function TSP-Reclaim( $delete\_buffer$ )
 2:     sort($delete\_buffer$)
 3:     lock exchange block: make each page read-only
 4:     for each thread $t$ do
 5:         signal($t$, scan)
 6:     end for
 7:     TS-Scan($delete\_buffer$)
 8:     wait for ACK from all other threads
 9:     TSP-Scan-Exchange-Block( $delete\_buffer$ )
10:     unlock exchange block: make each page read-write
11:     for each pointer $p$ in $delete\_buffer$ do
12:         if $delete\_buffer[p]$.marked == false then
13:             free($p$)
14:         end if
15:     end for
16: end function
17:
18: function TSP-Scan-Exchange-Block( $delete\_buffer$ )
19:     for each word $chunk$ in exchange-block-range do
20:         $index$ = binary-search($delete\_buffer$, $chunk$)
21:         if $index \neq -1$ then
22:             $delete\_buffer[index]$.marked = true
23:         end if
24:     end for
25: end function
```

Progress. Because the signal handler takes precedence over the application code, anything in the user code that is stalled, including waiting on locks, is preempted. Although the threads do not operate at the OS level while they are handling signals, the only things that can cause them to block are in the ThreadScan code, itself, or other signals. Therefore, if the OS scheduler is fair, ThreadScan can be analyzed in isolation from the user code.

Page-Write Access Violations. When a thread attempts to write to a write-protected page, a seg-fault interrupt is generated. The ThreadScan+ implementation catches such interrupts and determines which seg-faults are due to its execution, or whether they are actual application seg-faults. If they are due to the ThreadScan+, i.e., if the thread was trying to write to an address in the exchange block when it was locked (read-only pages), then that thread busy-waits for the exchange block to be scanned and released. Once the exchange block is released (becomes writable), the busy-waiting thread resumes from the interrupted memory write instruction.

Stack Boundaries. Our implementation assumes a simple stack structure: each thread's stack is a contiguous memory range for which the base of the stack is the the same base as when the thread was created, and an *RSP* register that points to the top of the active stack. We use this assumption to simplify our proof-of-concept implementation and note that it is possible to have more complex non-contiguous stacks that require special customized code to walk the stack correctly.

Reclamation. Our presentation assumed a single shared delete buffer to which pointers to reclaimed nodes are added. We implement a distributed version of this buffer to avoid false sharing on the buffer and contention on the index. Specifically, each thread has its own local buffer to which it adds pointers. When an individual buffer becomes full, that thread becomes the reclaimer by grabbing a *reclamation lock*. The reclaimer aggregates the pointers from all of the threads' buffers into a master buffer used during the scan. Individual buffers are circular arrays that are guaranteed to be single-reader, single-writer. Monotonically increasing head and tail indices are kept and incremented as elements are added to or removed from a buffer. Since the indices are absolute, they can always be compared to one another to see if values are waiting or if the buffer is full. An index can be converted into an address by masking the low-order bits, since the buffers are powers of two in length. Additionally, there are no race conditions on the individual buffers. The owning thread adds values by inserting an element and incrementing the head pointer. If the buffer is full, it tries to become the reclaimer. The reclaimer empties a queue by caching the head, performing a `memcpy` on the range of elements it wants to add to the master delete list, and updates the tail pointer.

The reclamation lock ensures that there is always at most a single active reclaimer. In general, large delete buffer sizes ensure that this lock is not contended. Also, the above buffer construction has the consequence that a thread waiting to become a reclaimer will discover that its buffer has been drained into the master buffer, and that it can go back to work.

Pointer Obfuscation. In keeping with conservative GC, ThreadScan assumes that the underlying code does not use pointer obfuscation beyond using the low order bits. If the pointer is obfuscated in some other way (like XOR-ing) or if it is not word-aligned, the reference will not be detected. It also assumes that arbitrary memory chunks cannot be interpreted as addresses to existing nodes in the delete buffer. While breaking this assumption does not affect the correctness of our protocol, it could prevent the reclamation of the target nodes. We did not observe this phenomenon in practice, and considered it highly unlikely.

3.1.5 ThreadScan Correctness Properties

Linearizability. Applying ThreadScan or ThreadScan+ does not affect the linearizability of method invocations.

Lemma 1. *For any correct, linearizable implementation of a method call m, its variant m', which uses the ThreadScan or ThreadScan+ algorithms to reclaim memory, is also linearizable.*

Proof. There are two points of interference introduced by ThreadScan/ThreadScan+: the static one in which a node is retired during a removal operation, and the dynamic one in which a thread is interrupted at an arbitrary time.

In the static case, this is a matter of a call to `retire` from within the operation. Since the ThreadScan library has no internal knowledge of the data structure, none of its writes alter it in any way. The only possible exception would be the pointer to the retired node, but this is only read and never dereferenced unless the thread becomes the reclaimer. As such, the time ThreadScan might alter the linearizability property is during reclamation. Therefore, `retire` is simply a place within the operation like any other, and the dynamic interruption case covers it.

The dynamic case involves pausing all work in the program. Since m was linearizable before ThreadScan was introduced, m' remains linearizable if ThreadScan can be modeled, from the data structure's point of view, as an arbitrary delay in any subset of threads. Indeed, if ThreadScan's operations on any user-reachable memory is purely reads, then it can be modeled as such. The scan is entirely reads, except for marking pointers in the delete buffer, which isn't reachable memory from the user code. Calling `free` on objects deemed to be truly retired involves writes for most allocators, but none of those objects are reachable, assuming the algorithm is correct.

Therefore, there is no point at which ThreadScan writes to user-reachable memory, so it is perceived as an arbitrary delay by m', which is only different from m possibly by the inclusion of a call to `retire`. If m is linearizable, then so is m'.

$\square$

Correctness. Under the standard conservative assumptions, any node reclaimed by our algorithms cannot be accessed by threads.

Lemma 2. *Under Assumptions 1 and 3, every node* reclaimed *by ThreadScan has already been* retired. *The same holds for ThreadScan+, under Assumptions 2 and 3.*

Proof. Assume, for the sake of contradiction, a node is reclaimed that wasn't retired. Since ThreadScan and ThreadScan+ pause the process, allowing them to scan a snapshot of memory, that would mean that they missed an outstanding reference. That reference must have existed as a heap reference or a shared reference by the definitions in Assumptions 1 and 2, for the respective algorithms. But under Assumption 3.1 all outstanding references in any region scanned will be detected.

A contradiction. Therefore, no reclaimed node was not retired. □

Progress. Finally, our algorithms ensure termination and reclamation under fair schedulers.

Lemma 3. *Under Assumption 3 and any fair scheduler, the ThreadScan and ThreadScan+ calls complete within a finite number of steps, irrespective of the progress conditions of method calls in the original implementation. Moreover, all nodes which can not be accessed through references in stacks or registers at the beginning of the phase will be retired.*

Proof. We can compute the maximum number of steps t threads take in terms of the stack size, s_s, the number of nodes in the pool waiting to be retired, n, and the average node size, s_n. A thread must scan its own stack, which takes $t * s_s$ steps. Recursive search could, theoretically, reach every node in the pool for an additional $n * s_n$ steps. Threads don't duplicate the recursive search, so $n*s_n$ is a fixed maximum, however it's divided. With t threads, therefore, the maximum number of steps is $(t*s_s)+(n*s_n)$.[3]

Regarding accessability, all nodes for which no references were found are unmarked and are therefore acknowledged as truly retired. The proof is similar to that of Correctness. The nodes recognized as retired are subsequently reclaimed.

□

3.1.6 Experimental Results

Experimental Setup. We tested ThreadScan on an 80-way Intel Xeon 2.4 GHz processor with 40-cores, where each core can multiplex 2 hardware threads. ThreadScan was configured to store up to 16,384 pointers per thread. Threads tended to have relatively full buffers, so the total number of pointers any reclaimer worked with

[3]Practically, the number of steps is closer to $t * s_s$ since few retired nodes will have outstanding references. This property is exploited by Forkscan and discussed in that section.

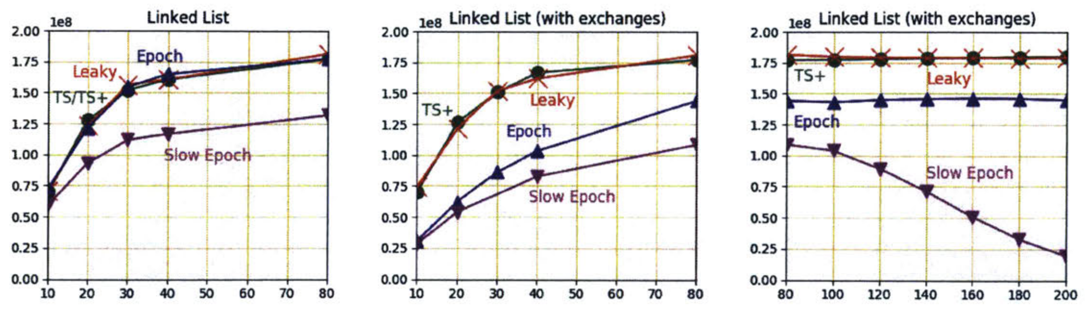


Figure 3-5: Throughput results for the lock free linked list: X-axis shows the number of threads, and Y-axis the total number of completed operations. From left to right: No exchanges, simulated exchanges, and simulated exchanges on an over-subscribed system (we use an 80-way machine).

was roughly 16,000 times the number of threads in the process. For all tests, we use the highly scalalable TCMalloc[41] allocator.

Data Structures. The data structures that were tested were a lock-free linked list[52], the lock-free hash table from Synchrobench[47], and a lock-based skip list provided as part of StackTrack[1]. The linked list nodes were 172 bytes to simulate nodes from a data structure of reasonable size and to avoid false sharing between nodes. It was implemented as a conversion from the Java code provided in[52] within Synchrobench. The hash table used lock-free linked lists as its buckets, so individual nodes also were 172 bytes. The StackTrack skip list nodes were all 104 bytes, representing their maximum size due to height. No padding was added to the skip list nodes.

Techniques. We tested the data structures using the following reclamation techniques.

1. **Leaky:** The original memory leaking data-structure implementation that has no memory reclamation.

2. **Hazard Pointers:** As introduced by Michael et al.[71]. The programmer manually declares and constantly updates the hazard pointer tracking information

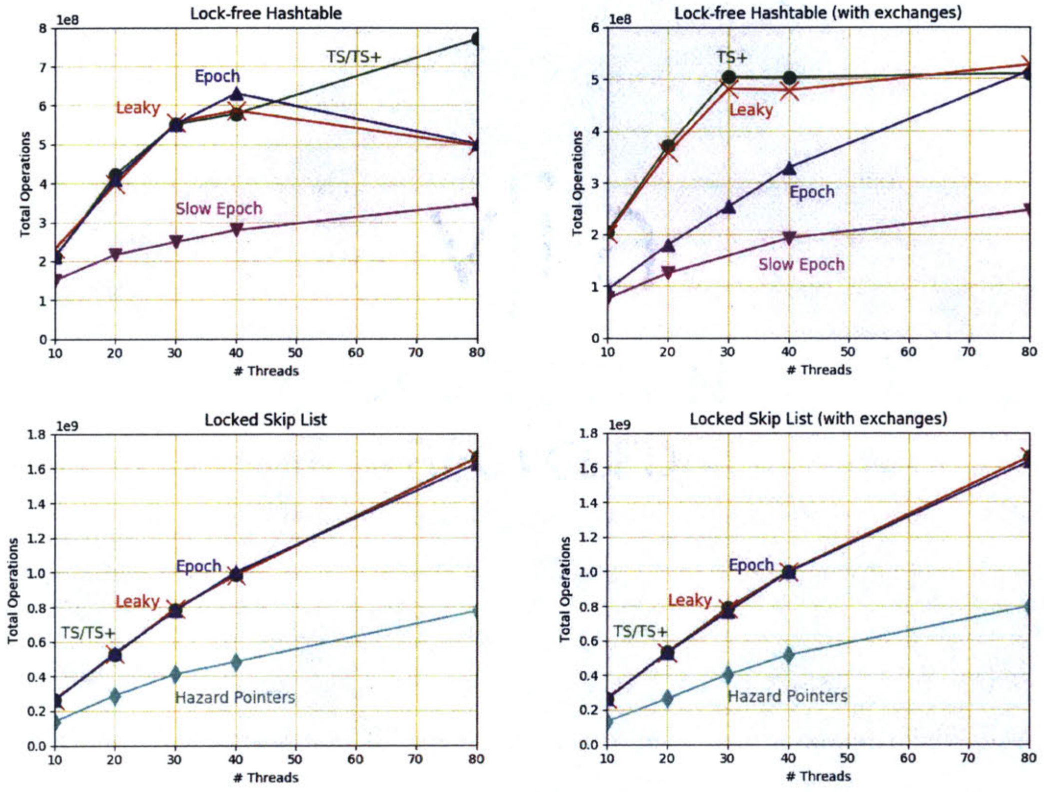


Figure 3-6: Throughput results for the lock free hash-table (first and second) and lock based skip-list (third and fourth).

for shared memory accesses, and the reclaiming thread scans this information to determine nodes that can be deallocated.

3. **Epoch:** As introduced by Harris et al.[49] and McKenney et al.[67]. The programmer delimits the epoch-start and epoch-end points in the code, and the reclaimer waits for the epoch to pass, at which point it is safe to deallocate nodes.

4. **Slow Epoch:** Represents the sensitivity of Epoch to application code that has thread delays: simulated by a 10ms busy-wait in the epoch phase, that executes infrequently, only once per 65,536 epoch phases (operations).

5. **TS (ThreadScan):** Our new fully automatic technique as described in Section 3.1.4, that solves the classical memory reclamation problem.

6. **TS+ (ThreadScan+):** Our new semi-automatic technique as described in Section 3.1.4, that solves the extended problem. When both protocols have the same performance we use the TS/TS+ notation.

Simulating reference exchanges. For each data structure, for each technique, a variant of the test was run that simulated reference exchanges using the exchange block, 4KB (one page) in size. The simulation idea is to execute infrequent shared memory writes to this exchange block, and in this way, simulate page write-access violations that occur when this block is locked by the reclaimer scan procedure in the ThreadScan+ protocol.

As we said before, Hazard Pointers and Epoch cannot detect reference exchanges, and therefore, we are forced to modify these algorithms to provide a comparison to ThreadScan+. Our modifications are simple and work as follows. For Hazard Pointers, we set a hazard pointer on the reference, when the exchange starts, and only release the hazard pointer when the exchange is completed (and there is no concurrent scan). For Epoch, we delay the epoch-end point to the point where the receiver is "done" with the reference (simulated using a short 6-10μs busy-wait). Notice that these modifications do not apply for the standard executions that have no exchanges, and that there may be other ways to implement these modifications, which is a topic for future work but beyond the scope of this thesis.

Methodology. Each data point in the graphs represents the average number of operations over five executions of 10 seconds. The update ratio was set at 20%, so about 10% of all operations were node removals. In tests where an exchange was simulated, those exchanges were performed alongside remove operations, so about 10% of operations included a reference exchange.

Results. Figure 3-5 shows the results for the lock free linked list. Lists were given an initial length of 256 elements. The only modification made to the code was the insertion of a call to `TS-Collect`(*node*) after an attempted physical removal of the node in the delete operation.

Linked-list results show that TS/TS+ scale perfectly along with Leaky up to the full 80 threads, and this indicates that the costs of processing seg-faults, signals and waiting for scans, are effectively amortized among the `TS-Collect` calls. However, this is not the same for the Epoch scheme, that scales well for the execution that has no exchanges, but incurs severe overheads when the application code has thread delays or when there are reference exchanges. Recall that for each exchange, an epoch-based thread must delay the epoch until the receiver is "done" with the reference, and this penalizes the Epoch's execution that must wait for the epoch to complete.

The linked-list results also show trials in which the system is over-subscribed (right graph): it executes up to 200 threads. These results demonstrate that Epoch is sensitive to context-switch delays, which effectively introduce delays into the application code and penalize the Epoch. ThreadScan+, however, is not sensitive to those delays and maintains full scalability along with Leaky.

The the Leaky hash-table (Fig. 3-6) peaks around 40 threads, and we can see that the TS/TS+ outperforms the Leaky as the number of threads grows. This advantage is due to "node re-use": The memory footprint remains relatively small and stable (less paging) when new nodes are allocated from existing memory. The benefit of reusing nodes is visible on the hash table because its operations are inexpensive as compared with the linked-list operations (expected $O(1)$ versus $O(n)$). As with the linked list, Epoch scales similarly well, but incurs severe overheads when there are application delays or reference exchanges.

The locked skip list experiments (Fig. 3-6) were conducted on skip lists initialized to length 100,000. TS+ remains comparable to the Leaky with or without exchanges, and we can see that Hazard Pointers, that explicitly update tracking information, have poor scalability in either case. Notice, that Hazard Pointers experience the same severe penalties for the linked list and the hash table, as was shown in [1] (we do not show these results here).

3.1.7 Summary

ThreadScan is an automatic and scalable approach for solving the *classical* memory reclamation problem. In addition, it defines the *extended* memory reclamation problem: an extension of the classical problem that can also detect reference exchanges between threads, and have provided the semi-automatic ThreadScan+ extension that exhibits the same scalability benefits as the original ThreadScan algorithm. Note that these techniques differ from GC in that they don't track any memory the programmer hasn't flagged for retirement, but they still track the memory automatically once it has been flagged.

Empirical results show that the ThreadScan protocols match or outperform conventional memory reclamation techniques, while requiring negligible programming effort beyond the standard use of Malloc and Free.

However, the algorithm requires the programmer to flag the space used for transfers if such tranfers are expected to occur. The reason for this is simple: ThreadScan ensures the safety of its scan by pausing all threads for its duration. In a library setting where references are returned to end-programmers, the safe thing to do is to flag the whole program heap (excluding the memory used directly by ThreadScan). Naturally, the resulting delay is proportional to the size of the process and the proof that the number of steps is bounded becomes invalid.

We can't side-step this theoretical limitation by arguing the practical delay is short or system-related. Root-finding on the whole process is time intensive (as we will see in Forkscan), and although ThreadScan can be categorized as a system in the same way that an allocator is, nobody tolerates a system that doesn't practically perform.

ThreadScan is a step forward in concurrent memory reclamation inasmuch as it solves the interface problem without creating new dependencies on compiler. Moreover, for libraries implemented using it, the contractual burden on library-users is far lighter than it would be for, e.g., an Epoch-based mechanism. Merely, the contract still exists, and we transition to Forkscan to examine how to eliminate it altogether.

3.2 Forkscan

3.2.1 Overview

In contrast to ThreadScan, Forkscan is both fully automatic and performs a complete scan of memory. It maintains the same *retire* as ThreadScan, but scanning all of memory means the only restrictions on pointer usage are those imposed by conservative GC. At a high level, Forkscan has a simple division of labor with execution structured around *collection operations*. Each such operation consists of two phases: *freeze-and-fork* and *scan-and-mark*.

The freeze-and-fork phase works as follows. We reserve a reclaiming thread (the "reclaimer"), whose purpose is to wait for and receive a list of pointers to memory blocks that are candidates for deletion. Upon receiving such a list from a user thread, the reclaimer starts a reclamation operation by sending a signal to all other threads. Upon receipt of this signal, each thread writes out its current stack boundaries and register contents, replies with an acknowledgment, and waits. When the reclaimer has received acknowledgments from all threads, the memory is "frozen" of thread activity. At this time, the reclaimer thread forks off another process (the "scanner"), and then immediately signals all threads to resume their regular execution.

Next, the child process performs a parallel scan of memory. Note that the child inherits a proper memory snapshot at the "freezing" point, whose consistency is maintained by the system through the copy-on-write mechanism. The scanner partitions memory, and spawns siblings which iterate sequentially through each partition, attempting to interpret words as pointers to nodes in a list of reclamation candidates.[4] If a presumed pointer to a node is found, it is marked and recursively searched. This technique allows Forkscan to detect any cycles between retired nodes. At the end of the scan, the last forked child notifies its parent and terminates.

The Forkscan implementation involves a few non-trivial observations and techniques:

First, it is important to note that having a *retired node list* can make collection significantly more efficient. ForkScan's implementation exploits this fact to perform a *linear scan* of memory (as opposed to tracing) comparing each memory location against the list via a carefully optimized binary search procedure. This is a critical performance optimization, since most of the memory is "outside" the retired node list, and this memory is scanned linearly by the first phase, which is friendly for the CPU pre-fetching, caching, and page-fault mechanisms. As a result, the second phase, which performs an expensive recursive search and mark and is responsible for detecting cycles (in GC style), needs to scan much less memory – only memory that is part of the retired node list.

Second, node de-allocation is carefully piggybacked on top of allocation calls, to avoid the overheads of bulk deletes while bounding memory usage. A thread which wants to perform an allocation must first see if nodes are available to be freed. This allows the user threads to perform cleanup without introducing unpredictable wait times.

Finally, Forkscan induces blocking thread behaviour in theory; however, the handshake mechanism is implemented through signaling, and each thread must complete

[4]Since it always pessimistically assumes that matched pointers are node references, Forkscan is *conservative* in the same way as ThreadScan and conservative GC.

handler code before returning to user-level code. Thus, a thread is unresponsive only if starved for steps by the operating system, which only occurs in extreme conditions. Hence, we argue that in practice, Forkscan preserves the non-blocking nature of data structures.

Forkscan is allocator-agnostic. Whereas garbage collectors tend to own their allocators, Forkscan will sit on top of another allocator, such as [37, 41], and `malloc()` and `free()` from it. This provides some flexibility to low level programming languages that use it, since programmers often choose allocators tailored to their specific workloads.

3.2.2 The Forkscan Algorithm

Forkscan uses the following pattern: Each thread maintains a pool of nodes to be reclaimed. The pool is populated by concurrent data structures, which call `forkscan_retire()` on nodes as they are removed. In the pattern of popular concurrent data structures, these nodes are unlikely to be seen by other threads. When a thread's pool becomes full, it consolidates all thread pools and hands it to a specialized *reclaimer* thread from the Forkscan runtime. We call the consolidated set of pools the *delete buffer*.

The reclaimer then starts a *reclamation phase*, attempting to purge the delete buffer. A reclamation phase consists of several steps. We describe these steps in detail below, and present pseudo-code in Algorithms 3 and 4.

Step 1: Freeze-and-fork. The first step aims to obtain a coherent snapshot of the application's memory. For this, the reclaimer first broadcasts a signal to all other threads, which handle the signal by writing out their current stack boundaries and register contents. Subsequently, each thread sends an acknowledgment, and waits for confirmation. Once the reclaimer has collected acknowledgments from all application threads, it *forks* a new process, whose task will be to scan memory. As soon as the fork returns, the reclaimer releases all other threads to return to their regular execution, and waits for the child to complete the scan.

Two observations are important at this point. The first is that, at the time T when the reclaimer calls `fork()`, all threads have written their current stack boundaries and register contents to memory, without executing further. Second, by the semantics of `fork()`, the child process will observe a *consistent snapshot* of the program's memory at time T, including heap, stack, and register contents. Therefore, to determine whether outstanding references to delete candidates still exist, it is sufficient for this child process to simply scan the heap, stack, and register contents as it observes them.

Step 2: Scanning. The child process begins by identifying the memory ranges which need to be scanned, and partitions them into M disjoint ranges, where $M \geq 1$ is a parameter. It then forks $M - 1$ sibling processes, such that each has a subrange that can be scanned in parallel. Scanning is broken into two parts: 1. *find roots* and 2. *recursive mark.* Finding roots means searching through memory for references outside of the delete buffer to nodes inside of it. The processes scan memory, avoiding the nodes in the buffer, for references, and marking the pointers in the buffer when they are found. The delete buffer is in shared memory between the sibling processes, so writes are visible to all siblings.

After the roots have been found, the delete buffer is broken up into chunks for the sibling processes, and marked references are recursively searched for further references. At the end of this phase, all nodes that are visible from the user program have been discovered and marked. Any unmarked nodes are no longer known to the user, and are available to be freed. This marking technique allows cycles to be discovered, so that nodes that point to one another, but are not pointed to from outside can be freed.

When the scan is complete, the last of the children notifies the reclaimer thread in the parent (via a pipe) and winds down. The reclaimer thread runs down the buffer looking for reachable nodes and preserves them for the next round of reclamation.

Step 3: Deletion. The previous step identified a set of nodes which can be safely deleted. It is tempting to free all these blocks at this time. However, in practice this leads to poor performance. If freeing is done by the reclaimer, reclamation iterations are delayed. If it is done by other internal threads, those threads compete for time and memory resources with user threads. And if user threads are signaled again to perform the task, Forkscan introduces high latency (which is what low level programmers want to avoid).

We therefore delay the free calls by piggybacking them on future malloc calls, allowing the user threads to perform the deletion phase without introducing high latency. This amortizes the free calls via malloc calls, while at the same time roughly matching the frequency of allocation with that of de-allocation.

The delete buffer is preserved, and a user thread that wants to allocate memory will first free some of the nodes in it. Each thread, when it has no nodes to free, will reserve a portion of the buffer that it is responsible for freeing. With each allocation, the thread will traverse part of its range, identify a small number of unmarked references, and free them. The delete buffer is reference counted, so when it has no more ranges to reserve, and when its reference count hits zero, it can be reused.

3.2.3 Implementation Details

Allocation and Retirement. `forkscan_malloc()` is provided as a wrapper for `malloc()` and has the same profile. If nodes are available to be freed, it will do so before returning new memory to the user. The main interface to Forkscan is `forkscan_retire()`, which behaves like `free()` from the user's perspective except that instead of immediately freeing the node, it adds it to the thread's pool, checks to see if the pool is full, and possibly becomes the consolidator. It should be noted that `forkscan_retire()` is a proper-replacement for `free()` in that retiring the same node multiple times or from multiple threads could have the same unpredictable effects as a double-free. Likewise, both retiring and freeing a node will lead to unpredictable behavior.

Forkscan is allocator-agnostic and treats the underlying library as a black box. The je_malloc allocator [37] was selected for experimentation because it had the best performance in trials on the microbenchmarks. However, Forkscan only requires

Algorithm 3 Parent Process Pseudocode.

```
 1: function CONSOLIDATE_PTRS
        ▷ Called when a user thread pool is full.
        ▷ Aggregate pointers from all threads
 2:     delete-buffer ← $\emptyset$
 3:     for th $\in$ threads do
 4:         delete-buffer $\cup =$ GET_PTRS_POOL(th)
 5:     end for
 6:     SORT(delete-buffer)
 7:     SIGNAL-CONDITION-VAR(reclaimer-conditional)
 8: end function

 9: function SNAPSHOT_SIGNAL_HANDLER(ctx)
        ▷ Executed by thread on snapshot signal
10:     Spill registers and stack boundaries to stack
11:     Send ACK to reclaimer-thread
12:     Wait for resume signal                              ▷ Freeze point for snapshot
13: end function

14: function RECLAIMER_THREAD(ctx)
15:     while 1 do
16:         WAIT-ON-CONDITION-VAR(reclaimer-conditional)
        ▷ Signal all threads
17:         for th $\in$ threads do
18:             SIGNAL(th, snapshot)
19:         end for
20:         wait for ACK from all threads
        ▷ At this point, the system is "frozen," so we fork
21:         pid ← FORK()
22:         if pid = 0 then
        ▷ The child scans the memory snapshot
23:             SCAN(ctx)
24:             EXIT()
25:         end if
        ▷ This is the parent
26:         resume all threads via signal                       ▷ Child got snapshot
27:         wait for child to finish
28:         PUSH-BACK(delete-buffer)                                ▷ Free memory
29:     end while
30: end function
```

Algorithm 4 Child Process Pseudocode.

```
1: function SCAN(ctx)
2:     memory-ranges ← GET_MAPPED_RANGES()
3:     Split memory-ranges into M partitions
4:     for id ∈ [1..M - 1] do
5:         pid ← FORK()
6:         if pid = 0 then
7:             SCAN_FOR_REFS(memory-ranges[id])
8:             EXIT()
9:         end if
10:    end for
11:    SCAN_FOR_REFS(memory-ranges[0])
12:    Wait for children to finish
13: end function

14: function SCAN_FOR_REFS(memory-ranges)
15:    for each word ∈ memory-ranges do
         ▷ Check if word is a reference to some object
16:        i ← BINARY-SEARCH(word, delete-buffer)
17:        if i $\neq$ 0 then
         ▷ Found a reference → record it
18:            SET-LOW-BIT(delete-buffer[i])
19:            SCAN_FOR_REFS(delete-buffer[i])
20:        end if
21:    end for
22: end function
```

the names of `malloc()`, `free()`, and `malloc_usable_size()` (the last is a function that, given a pointer, returns the number of bytes available in that block). It can be configured at run-time to use any allocator the application developer prefers, as long as it implements those three functions.

Thread List Consolidation. The per-thread lists of addresses are implemented as deques: values are pushed on one end by the owner as allocations occur, and are popped from the other by the consolidating thread. The consolidating thread grabs a global lock before it begins, so deques are only popped by one thread at any time. This makes them single-reader, single-writer data structures, even though the popping thread may be different each time consolidation happens. The implementation is taken from ThreadScan.

Once the thread buffers have been drained by the consolidator, the thread pushes the consolidated buffer onto a list of waiting buffers for the reclamation thread to find. The final delete buffer is created by the reclamation thread as an aggregate of waiting consolidation buffers and leftover addresses from previous reclamation iterations.

This architecture is highly concurrent, in spite of the global lock, when thread buffers are big enough to make consolidation rare. Therefore, contention is low. The thread buffers have a configurable size, but they default to 65,536 (64K) addresses, a number that was picked based on hand-tuning.

Capping Memory. Without an unreferenced memory limit, a process could grow unbounded for data structures with frequent writes if write operations, that might cause fast memory turnover, are exceedingly fast. Forkscan caps the amount of unreferenced memory by limiting the number of unreferenced pointers. Thus, the cap is proportional to the average size of allocations. The limit of unreferenced pointers is enforced by the consolidation system: When a consolidated buffer is pushed onto the waiting list, the consolidator increments a counter. If the counter exceeds the waiting limit, the consolidating thread stalls until the counter is reset by the collection thread when it starts a new iteration. Memory is bounded because no other thread can become the consolidator until the current one relinquishes its role, thereby throttling user threads when memory threatens to grow beyond the (configurable) predetermined limits.

Naturally, a lack of memory can lead to massive throttling, thereby introducing latency. But the user can configure the size of the thread buffers and maximum number of waiting consolidation buffers, trading off memory for latency (as demonstrated in the results).

Thread Handling. In order for Forkscan to function, it must know about all threads that may access the data structures that use `forkscan_retire()`. To obtain this information, Forkscan wraps `main()` and as well as all user calls to `pthread_create()`, storing metadata about the thread ID and stack bounds. At present, it does not allow threads to opt out. This behavior is similar to that of the classic BDW collector [34]. The Forkscan algorithm does not inherently disallow threads from opting out. But such threads would have to be restricted from operating on concurrent data structures or moving pointers to concurrent nodes around in memory, thereby "hiding" them.

Signaling. Signaling is based on `pthread_kill()`, an API originally intended for killing threads, which targets a thread by ID. When the process starts, it registers a signal handler that catches the signal on the targeted thread. A thread that receives a signal will be interrupted unless it is in the midst of a system call, in which case it responds before it returns to user code. With the ID of all of the threads in the process, the collector is able to iterate through the list and force every thread to pause its execution and respond through the signal handler.

The signaling mechanism additionally forces the thread to dump its registers to the stack for the purpose of saving the context. The operating system will use this context to resume the thread after the signal has been handled. However, having preserved its register contents, the forked process can see the register contents each thread had at the time it paused.

Forking. When the collector thread forks, the children need to communicate a potentially large amount of data with the parent about the reference counts they calculate. To make this communication efficient, the delete buffer is allocated on shared pages. Changes to the reference counts in the delete buffer made by child processes are visible to the parent without any explicit communication. Since the reclamation thread in the parent has no work to do, it waits on a pipe for notification that the scan has completed. The children can exit as they finish scanning their regions of memory, atomically incrementing a shared `scanner_completed` counter as they leave. The last child sends a message to the parent through the pipe, waking it up and allowing it to proceed.

Finding Memory to Scan. The first `fork()` generates a scan child that calculates how much memory needs to be scanned by reading from the `/proc/self/maps` file. It keeps track of memory ranges that might contain references to concurrent nodes, excluding regions allocated by Forkscan, itself. The latter exclusions are easy to detect because Forkscan does all of its own internal memory management.

Expedited Scanning. Comparing a range of memory addresses, m, to an arbitrary list of delete buffer addresses, d, is a $O(m \times d)$ problem. Forkscan sorts its delete buffer to make the scan a $O(m \times \log d)$ problem. However, accessing the delete buffer is still slow because it can potentially fill thousands of pages, leading to frequent cache misses.

Performance is improved by creating a minimap of addresses: a subset of addresses from the bigger pool. The minimap is created by striding across the overall delete buffer, a page at a time, and collecting the first address stored on each page. Therefore, each entry in the minimap corresponds to the first entry of each page in the delete buffer. When searching for an address, p, the minimap can be queried for the closest address without going over, q. Since the delete buffer is sorted, the location of q in the minimap identifies the exact page on which p exists, if it is present in the delete buffer.

For 4096-byte pages and 8-byte addresses, this means the minimap represents a 512-fold reduction in space, or the equivalent of 9 steps in a binary search. In general, it also means that the whole minimap fits in cache. The consequence is that a search for a particular address typically misses the L3 cache exactly once. In practice, this optimization reduced Forkscan's overhead to negligible levels in many tests.

Cache-friendly Scanning. In the root finding phase of the scan, potential references are collected and not searched in the delete buffer until a threshold has been reached. Once enough potential references are found, they are sorted, and searched sequentially. A pointer to the last searched location in the delete buffer is retained since the next potential reference is likely to be very close. The next reference can be checked against the rest of that cache line in the delete buffer. This makes binary searches rare during root finding, and keeps accesses mostly sequential.

Scan Parallelization. The amount of memory to be scanned determines the number of siblings the first child will `fork()` to help. In practice, every extra 128 MB warrants another scanner, up to a system maximum of 16. This number was selected based on trials run on three different machines (with very different architectures) that all gave best performance at this number. The subsequent `fork()` calls are cheap, and the processes are lightweight, because they modify almost no memory except for what is in the shared buffer. The memory they scan is treated as read-only, so copy-on-write is never invoked.

Scan children communicate with one another about what addresses they have seen using the shared delete buffer. All manipulation of the reference counts are done with an atomic increment, which is a Read-Modify-Write (RMW) operation. The struct at the head of the delete buffer is shared, itself, and contains the `scanner_completed` counter.

De-allocation. Our experiments show that deallocating memory via a long sequence of `free()` calls is expensive due to the system calls to `madvise()` (controls page release/purge to the Linux OS). Therefore, to avoid contention and latency, nodes which are marked for deletion are not freed immediately. Instead, the protocol pushes the delete buffer onto the back of a list of delete buffers from previous iterations. Threads that want to allocate memory and don't have anything to free query this list and reserve a subrange from the front delete buffer.

Potentially, a thread that only calls `forkscan_malloc()` once could retain a reference to a delete buffer and keep it from being reused, but no thread could monopolize more than one delete buffer. And practically, a thread that mutates a concurrent object once is likely to do so again.

False positives. Nodes from previous iterations that have no outstanding references, but are still waiting to be freed, may contain references to nodes in the current reclamation iteration. In practice, this is a significant source of false positives that leads to loss of scalability. The reclaimer, therefore, creates a list of *dead nodes* from the previous delete buffer to give to the forked children for reference. A child, while it is scanning memory, uses this list and skips scanning anything from a dead node. List creation is performed after all threads have acknowledged the signal, but before the `fork()`, making the protocol slightly more costly.

3.2.4 Forkscan Correctness Properties

Forkscan makes the following set of assumptions.

1. (*No False Negatives*) References to memory blocks in the scanned space are word-aligned, and can be matched to the interior of allocated blocks by comparison. Additionally, Forkscan masks off the low 3 bits of any word it reads when scanning. This covers the common form of "pointer-hiding" used by many data structures, and it means those that overload those bits are discovered.

2. (*No Thread Crashes*) Threads do not crash.

3. (*Bounded Allocation Rate*) There exists a finite bound on the allocation rate of the application.

4. (*No False Positives*) Arbitrary memory words do not match block addresses.

Under these assumptions, Forkscan provides the following guarantee:

Theorem 1. *Forkscan ensures the following.*

1. *Reachable memory blocks cannot be de-allocated.*

2. *Every unreachable memory block is eventually de-allocated.*

3. *There exists a finite bound on the amount of memory employed by the application at any point in time.*

Proof (Sketch). For the proof of the first two statements, consider an arbitrary collection phase, and let T be the time when the `fork()` call completes. A key invariant is that allocated memory nodes which are not referenced (either in thread stacks and registers or in the heap) at time T cannot be referenced at later times in the execution (unless first recycled), as they are currently *unreachable.* Further, we rely on the fact that the child thread resulting from the `fork()` operation receives a consistent *snapshot* of the entire parent process memory at time T. By assumption (1), every reachable node will have a non-zero reference count at the end of the scan. By assumption (2), no unreachable node can have non-zero reference count at the end of the scan. These two properties will imply that no reachable memory blocks can be allocated. Since Forkscan checks for cycles, every unreachable block is eventually de-allocated. The third property follows since we assume no false positives, and that there exists an upper bound on the allocation frequency.

3.2.5 Experimental Results

Setup. Forkscan was tested on an 40-core (4 sockets, 80 threads) Intel Xeon computer at 2.4 GHz with 1TB of RAM running Ubuntu 15.04 with the 3.13.0-57 kernel. Software threads were scheduled by the operating system, though the Linux kernel tended not to migrate threads very often, but instead scheduled threads on the same cores, generally. The data structures that were tested were a lock-free linked list[49, 70], a lock-free hash table from Synchrobench [47], and a lock-based skip list from StackTrack [1].

For comparison, we used versions of these structures that leak memory ("Leaky"), the latest Boehm-Demers-Weiser Garbage Collector 7.4.2 ("BDW-GC") [16], and simulated a Hazard Pointers [71] implementation by burdening reads during list traversals. StackTrack had an actual Hazard Pointer implementation. We compiled BDW-GC with parallel mark and thread local optimizations. The Leaky, Forkscan, and Hazard Pointer tests ran with the JEMalloc 3.6 [37] allocator. BDW-GC uses its own internal allocator.

The linked list was initialized with 1024 nodes and executed with 2048 possible values. Nodes were padded to 176 bytes in order to avoid false sharing and prefetching. This was beneficial for all systems. The skip list was given 12,800,000 nodes and 25,600,000 possible values. Unlike nodes in serial skip lists, which are only as large as they need to be, ours were made 256 bytes with a maximum height of 20. Again, eliminating short nodes was necessary to eliminate false sharing. It is worth noting that at that height, guaranteed $O(lgn)$ access complexity only allows for 1,048,576 nodes, so accesses were slightly more expensive. The total size of the skip list was about 3.1GB. Last, to test a high performance structure, the hash table was given 32,000,000 initial nodes with a range of 64,000,000 possible values. Buckets were implemented using the lock-free linked list, with 32 average list length. The hash table's size was about 5.4GB.

Forkscan was configured for conservative memory usage: each thread had a pool capacity of 16K nodes, and no more than 4 aggregate lists could queue up before Forkscan began throttling threads trying to allocate memory. No tests were run with larger per-thread pools because performance was good with the smaller ones.

Figure 3-7 shows benchmark results on the data structures. Results are averaged over 3 executions of 4 minutes each. The benchmarks were set to perform 20% modify operations (reads and writes), a very heavy workload, to show performance under pressure.

In the linked list case, BDW-GC performed along a similar curve to "Leaky" with

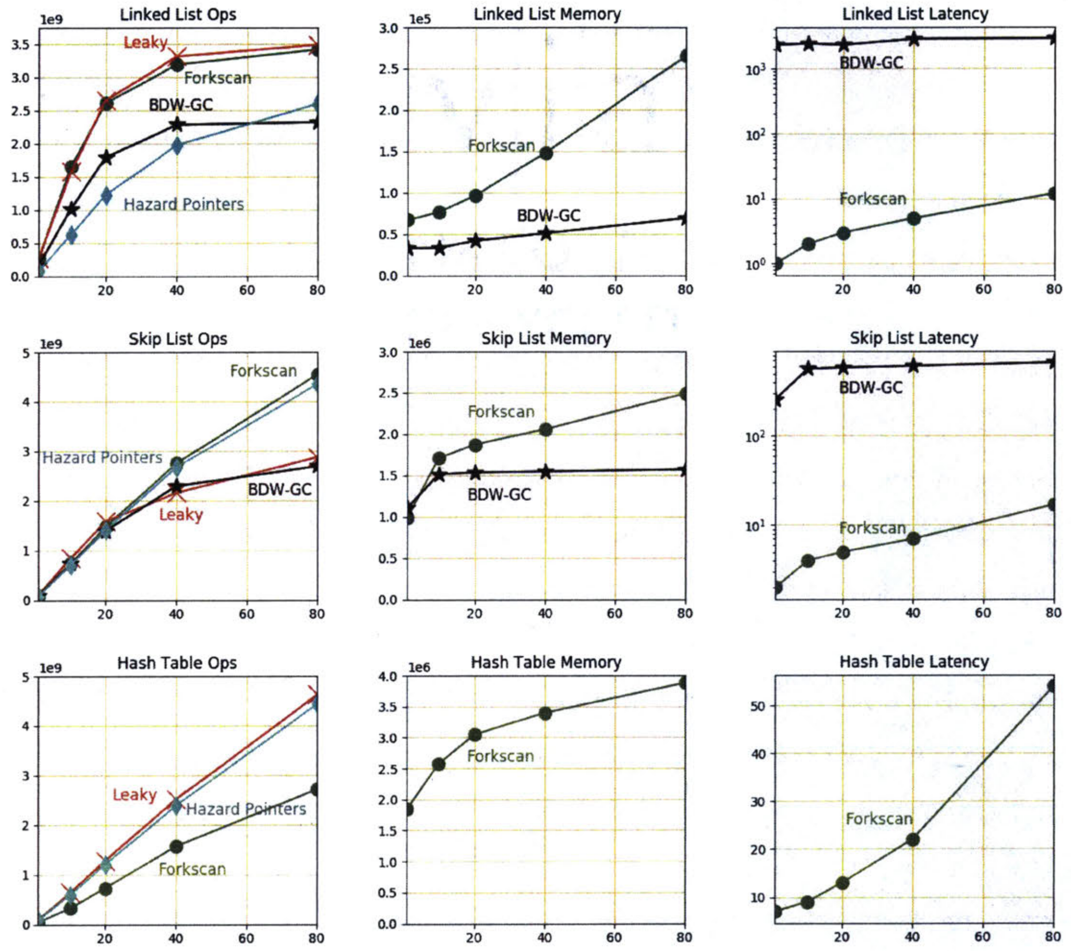


Figure 3-7: Performance results on the linked list, skiplist, and hash table data structures. For each: total operations (threads x ops), memory usage of the application (threads x 4KB pages), and average latency per reclamation iteration (threads x time in ms; logscale for the first two structures, to compare Forkscan with BDW-GC).

visible overheads, and the Hazard Pointer simulation showed the cost of adding writes to every read. However, this structure has slow enough operations that Forkscan's performance overheads were almost unnoticeable. This is also visible in the memory usage graph, which has no elbow in the curve. Memory usage is expected to grow linearly with the number of threads, since each thread has its own pool, and a consolidation buffer size is proportional to the pool size and the number of threads. A linear growth shows that Forkscan did not need to allocate extra consolidation buffers or delete buffers. BDW-GC, on the other hand, has a roughly fixed overhead and is able to track all of its pointers because it owns its allocator. Snapshot latency was especially low for Forkscan, topping out at 12ms on 80 threads, because the whole application used very little memory. BDW-GC's latency was roughly 242x above Forkscan because collection happens inline with the running benchmark. The extremely high latency, in this case, was likely due to the length of the chain.

The skiplist has cheaper operations overall, even though it is over-filled. Additionally, although StackTrack's skiplist takes out locks during add and remove operations, there is not very much contention and the skiplist beats the linked list based on its access time. In this case, Hazard Pointers and Forkscan both outperformed the Leaky implementation. Leaky performed poorly because, on the 40 and 80 core executions, the large size caused it to have poor cache performance. BDW-GC took a hit in about the same place but in this case, it was probably due to excessive scan times on the large data structure. Even though Forkscan performed better than Leaky, it was burdened due to throttling. As we demonstrate below, the throttling latency can be overcome.

Memory usage is again higher for Forkscan than BDW-GC, since Forkscan uses extra memory proportional to the thread count. The elbow in memory usage happens early, at 10 threads, as the cheaper operations caused Forkscan to queue consolidation buffers. It never quite found equilibrium before the queue filled, and throttled user threads. At 80 threads, it reached 60ms, as high as any of Forkscan's trials. However, when compared against the average 6.3 second scan time of BDW-GC, Forkscan's latency was still very low.

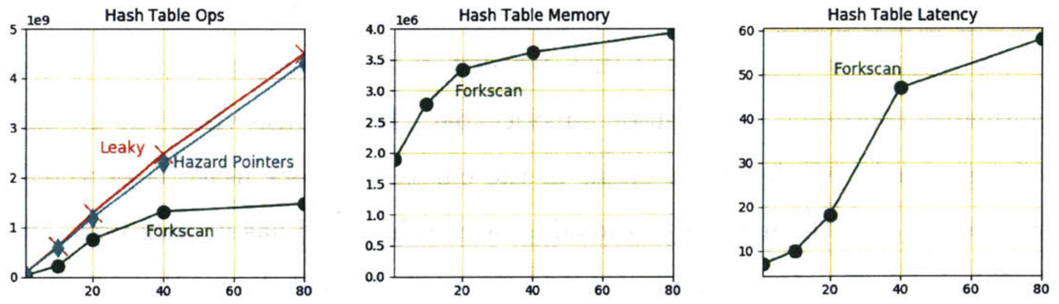


Figure 3-8: Performance results for a hash table with 40% update operations. The axes are the same as above.

Hash tables have cheaper operations than skiplists. In general, accesses are expected to be constant-time operations, so the Hazard Pointer simulation performs very well. From the outset, Forkscan has a difficult time keeping up with it, but continues to scale linearly (albeit, linearly with a small constant multiplier). Again, the overhead can be attributed entirely to throttling of user threads as they perform allocations. The time it takes to perform a snapshot, even on this large data set, however, remains low: reaching a maximum of 54ms on 80 threads. The BDW-GC collector was unstable on this workload, even when provided with lots of extra memory, so it could not be tested.

An additional stress test was run on the hash table to demonstrate the breaking point of Forkscan. Instead of 20% updates, a high value for real world applications, 40% updates was specified. The results are shown in fig. 3-8. As above, read and write operations are all about the same to Hazard Pointers. However, Forkscan does not gain appreciably from doubling the number of threads from 40 to 80. At this point, throttling is significant and almost all of the overhead is attributable to that, as the snapshot time has not increased appreciably over the 20% update trials.

High latency due to throttling may not be any more palatable to C/C++ users than if the memory reclamation system simply stopped the world and did all of its work using the user threads. However, further tests demonstrate that, unlike stopping the world and recruiting the user's threads for memory scanning, latency due to throttling can be reduced where additional memory is available. In the cases above, most overhead was throttling and very little was due to stopping the world to

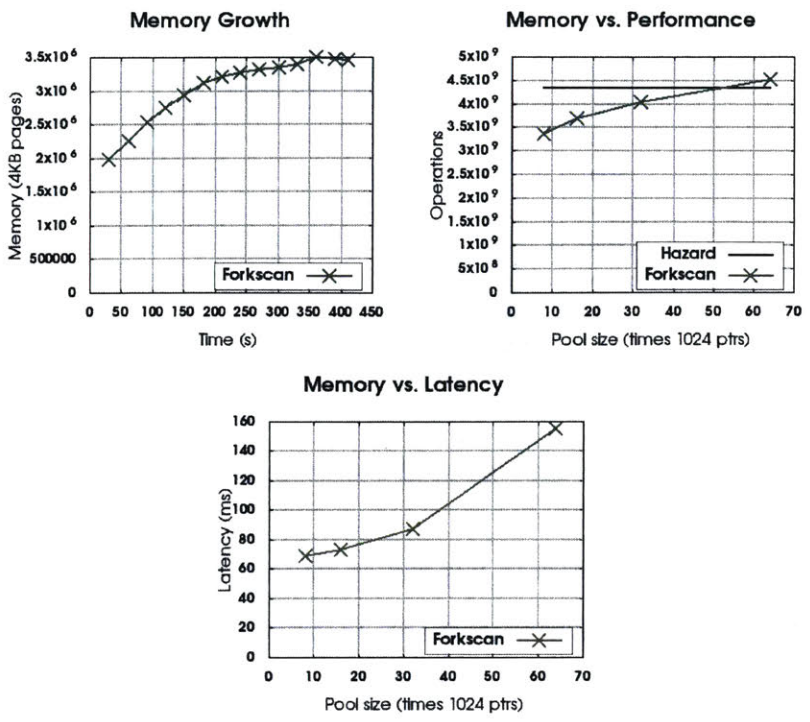


Figure 3-9: Forkscan memory usage and memory/latency vs. performance tradeoffs.

take a snapshot.

Figure 3-9 shows how memory is used by Forkscan and how it impacts performance. The first graph shows the total memory footprint of Forkscan run on the hashtable with 20% updates over a 7 minute execution. Forkscan can queue up to 4 consolidation buffers (configurable) at a time before throttling, and delete buffers are only reused after all freeable nodes are actually freed. These buffers are proportional to the number of threads and individual thread pool size, and the total overhead corresponds to the total number of pointers times the average size of nodes.

The second graph is based on the skiplist run with 80 threads and shows that performance can be bought with more memory, as is typical with other automated reclamation systems. The Hazard Pointers result on 80 threads from fig. 3-7 is shown for comparison. This tradeoff is the mechanism by which the user amortizes the cost of reclamation over the normal cost of performing operations in the application. The third graph, however, shows the snapshot latency over those same executions. Whereas, in garbage collectors, increased memory to improve performance increases

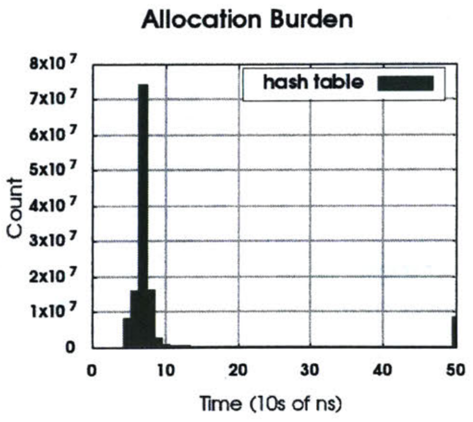


Figure 3-10: Histogram of overhead per allocation.

individual thread latency, Forkscan imposes no significant increase in this metric.

This demonstrates that Forkscan is able to provide comparable performance to manual memory reclamation schemes at a fraction of the latency of traditional automatic reclamation systems. A larger process takes longer to fork than a small one, but the vast bulk of the cost of doing memory reclamation is invisible to user threads, even when Forkscan is configured to use large amounts of memory. The maximum 155ms (at 64K pointers per thread pool) is human noticeable, but it is a short duration compared with conventional automatic systems.

The last point regarding latency of concern to C and C++ programmers is the overhead on the burdened allocation. Since `forkscan_malloc()` attempts to free nodes from previous iterations, the actual allocation is more costly than in a serial execution. The overall amount of work is no more than in a serial application since one `free()` corresponds to one `malloc()` in the underlying allocator in both cases. But a call to `forkscan_malloc()` attempts to free multiple nodes per allocation in order to keep memory low.

The Forkscan library was instrumented to capture the amount of time spent freeing nodes, and the original hash table trial was rerun with 80 threads. Figure 3-10 shows a histogram of the amount of time (in tens of nanoseconds) spent on each allocation. The vast majority of allocations were burdened by no more than 100ns, though there was an extended tail due to differences in operating system scheduling. Calls with overhead of more than half a microsecond were all collected into the last bucket, causing the apparent bump.

The lack of variance during freeing is expected since the nodes have been sorted, and for adjacent pointers in the list, those pointers are likely to have good spacial locality. Therefore, many calls to `free()` should not be much more costly than a single one. Since this experiment is dependent on the implementation of the underlying allocator (JEMalloc, in this case), the shape of the graph is more interesting than the specific numbers.

Real-world application Finally, to demonstrate Forkscan's effectiveness in a real-world application, memcached [39], was modified to create Leaky and Forkscan versions, replacing its default reference-counting. In the altered versions, the builtin slab allocator was removed and replaced with je_malloc for simplicity.

In the Forkscan version, all accesses to individual item reference counters were eliminated, and when an item was unlinked from the structure, the thread that succeeded in unlinking it then retired it. The Leaky version differed only in that the retire call was commented out. Since the memcached implementation was not changed, apart from how memory was managed, Leaky was intended to act as an upper-bound for performance.

To test performance, it was necessary to create a large enough database that many connections would be supported without making contention on individual items a bottleneck. Such a bottleneck would have masked the best-case (for memcached) scenario limiting factor. However, this had to be balanced against the ability to fill the database quickly and force replacements to happen frequently. Therefore, memcached servers were created with 1GB of memory, storing items of 1024 bytes, allowing roughly 1 million individual items.

The memcached servers were configured to run locally, avoiding network overhead and latency. Trials were run using memtier_benchmark [63] with 16 threads for 12 seconds with a set/get ratio of 1:4, and using 40 multi-key gets to inflate the number of requests through a limited number of connections. Each trial, for each version,

Threads	Default	Leaky	Forkscan
1	156532	160715	173726
2	233993	276594	227535
4	306870	280548	314001
10	523586	534209	510895
20	245087	277168	259428
40	193803	198706	200100

Table 3.1: memcached performance in operations/second.

was run 3 times and the average number of operations/second was computed. Trials above 40 threads were not run because of performance degradation for all versions.

Table 3.1 shows the results. Performance was comparable in all cases. The high variance in execution times between trials indicates that other factors are more important to performance than memory reclamation (or lack thereof), as sometimes Leaky was outperformed by Forkscan or the Default reference-counted implementations. A drop-off in operations per second occurred after 10 threads making locks a likely culprit. The amount of time it took to freeze and fork was typically around 5ms or less, and never exceeded 9ms.

These results indicate that Forkscan works in a practical setting, making the code simpler without impeding performance. Moreover, the individual data structure benchmarks, especially the hash table, test Forkscan far more strenuously than memcached. The simplified application code, which no longer needs to count references nor carefully needs to verify correctness, makes Forkscan a valuable alternative for reclaiming memory from concurrent data structures.

3.2.6 Summary

Forkscan shows that it is possible to provide fully scalable conservative memory reclamation for low level languages like C and C++ by exploiting parallelism and tailoring it to take advantage of mechanisms that are highly optimized in modern operating systems. The present library implementation focuses on the Linux operating system [20], but we believe the ideas behind it can be applied to other state-of-the-art operating systems as well.

Performance of Forkscan is competitive with other automatic reclamation systems such as the popular BDW garbage collector. On the other hand, although manual application of certain memory reclamation systems can eliminate unpredictable delays, they are often difficult to apply correctly and impose themselves on the end-programmers who use the data structure. Forkscan takes a meaningful step in the direction of reduced latency imposed on user threads, while maintaining an automated interface. Notably, even in applications with large data sets Forkscan's snapshot causes only brief interruptions.

Our experimental setup was developed for Linux, which offers an efficient copy-on-write mechanism through `fork`. In theory, a similar mechanism can be implemented in Windows via Virtual Memory Functions [73]. An implementation such a copy-on-write mechanism is described and benchmarked in [84], to provide concurrent garbage collection for the D programming language. Although in theory this implies that Forkscan could work on Windows, its implementation would probably be quite complex.

Forkscan's interface, requiring a *retire* call, is a feature designed to improve performance (both in time and memory) and give the programmer control over how memory is handled. Memory that is known to be visible to a single thread can be free'd directly. That memory need never be tracked by Forkscan, saving time and resources. A programmer can simulate a GC-style interface by retiring a pointer as soon as it's allocated – Forkscan even provides an `automalloc()` function as an alternate interface – but one expects that low level programmers prefer the control of the default interface.

Limitations

Conservative Reclamation. Forkscan is conservative, in that memory words which could be pointers to a memory block are automatically treated as references. It shares this limitation with other automatic reclamation systems for C/C++ [34, 85]. In theory, this assumption could prevent memory from being de-allocated, e.g. in the case of a list whose head node has a false reference. A study by Boehm [14] considers this issue in detail, and concludes that, in most practical scenarios, the space overhead of conservatism is bounded by an additive constant. This analysis generalizes to Forkscan.

We also assume that references are word-aligned, and that the programmer does not obfuscate references to memory blocks. This assumption is also standard for conservative reclamation. Forkscan mitigates this, slightly, by masking the low 3 bits when it reads a word, since some data structures overload those bits to save space. There are, of course, many ways to hide pointers that are not detectable in a general sense, but this is the most common and is easy to catch.

Multiple Threads Retiring a Single Node. At present, retiring a node multiple times from different threads might cause the same node to be tracked (and not found) in two subsequent iterations, causing a double-free. For this reason, adapting Forkscan to support this usage model would require fundamental design changes. However, in each of the microbenchmarks, as well as in memcached, finding the right place in the code to call `forkscan_retire()` was obvious: the thread that successfully marked the node removed was responsible for retiring it. A quick look at a variety of concurrent data structure designs in Herlihy and Shavit [52] shows that this is a common pattern. Therefore, we think that finding this place is generally easy, so there is no need to support multiple threads retiring the same node. That said, this is a possible avenue of future work if many concurrent data structures require or are simplified by that interface.

Space Usage. Currently, Forkscan introduces a constant multiplicative overhead by storing a pointer to each retired object. This can be eliminated by directly utilizing the node information stored in the allocator, which subsumes these lists, by merging Forkscan with the allocator, itself. Many C/C++ programmers choose an allocator that suits their specific performance needs, however, and marrying Forkscan to its allocator makes it less general purpose. For portability, Forkscan is designed to work with any allocator.

Cases of Non-trivial Latency. After threads are signalled, they busy-wait until the fork is complete and they are allowed to continue. There is a cost associated with responding to the OS signal, as well as time spent waiting. Altogether, this does not impose significant latency, but the latency increases with thread count. On future architectures, it may begin to impose an unacceptable burden. The latency could be reduced if the `fork()` procedure, itself, had knowledge of the Forkscan algorithm. Instead of signalling the threads, Forkscan could simply fork the process, allowing each thread to continue until it hit a copy-on-write page, at which point the OS would know to stall it until the fork is complete. The OS also has access to the contents of the registers, allowing it to spill them into a special location known to Forkscan. Any thread that has not performed a write when the fork is complete can be signalled by the OS, and its register state recovered. This optimization would allow Forkscan to avoid signalling, and threads could run until the last possible moment.

3.3 Discussion

With the development of Forkscan, there exists a library that meets the criteria outlined in Chapter 2: automatic, cheap reads, low latency, and scalable. Latency and scalability require a short explanation, however. `fork()`'s cost is proportional to the size of the process since pages need to be marked copy-on-write, and the cost of flushing the page table is borne by the whole process. Moreover, as discussed explicitly, the more threads in the process, the longer synchronization takes. In practice, these add up to user-detectable latency (>10ms) only in extreme circumstances. But the caveat exists.

Scalability, likewise, was tested under write-heavy workloads with many threads, and caused Forkscan to throttle. In comparison with, e.g., Epoch-based techniques, this may seem heavy weight and limited, but recall that the microbenchmarks represent the most extreme circumstances. In the case of memcached, the overhead versus even the Leaky implementation was undetectable and lost in the noise.

Although these limitations warrant further investigation, the pause is bounded

Algorithm 5 Augmented Remove Method From a Concurrent Linked List

```
 1: function LIST_REMOVE(list, x)
 2:     while true do
 3:         { pred, curr } ← WINDOW_FIND(list.head, x)
 4:         if curr.x ≠ x then
 5:             return false
 6:         end if
 7:         succ ← GET_UNMARKED_REF(curr.next)
 8:         marked_succ ← GET_MARKED_REF(succ)
 9:         if false = CAS(curr.next, succ, marked_succ) then
10:             continue
11:         end if
12:         CAS(pred.next, curr, succ)
13:         retire curr
14:         return true
15:     end while
16: end function
```

and short, and the scalability is high in real world applications. I argue these aren't an impediment, and Forkscan represents a meaningful starting point for practical automatic concurrent memory reclamation in low level programming languages. Additionally, what has emerged is an interface that provides automatic reclamation without imposing compiler-related or contractual burdens on programmers beyond what's required by conservative GC but without the overheads.

Independent of whether the future is Forkscan or something else, the *retire* interface has endurance power because of its simplicity and generally applicability in concurrent data structure methods. In Algorithm 5, the augmented removal method for Harris's common concurrent linked list[49] is shown. Literally, the only change is on line 13: `retire` the node that was removed. If two threads try to perform a logical remove the same node, only one `CAS` will succeed on line 9. The one that fails will try again, and the one that succeeds will try to swing the predecessor's pointer (whether it succeeds or fails is irrelevant since other threads traversing the list will try to help). This thread can safely `retire` the node without worrying about a double-retire because no other thread will return `true` (success) for this node's removal.

Indeed, in nearly every data structure that's been modified using this interface, the change has either been this one-liner or a test of the return value of a `CAS` (that wasn't tested in the original code) plus the `retire` statement.[5] I argue that this is transformative for the way concurrent data structures are able to be expressed in the literature since a researcher need only express where a thread can uniquely identify when a node is removed.

[5]More examples are provided in the chapter on DEF, since they have actually been implemented in that language.

Chapter 4

DEF Overview

4.1 Introduction

DEF is designed to fill a void in programming languages – the intersection of low level and scalable. This chapter is primarily intended to demonstrate how it fills that void, centering on the implementation of concurrent data structures. That said, a general-purpose language is a great beast and one can't avoid explaining and justifying its various features and design details. It would be inappropriate to hand-wave this aspect, so justification will be measured against the following criteria:

- **Close to the machine by default:** DEF must not impose hidden overheads for basic operations – if you can do it in C, you should be able to do it in DEF. Differences in performance for equivalent code should be explainable in terms of back-end compiler optimizations. This is, of course, a big part of the motivation for the language.

- **Principle of least surprise for C programmers:** DEF has many similarities to C so that it's easy for C programmers to learn. Where syntax is similar, semantics must be the same unless there's strong justification for different semantics (or the difference is detected during compilation and a warning or error is issued by the compiler).

```
/* Compute the n'th Fibonacci value.  Slowly.
 */
def fib (n i32) -> i32
begin
    if n < 2 then // base case.
        return n;
    fi
    var a i32 = fib(n - 1);
    var b i32 = fib(n - 2);
    return a + b;
end
```

Figure 4-1: DEF Fibonacci example code.

- **ABI/API bi-directional compatibility with C:** Types have equivalents in C and module ABIs and APIs are compatible except for DEF API features unavailable to C (e.g., tuples). There are massive bodies of legacy C/C++ code, and DEF should integrate as painlessly and transparently as possible into existing applications.

- **Non-invasive high level features:** Where DEF has features that require runtimes, those runtimes should impose no detectable performance overheads unless and only insofar as they are used by the application. This excludes, for example, GC, but permits Forkscan since the latter never runs unless objects are being retired. Both as its own principle and in light of the aforementioned prevalence of legacy C code, programmers shouldn't have to think twice about using or incorporating DEF on account of a performance hit due to runtime libraries.

4.2 Basic Syntax

4.2.1 Fibonacci

Fig. 4-1 shows the recursive implementation of Fibonacci in DEF. This is generally intuitive to a C programmer with the most striking difference being the lack of curly braces for scoping. Keywords are used in their place for readability: the perennial

```
int (*foo ())();
```

```
decl foo () -> () -> i32;
```

Figure 4-2: C (above) and DEF (below) declarations of a function returning a function pointer that returns an integer. `decl` is used for function declarations in DEF.

problem of, e.g., inserting code at the end of a scope when there are multiple closing curlies and having to figure out which one applies to the scope the programmer cares about, is avoided. Moreover, curly braces can now be reclaimed and repurposed for another use.[1]

Another apparent difference is the function profile: the `def` keyword is used to define a function, and the return value has been moved to the right-hand-side using arrow notation, as in ML or Go. This syntax has the benefit that returning a complex type doesn't lead to indecipherable code. Consider the function declarations in fig. 4-2 in which a simple function pointer is returned from a function. In general, DEF is a left-to-right language, as this makes code more comprehensible to humans.

The third notable difference is that the variables `a` and `b` are declared with the `var` keyword, and their type is declared after the variable name (in keeping with left-to-rightness). The types of these variables can be omitted and inferred in this example – they're included for demonstrative purposes.

The last obvious difference is the replacement of `int` with `i32`. Integers are denoted by `i` or `u` (signed or unsigned, respectively), and a bit width. Floating point numbers are `f` with the bit width. Permitted widths are based on the underlying hardware.

There is a hidden difference, versus C, that is detected at compile time (slightly surprising, but in keeping with the *principle of least surprise* as specified above): `fib` is local to the module in which it was declared. In order for a symbol or type to be visible to other modules, the keyword `export` is used. This is the reverse of C (which uses the `static` keyword to declare a symbol local to the module) and was changed for the following practical reasons:

[1]For completeness, if-statements use the `elif` syntax (as opposed to `else if`) for parsing reasons.

```
def remainders (n i32) -> { i32, i32 }
begin
    return { n % 3, n % 4 };
end

def zeroes (n i32) -> i32
begin
    switch remainders(n) with
    xcase { 0, 0 }:
        return 2;
    xcase { 0, _ }:
    ocase { _, 0 }:
        return 1;
    xcase _:
        return 0;
    esac
end
```

Figure 4-3: DEF toy example of a tuple and a switch statement, including wildcards.

1. Default global visibility tends to pollute the symbol table unnecessarily; doubly in the absence of namespaces (which C lacks).

2. Local functions, in principle, can sometimes be optimized in ways that exported ones can't.

3. Being explicit about what's exported makes automatic generation of headers and interface files easier.

Reason 1 is non-negotiable: DEF can't have namespaces and maintain API compatibility with C. Reason 2 is something that expert C programmers know, and they're always careful about visibility, but less experienced C programmers potentially lose optimizations (such as knowledge about pointer aliasing from the complete set of callsites). Therefore, local visibility provides the best optimization opportunities by default. Reason 3 is discussed along with the `defghi` utility.

4.2.2 Tuples and Switch Statements

Fig. 4-3's `remainders` function shows a tuple. The function takes in an integer and returns a tuple of two integers: the remainders of the input divided by three and four, respectively. The return statement is able to create the tuple on the spot or return a variable that has the correct tuple type. Tuples in DEF are nothing more than anonymous structs; they have the same allocation and copying properties as native values, and to keep one alive on the heap it must be declared as a pointer. This syntax looks like static array definitions in C,[2] which might appear to violate the principle of least surprise, but a tuple of n elements of type t has identical memory layout semantics as a static array of n elements of type t. Also, any treatment of the tuple as an array will be caught by the compiler.

The `zeroes` function calls `remainders` and returns a count of the number of zeroes in the tuple using a switch statement. For DEF, switch statements are for more general pattern matching than in C. In this code, the switch examines the contents of the tuple and uses the underscore, as a wildcard, to match any value. Patterns are matched from left to right, and the first successful case is the one that's executed. A difference from C is that no `break` is used to exit a switch statement. Instead, when a case completes, the next kind of case indicates permeability into its block: `ocase` means that the case falls through, and `xcase` means that it does not. Note that the `ocase` on line 12 means that the previous case will fall through, since both cases have exactly one zero. The last case, in this example, is the default one; there is no `default` keyword because matching a wildcard catches anything that wasn't caught by a previous case.

Switch statements will also do array comparisons, including strings. Related sets of string comparisons are extremely common, particularly when reading input, and leveraging switch syntax improves readability. Note that switch statements in C can't match any value other than a primitive integer – a category that excludes pointers – so there are no conflicting expectations on the part of C programmers.

Last, matching a tuple is one thing, but accessing elements from a tuple has no analog in C. Fig. 4-4 shows the ways elements of a tuple can be read. Given the `remainders` function from the previous example, the values can be unpacked dynamically (lines 3 and 6), or individual elements can be selected with a zero-

[2]Static arrays in DEF use this syntax except with square brackets instead of curly.

```
1  // Dynamic unpacking with existing variables.
2  var a, b i32;
3  { a, b } = remainders(10);
4
5  // Dynamic unpacking with new variables.
6  var { c, d } = remainders(20);
7
8  // Accessing variables from an existing tuple.
9  var tuple = remainders(30);
10 var e = tuple.{0};
11 var f = tuple.{1};
12 tuple.{2} = -1; // out of bounds.
```

Figure 4-4: Ways in which elements of a tuple can be accessed.

```
1  typedef string = *char;
2  typedef complex_t = { c_real f64, c_imaginary f64 };
3  typedef state_t = enum
4      | STATE_1
5      | STATE_2
6      | STATE_3
7      ;
```

Figure 4-5: Examples of defining named types.

indexed syntax (lines 10 and 11). Accessing individual elements is designed to look like selecting a member from a struct for readability. Line 12 shows trying to select a field that doesn't exist, but since the tuple layout is known to the compiler it will issue an error.

4.2.3 Types

Defining a named type uses the `typedef` keyword, as in fig. 4-5. A `string` type is defined (line 1) that shows the left-to-rightness of types: a pointer to a char. A struct (line 2) is simply a tuple with the field names specified. And an enum (lines 3-7) is expressed as a set of alternatives, like in ML.

Accessing members of a struct looks like it does in C or Java. There are no arrows when selecting through a pointer, however, in favor of the dot notation. The rationale for unifying the two is that the compiler knows whether the object is a pointer or not and can just do the right thing; two different operators is unnecessary and frees up

```
1 var a [10]*i32;        // compare to C: int *(a[10]);
2 var b *[10]i32;        // compare to C: int (*b)[10];
3 var c * volatile i32;  // compare to C: int volatile *c;
```

Figure 4-6: Types in DEF along their corresponding C code.

the arrow for future use.

Expressing types, generally, is designed to read left to right without the use of parentheses for associativity. Fig. 4-6 shows a side-by-side comparison of identical types between DEF and C[3]: Lines 1 and 2 show a distinction that, in C, is difficult for a human to parse. DEF, line 1, reads as, "variable `a` of an array of ten pointers to integers." Line 2 is, "variable `b` of a pointer to an array of ten integers."

In C, by contrast, the trick is to find the variable name, read right, then read left. Line 1, therefore, requires no parenthases, but representing line 2 requires them. And, of course, in C it's trivial to generate practical types that become arbitrarily complex requiring more parenthases, or where a programmer will tend to insert them when unneeded simply to be explicit (either for their own comprehension or subsequent readers').

Line 3 is included to show the difference in the usage of `volatile`, which has the same semantics in the two languages. The keyword could, in C, be moved to the left side of `int` and mean the same thing, but the recommended order is shown. Again, reading left to right in DEF makes it clearer which part is being specified as a volatile type. It's easy to distinguish the volatility of the integer versus the pointer to it.

Last, note that DEF arrays (as in C) carry no extra data. They represent primitive semantic structure around pointers, but they don't carry around length or stride. There can be no hidden bounds-checking by design.

[3]The `int` type in C is not required to be 32 bits, but it is on all major platforms. For the purposes of this document, `int` will be treated as equivalent to `i32` in DEF even though they would not be on some systems.

```
decl printf (*char, ...) -> i32;  // get printf from the CRT.
```

Figure 4-7: Example of making `printf` available to programmers within a module without including external files.

4.2.4 Summary

Not only is it important to justify syntactic choices, particularly where DEF is different from C, but to recognize that the primitives are similarly close to the machine. In principle, limiting oneself to these low level features is the same as using a "better C" in that there is improved readability and some conveniences not available in C. Writing an application or module in DEF requires no sacrifice in performance or space usage.

Of course, a "better C" would hardly be a substantial basis for research, but establishing that DEF is as capable as C by virtue of having equivalent functionality for performance engineering is a prerequisite to arguing that one can (and should) write concurrent data structures for C in DEF. DEF satisfies the design goal of "close to the machine by default." Moreover, with the type definitions discussed above, it's clear that there is ABI compatibility with C. Types have direct correspondence between the two languages, so no transformations have to be performed to call one or the other language.

The remaining question, for compatibility, is the nature of sharing APIs.

4.3 C API Compatibility

4.3.1 From C to DEF

It's established that DEF has C ABI compatibility, and fig. 4-7 shows how to make a function (in this case, from the C runtime library) available to a DEF module. Using this code, a programmer can print from DEF functions just as they would in C. Likewise, defining any type available to a C module is a trivial conversion from C syntax to DEF. However, this is nothing more than a coating of the ABI. For true

```
import "stdio.h";
```

Figure 4-8: DEF can import C header files directly.

API compatibility, the DEF compiler understands C code and can import C header files directly.

Fig. 4-8 shows that C headers can be read using the `import` keyword. This is the same syntax used for importing DEF interface (`.defi`) files. DEF will look for headers in the same places as the C compiler, and knows about textttstdio.h and other system headers. This is implemented using the Clang[79] frontend, which reads the C code and parses it into its abstract syntax tree, which is then converted to the underlying representation used by the DEF compiler.

At the time of writing this thesis, DEF understands C types, function declarations, and global variables – including those represented in macros expanded by Clang – but not the macros, themselves, or inline functions defined in the header file. This means that anything expanded within the header file is available to a DEF program that imports it, but using C macros in DEF and inline functions from C are not. They are read by Clang but not transformed into an intelligible format for DEF, and are therefore ignored.

4.3.2 From DEF to C

In the other direction, since it is clear that DEF has incompatible syntax versus C, there can be no direct means of including DEF interface files in C source files. However, because of ABI compatibility, most global types, functions, and objects can be translated directly into C, programmatically. Therefore, the same means of generating DEF interface files (`.defi`) from DEF source files (`.def`) can be used to generate C header files.

A special utility, `defghi` (**DEF** **G**enerate **H**eaders and **I**nterface files), accompanies the compiler. Any exported type, function, or object will appear in the header file unless it involves an unnamed tuple, since C doesn't have them.[4]

[4]It is technically possible to generate, e.g., functions that return tuples in C header files using

```
export
typedef complex_t = { real f64, imaginary f64 };

/** Given a complex number, return the real component.
 */
export
def project_real (n *complex_t) -> f64
begin
    return n.real;
end

/** Given a complex number, return the imaginary component.
 */
export
def project_imaginary (n *complex_t) -> f64
begin
    return n.imaginary;
end
```

```
/* complex.h:
 * THIS FILE WAS AUTOMATICALLY GENERATED.  DO NOT MODIFY IT OR YOUR
 * CHANGES MAY GET CLOBBERED.
 */

#pragma once
#ifdef __cplusplus
extern "C" {
#endif

typedef struct complex_t complex_t;

struct complex_t
{
  double real;
  double imaginary;
};

/** Given a complex number, return the real component.
 */
double project_real (complex_t *n);

/** Given a complex number, return the imaginary component.
 */
double project_imaginary (complex_t *n);

#ifdef __cplusplus
} // extern "C"
#endif
```

Figure 4-9: Listing for a complex numbers package. Above: The DEF source file. Below: The automatically generated C header file.

```
export opaque
typedef complex_t = { real f64, imaginary f64 };
```

```
/* complex.h:
 * THIS FILE WAS AUTOMATICALLY GENERATED.  DO NOT MODIFY IT OR YOUR
 * CHANGES MAY GET CLOBBERED.
 */

#pragma once
#ifdef __cplusplus
extern "C" {
#endif

typedef struct complex_t complex_t;

/** Given a complex number, return the real component.
 */
double project_real (complex_t *n);

/** Given a complex number, return the imaginary component.
 */
double project_imaginary (complex_t *n);

#ifdef __cplusplus
} // extern "C"
#endif
```

Figure 4-10: Same example as above, but using the `opaque` keyword. The original DEF file is truncated for brevity.

The code in fig. 4-9 provides a trivial example of code generated by `defghi`. Lines 7-9 is a standard C-ism for making C header files compatible with C++. Line 11 declares the `complex_t` struct, and 13-17 define it. The function declarations and their documentation are emitted below the types.

In some cases the author of a module would want to keep a type opaque to other modules. To accomplish this, use the `opaque` keyword when exporting it, as in fig. 4-10.

The `defghi` utility has the added benefit of maintaining API/module compatibility and keeping documentation up to date, since headers and interface files can be generated as part of the build process. Emphasis is placed on not duplicating code

name mangling, as C++ has. However, using them is no more simple than defining structs in C, and since a mangled name isn't especially handsome from a readability standpoint, I haven't found an application where this functionality is desired.

```
1 var node = new node;
2
3 // Do work.
4
5 if address_may_be_visible then
6     retire node;
7 else
8     delete node;
9 fi
```

Figure 4-11: Trivial example of using `retire` vs. `delete`.

unnecessarily; as a former colleague of mine once said, "If the same code is in two places, it's wrong in at least one."

4.4 Support for Concurrency

4.4.1 Retire

DEF has native memory management commands: `new`, `delete`, and `retire`. "Native," in this context, requires explanation since C doesn't have native memory management and the question arises of how use of the keywords integrates with C modules. They are implemented using a modified version of Forkscan, which invokes the default allocator unless explicitly configured otherwise. Therefore, the keywords, themselves, can be configured to use custom allocators. The only limitation is that there must be a `malloc_usable_size` function that takes an object and returns the amount of space it occupies.

Fig. 4-11 shows a simple example. Line 1 shows the allocation of the pointer to a node. As in C++, `new` takes a type and returns a pointer to it.[5] The divergence from C++ is in the alternatives provided in lines 5-9: If the programmer has published the address of `node` (such as storing it in a concurrent data structure), and can't be sure whether another thread is looking at it, the address needs to be retired. Otherwise, if it's known to be safe to free the memory, `delete` may be used instead.

[5]More complex examples are possible, including inline initializers, array allocations, etc., but they're for convenience and don't require special treatment in this document. Some of this is used in sample concurrent data structures in the next chapter and is discussed there.

```
a atomic += b;        // C: atomic_fetch_add(&a, b) + b;
a atomic -= b;        // C: atomic_fetch_sub(&a, b) - b;
a atomic /= b;        // No built-in C support.
```

Figure 4-12: DEF atomic operations and their C counterparts.

The `address_may_be_visible` variable is for demonstration in this example and not part of the language.

As mentioned, Forkscan was modified to bring it in line with the necessary DEF design goals. The primary change was to burden `retire` instead of `new` (implemented by `forkscan_retire` and `forkscan_malloc`, respectively). In anticipation of producer-consumer model applications in which C code is the producer and DEF is the consumer, the burden of freeing memory couldn't be placed on `new` or it might never get freed at all, since DEF never calls it.

Justification for this change is slightly subtle: It's reasonable to think the shift of burden doesn't impede performance of an application. The original burden was generally invisible because likelihood was high that memory that had just been free'd would be hot in the cache, and was therefore likely to be returned. Burdening `retire` is similar in that if retiring and allocating memory are done in close proximity, the memory is probably in the cache, and if they aren't, then the cost of bringing an evicted line back into the cache is assessed infrequently. Indeed, in various microbenchmarks any difference in performance was unmeasurable.

Last, it should be noted that no known technique for implementing `retire` on Windows has been documented. This stands out as the primary limiting factor in porting DEF to Windows since there is no native `fork`.

4.4.2 Atomic Operations

Although compiler built-ins for atomic RMW operations exist in DEF, just as in C (as of C11[25]), additional syntax is provided for readability and future integration with other features. Individual operations can be marked `atomic`, indicating sequentially consistent semantics.

```
int old;
do {
    old = a;
} while (atomic_compare_exchange_strong(&a, old, old / b));
```

```
    int tmp = b;
try_again:
    if (_xbegin() == _XBEGIN_STARTED) {
        a /= tmp;
        _xend();
    else goto try_again;
```

Figure 4-13: Two possible implementations of the atomic division operation as coded in C. Above: the C Atomics Library solution, below: a transactional approach using Intel RTM.

Fig. 4-12 shows three examples of DEF operations marked `atomic` and their C semantic equivalents. Note that an atomic operation is a description of the desired semantics and not an intrinsic denoting a specific instruction. The difference is that the latter is guaranteed to generate the RMW instruction specified, while the former describes the behavior and will use the RMW instruction to implement it if one exists. Specific instructions may be accessed through intrinsics in both languages, but C recommends the use of the Atomics Library for portability, and DEF recommends the `atomic` keyword.

Lines 1 and 2 show the annotation of the += and -= operations, respectively. The equivalent C code uses a library function and then adds or subtracts the value of `b` from the result. Presumably, the C function was designed to mimic the popular fetch-and-add instruction semantics which returns the old (dereferenced) value of the first argument. DEF's `atomic` annotation is designed to fit with the semantics of the non-atomic operation which returns the new value. It's a subtle distinction, but it makes the subexpression transition from non-atomic to atomic intuitive by not changing any of the other semantics.

Line 3 has no corresponding atomic C code. The operation, itself, isn't supported as a single RMW instruction on any major architecture.[54, 6, 24] Nevertheless, any assignment operator can receive the `atomic` annotation to provide the semantics of that operation performed atomically.

Possible implementations in C are in fig. 4-13. There are two obvious concerns regarding the DEF implementation: 1. a programmer doesn't necessarily know they aren't getting a native instruction out of the atomic code, and 2. DEF hides the lack of foward progress guarantee in this and other architecturally non-native implementations.

To the first concern, DEF uses native RMW instructions that correspond closely to the operations specified where possible. If an instruction set has the given RMW instruction, DEF will use it, so the programmer need not worry they are getting less than what the hardware provides. On the other hand, HPC programmers tend to know the instruction sets they're using and aren't likely to be surprised that, e.g., /= has no locked equivalent on x86. If greater specificity is desired, programmers can use intrinsics and be explicit in the way fig. 4-13 is explicit in C. But DEF provides a readable short-hand that covers the 90% case.

Regarding the loss of a forward progress guarantee, it *is* hidden by the DEF syntax and DEF depends on programmers to know what operations are backed by RMW instructions. This decision is justified by the fact that C's Atomic Library has the same limitations on hypothetical architectures that lack certain native atomics. There's a degree of abstraction in both languages that hides the progress property in a way that compiler intrinsics don't. However, on the reverse side, if some future architecture supported a native /= RMW operation, existing C code would have to be rewritten to take advantage of it, even if that code already used the Atomics Library. DEF code would require nothing more than a recompile.

4.4.3 Atomic Blocks

An atomic block is one in which all modifications to memory happen atomically with respect to other threads executing atomic blocks. A thread, while in a block, looking at the memory being modified by another thread should either see the full change or none of it: thus, atomic. The simplest implementation of this is to take a global mutex at the beginning of the block and released at the end, but this doesn't scale because threads tend to queue on the lock. Practically, this is akin to the naive

```
atomic begin
    // Perform operation.
end
```

```
atomic begin
    // Perform operation.
xfail TF_SPURIOUS:
ofail TF_OVERFLOW:
    // Reduce the size of the transaction.
xfail _:
    // Catch all:
    // Do some error handling before trying again.
    // ... or "goto" out of the block.
end
```

Figure 4-14: Basic atomic block syntax. Above: trivial atomic block executes until it succeeds, below: transaction with abort condition detection and error handling.

synchronization used by Java since lots of threads operating on a concurrent object all use the same object or class lock.

Alternatively, a transaction-based implementation causes contention only when real cache contention exists in a data structure or algorithm. Hardware transactional memory (HTM) is increasingly common in commercial microprocessors and threads that doesn't access the same cache lines don't obstruct one another. The drawback is that, as implemented by Intel, hardware transactions have no forward progress guarantee. Locks have that property.

By default, atomic blocks in DEF use Hybrid transactional memory (HyTM) which has an HTM fast path and a software slow path. A transaction that aborts some constant number of times falls back onto the slow path that's guaranteed to complete. This is a well studied problem.[56, 31, 19, 30, 29, 65]

A trivial DEF example is presented in fig. 4-14. The atomic block is simply a block with the `atomic` annotation, just as an atomic operation has. The second listing in the figure has code to match failure conditions before trying again. The abort value can be checked against a set of possible causes represented by an enum provided in an interface file. As with the switch statement syntax, pattern matching has a fallthrough (`ofail`) and a non-fallthrough (`xfail`) alternative.

Fig. 4-15 shows the same code written manually by a programmer using intrinsics

```
do_transaction:
    var rtm_status = __builtin_xbegin();
    if rtm_status == _XBEGIN_STARTED then
        // Perform operation.
        __builtin_xend();
    else
        if rtm_status | _XABORT_RETRY
           || rtm_status | _XABORT_CAPACITY then
            // Reduce the size of the transaction.
        else
            // Catch all:
            // Do some error handling before trying again.
            // ... or "goto" out of the block.
        fi
        goto do_transaction;
    fi
```

Figure 4-15: Equivalent explicit code from the bottom example in fig. 4-14 as implemented for Intel RTM.

for Intel's RTM. Clearly, it's possible to control the exact code path, as any low level programming language requires, but the atomic block syntax catches the bulk of how transactions tend to be used and presents it in a far more readable way.[6] This is the HTM interpretation of the atomic block (which can be activated with a compiler switch), though by default the compiler emits HyTM.

At present, DEF uses almost the simplest possible HyTM implementation: read a global mutex value at the end of the fast path and abort if it's being held. In the slow path, try to acquire the mutex to perform the operation. If that fails, wait until the mutex is released and, instead of trying to acquire it again, go back to the HTM fast path. Not waiting to acquire the lock prevents threads that might otherwise be able to complete their transactions in the hardware from serializing with each other unnecessarily.

In a future implementation, DEF is likely to employ RHNoRec[65] instead of using a lock. The code for the present implementation is complex, but requires only a single path through the user code. Code for RHNoRec (and other non-trivial HyTM implementations) requires multiple code paths making manual maintenance and even

[6]For completeness, returning from a function out of an atomic block, or otherwise leaving one, implicitly commits the transaction.

```
1  export
2  def fib (n i32) -> i32
3  begin
4      if n < 2 then
5          return n;
6      fi
7      var a = spawn fib(n - 1);
8      var b = fib(n - 2);
9      sync;
10     return a + b;
11 end
```

Figure 4-16: Fibonacci code in DEF.

value	fib(value)
0	0
1	1
n	$fib(n-1) + fib(n-2)$

Table 4.1: The recurrence for calculating values in the fibonacci sequence.

minor changes a logistical nightmare. By contrast, the atomic block is simple and correct. User code looks like any other and is as easy or difficult to modify as non-atomic code.

4.5 Support for Parallelism

DEF is an LLVM-based compiler[58] that uses the Tapir branch created by Schardl, Moses, and Leiserson.[83] Tapir adds LLVM instructions for fork-join parallelism using Cilk or OpenMP. DEF uses Cilk, but if users want an OpenMP option for compatibility, Tapir makes creation of the option easy.

A parallel recursive Fibonacci implementation is given in fig. 4-16. `fib` takes n as a parameter and returns the n'th value in the fibonacci sequence as implementing the recurrence in table 4.1. The base case is given in lines 4-6, and ignoring the `spawn` and `sync` the rest of the code is simply a recursive implementation of the computation. In fact, this is precisely what happens in the single-thread execution of the parallel algorithm in Cilk; everything is executed in the serial order as though the parallelism keywords weren't present.

```
1 pfor var i = 0; i < len; ++i do
2     histogram[data[i]] atomic += 1;
3 od
```

Figure 4-17: Histogram example using a parallel for loop.

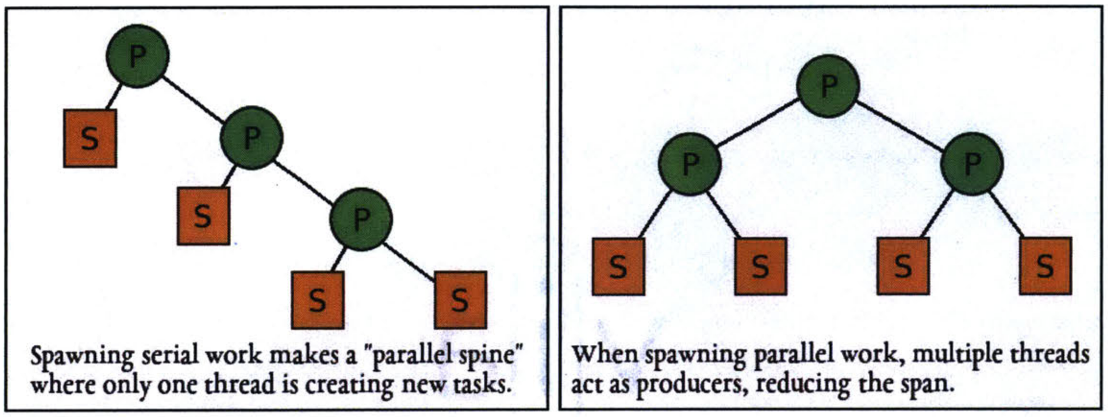

P
S
P
S
P
S
S
Spawning serial work makes a "parallel spine" where only one thread is creating new tasks.
P
P
P
S
S
S
S
When spawning parallel work, multiple threads act as producers, reducing the span.

Figure 4-18: Two ways of dividing a parallel loop. P-nodes represent parallel work, and S-nodes represent serial work. Left: spawn individual iterations, right: divide-and-conquer.

In the multicore execution, however, the `spawn` expresses that the spawned function may execute in parallel with its continuation. `sync` means that all of the parallelism in a function must complete before any code after is able to begin. From this, it's clear that no `spawn` is required on line 8 because there's no code in its continuation before the `sync`.

The other parallel construct is `pfor`, as shown in fig. 4-17. `pfor` is syntactically like `for`, except that iterations are allowed to execute in parallel. The similarity of expressions to C/C++ is apparent on line 2, where everything except for the `atomic` keyword transfers directly. `atomic` avoids the race condition on the element in the histogram array.

Tapir will compute the number of iterations in the loop, if it can, and use recursive divide-and-conquer to maximize parallelism (for most code). Fig. 4-18 shows the two methods. In the first case, when the number of iterations isn't known to the compiler, the loop behaves like a loop that spawns each iteration. Parallelism is maximized, however, when a worker thread can steal about half of the work. Divide-and-conquer

```
1 @[define a "hello world"]
2 @[define b [parse-stmts printf(@a);]]
3
4 def hello_twice () -> void
5 begin
6     @b @b
7 end
```

Figure 4-19: Hello World example using a DEFISM.

provides this behavior when individual iterations have about the same amount of work each.

DEF reaps the benefits of Tapir. Although it means that it isn't a major contribution to theory of parallelism, it's a crucial feature for a language with a scalability design goal. It also means that the various Cilk-related tools all work on DEF applications.[59, 82]

4.6 Interpreted Structural Macros

Interpreted Structural Macros (ISMs) are Lisp-like code geared towards generating DEF, being somewhat more powerful than C or C++ macros. They're designed to fill the role of C++ templates, except operate at more than just the function or type level, and also provide some processing power to the code generation, itself. Also, crucially, generating exported symbols can be C-friendly in a way that C++ templates can't be since the latter depend on name mangling to express the type.

A Hello World example is shown in fig. 4-19. DEFISMs start with an @ symbol and use square brackets instead of parentheses to reduce dependence on the shift key on most keyboards. Line 1 defines `a` as the "hello world" string using the `define` special form, and line 2 uses that definition to define `b` as a print statement. The `parse-stmts` function parses one or more DEF statements and terminates with the closing square bracket. On line 6, the `b` variable is used twice to print "hello world" twice.

Defining functions looks very Lisp-like, too. Fig. 4-20 shows the definition of `call-fcn` on lines 2-5. In this case, the `parse-stmts` appears inside a function

```
1  @[define fcn-names '["foo" "bar" "baz"]]
2  @[define [call-fcn fcn]
3      [parse-stmts
4          @[emit-ident fcn]();
5      ]]
6
7  def call_fcns () -> void
8  begin
9      @[concat-stmts [map call-fcn fcn-names]]
10 end
```

Figure 4-20: Mapping a DEFISM function.

definition and the type of the fcn parameter isn't known ahead of time. So we have to tell the parser what kind of token is being generated using the emit-ident function. When call-fcn is used, it will do its best to convert the argument to an identifier and generate an error if it can't.

call-fcn is used on line 9 along with the map function which applies it to the set of function names. The result is a list of statements, but DEF doesn't know what to do with a DEFISM list, so the statements are concatenated using concat-stmts to turn them into a single object. The result of this code is a DEF function that calls foo, bar, and baz (from line 1).

Naturally, there is a parse-expr function, emit-expr, functions for arranging statements and expressions in multi-branch if statements, string manipulation utilities, etc. The typical Lisp special forms are present, too, including let/let*, if statements, etc. Math functions are included for precomputing values and deduplication of numbers when a DEF variable isn't desired. In principle a program could hang the compiler with an infinite loop in a DEFISM, though the expectation is this is uncommon since the language is designed to generate DEF.

ISM has the power of the DEF parser, so it's easy to emit comprehensible syntax errors in the macros themselves. Moreover, since everything represented in a macro is treated as data by ISM, it's easy to operate on it in ways that aren't possible in simple text-replacement macros.

Chapter 5

Practical DEF

Having surveyed the DEF language and justification for various design elements, we move onto the question of practicality. Although a programmer doesn't have to reimplement existing C/C++ code in DEF, the ask is still quite large to learn a new language. DEF has some advantage in this area in that operating close to the machine isn't that different from C, apart from a few keyword and syntax changes. Nevertheless, it's another compiler in the build process and a development team or researcher would like to know that the language buys them something valuable for the added dependency.

This chapter begins by walking through a series of increasingly complex concurrent data structures, highlighting the DEF features that make their implementations easier and more robust, and their interfaces with C trivial. Performance numbers comparing C and DEF are also presented, even though we understand that any low level language that targets LLVM is going to be as fast as any other. This is to satisfy lingering concerns and recapitulate the idea that the LLVM optimizations that matter are performed on the IR.

5.1 Linked List

We start with a simple concurrent singly-linked list, adapted from the Java implementation in Herlihy and Shavit.[52] This has the typical bounds of a linked list and

```
 1 /** Concurrent linked list implementation. */
 2 export opaque
 3 typedef linked_list =
 4     { head list_node,
 5       tail list_node
 6     };
 7
 8 /** Individual linked list node. */
 9 typedef list_node =
10     { val i64,
11       next *list_node
12     };
13
14 /** Create and return a new list.
15  */
16 export
17 def list_create () -> *linked_list
18 begin
19     var ret = new linked_list
20         { head: { 0x8000000000000000I64, nil },
21           tail: { 0x7FFFFFFFFFFFFFFFI64, nil }
22         };
23     ret.head.next = &ret.tail;
24     return ret;
25 end
26
27 /** Create and return a list node with the given value.
28  */
29 def new_node (val i64, next *list_node) -> *list_node
30 begin
31     return new list_node{ val: val, next: next };
32 end
```

Figure 5-1: List types and initialization functions.

it has lock-free traversals, insertions, and removals.

Fig. 5-1 shows the type definitions (lines 1-12) and the creation routines (lines 14-33). The only type that needs to be exported for other modules is the `linked_list` type if the library author wants it to be opaque. Therefore, line 2 exports the opaque type, and the `list_node` remains private to the module. Note, here, that the order of definitions isn't legal in C – `linked_list` depends on `list_node`, but DEF (like many modern languages) allows types and functions to be defined in any order.

In the same way, `list_create` is exported while `new_node` is only used internally. Both functions use `new` to create their return objects, and they have inline initializers. The `linked_list` statically initializes the node objects within. Fields can be named

```
/** Internal find function: Get the two adjacent nodes in the list
    surrounding
 *  val.  If val is present in the list, it is the second one.
 */
def window_find (head *list_node, val i64)
     -> { *list_node, *list_node }
begin
     var pred, curr, succ *list_node = nil, nil, nil;
  retry:
     while true do
          pred = head;
          curr = get_unmarked_ref(pred.next);
          while true do
               succ = curr.next;
               while is_marked(succ) do
                    succ = get_unmarked_ref(succ);
                    if !__builtin_cas(&pred.next, curr, succ) then
                         goto retry;
                    fi
                    curr = succ;
                    succ = curr.next;
               od
               if curr.val >= val then
                    return { pred, curr };
               fi
               pred = curr;
               curr = succ;
          od
     od
end
```

Figure 5-2: The `window_find` function for getting the requested value (or where it would be if it existed.

or not, but I decided it improved clarity to name the containing object's fields, first and foremost. `nil`, on lines 20 and 21, is the null object pointer. This keyword was used instead of NULL so as not to conflict with the C definition once C macros get more support in the language.

This is not so different from the original Java code except that the constructors have been converted into functions.

`window_find` (fig. 5-2) is a function used by other functions to get a "window" on the location of a value in the list. It returns the predecessor node, and the node itself, or the one that would come after it if it isn't present. Additionally, it tries to swing pointers to nodes that have been logically removed (see [52] for data structure details).

```
/** Find the given value in the linked list and return
 *  true iff it exists.
 */
export
def list_find (list *linked_list, val i64) -> bool
begin
    var curr = &list.head;
    var marked = false;
    while curr.val < val do
        curr = get_unmarked_ref(curr.next);
        marked = is_marked(curr.next);
    od
    return curr.val == val && !marked;
end
```

Figure 5-3: The find-function that returns whether the specified value is in the list.

Line 7 declares the variables that are used by the function, and it must specify the type because no type can be inferred from `nil`. Line 23 returns a statically defined struct containing `pred` and `curr`.

`__builtin_cas`, on line 16, is (as it sounds) a builtin function that emits a CAS and returns the boolean success status. The functions that deal with pointers with overloaded bits are defined below. Again, this code looks very much like the Java code, but the tuple adds a level of convenience.

Fig. 5-3 shows the `list_find` function that returns whether the value is in the list. It doesn't need `window_find`, but performs a simple traversal. `bool` is a builtin type, as it is in both C++ and Java. The only thing that we haven't seen, yet, is the `&` operator, which returns the address of an object as it does in C.

`list_insert`, in fig. 5-4, shows the insertion code which uses `window_find`. DEF can unpack tuples as shown on lines 8-9, or access members using a dot followed by the number in curly braces. This is usually more readable if the tuple doesn't need to be kept around.

On line 19, the node can be deleted and there's no need for a `retire` because its address hasn't been published to other threads. The CAS on line 15 failed, so its address is known only to the thread that created it. This is the first substantial difference from Java, which simply drops the reference and lets its garbage collector clean it up. It takes some thought to prove to oneself that an object need not be

```
/** Insert a new value into the list.
 *  If it's already present, return false.
 */
export
def list_insert (list *linked_list, val i64) -> bool
begin
    while true do
        var { pred *list_node, curr *list_node } =
            window_find(&list.head, val);
        if curr.val == val then
            // Node already present in the list.
            return false;
        fi
        var node *list_node = new_node(val, curr);
        if __builtin_cas(&pred.next, curr, node) then
            // Success!
            return true;
        fi
        node.next = nil;
        delete node;
    od
end
```

Figure 5-4: Code for inserting a new value into the linked list.

retired, and this is the piece that a published data structure ought to specify: which objects have shared their references and which ones haven't? All are treated the same in the literature, but they aren't the same to the programmer working in a low level programming language.

The last significant function is the `list_delete` routine in fig. 5-5 that's essentially the reverse of insertion. In this case, the node must be retired (line 21) because it was visible to all threads.

Unlike the previous case, the reason for retiring it here requires more analysis. A node may be physically removed here or in `window_find`, but it can only be logically removed here. No other thread will help logically remove a node. Therefore, in this implementation I've decided that the thread that logically removes the node retires it. I could have decided otherwise: meaning both functions would have to `retire` conditional on the physical swing of the predecessor's pointer. In this case, only `list_delete` contains a `retire` statement, and the return value from the CAS that swings the pointer is irrelevant. Only the CAS that logically removes the node

```
/** Delete a value from the list.
 *  If it didn't exist, return false.
 */
export
def list_delete (list *linked_list, val i64) -> bool
begin
    while true do
        var { pred *list_node, curr *list_node } =
            window_find(&list.head, val);
        if curr.val != val then
            // Wasn't present in the list.  Return failure.
            return false;
        fi
        var succ = get_unmarked_ref(curr.next);
        var marked_succ = get_marked_ref(succ);
        if !__builtin_cas(&curr.next, succ, marked_succ) then
            continue;
        fi
        __builtin_cas(&pred.next, curr, succ);
        retire curr;
        return true;
    od
end
```

Figure 5-5: Code that removes a node from the list, highlighting the `retire` keyword.

matters. That CAS is significant because if it fails, then the pointer was changed, which may indicate that the node has already been logically removed. Retiring a node that some other thread logically removed would lead to undefined behavior – in Forkscan, potentially a double-free.

Last, by this design the `list_destroy` function knows what can be deleted and what's already been retired by checking the marked bit. Fig. 5-6 shows this function and the marking functions. The latter three functions use a shorthand for one-liners – when the whole function is just an expression to be returned – a function is set equal to the expression.

It also shows that casting looks slightly different from C, using the `cast` keyword. Although alternative kinds of casts are not yet implemented in the compiler, this allows the programmer to be specific about what kind of conversion between types is intended.

All of this code is extremely straight-forward. . . and that's the point. This is not substantially different from the Java source implementation. `retire` buys a developer

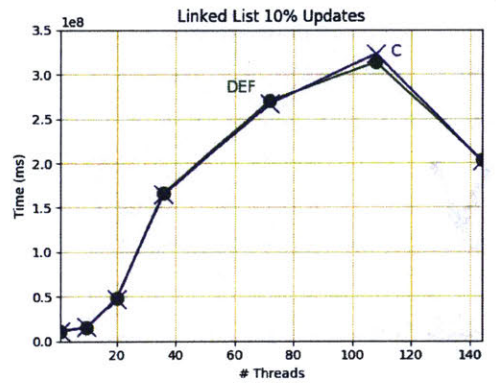
```
/** Perform cleanup on a list, freeing all nodes.  This is not
 *  thread-safe and should not be done concurrently with any
 *  operations on the list.
 */
export
def list_destroy (list *linked_list) -> void
begin
    var node *list_node = list.head.next;
    var prev *list_node;
    while node != &list.tail do
        if is_marked(node) then
            prev = get_unmarked_ref(node);
            node = prev.next;
            continue;
        fi
        prev = node;
        node = prev.next;
        delete prev;
    od
    delete list;
end

/** Given a pointer, return the same pointer with its low bit
    marked. */
def get_marked_ref (ptr *list_node) -> *list_node =
    cast *list_node (0x1I64 | cast i64 (ptr));

/** Given a pointer, return the same pointer without its low bit
    marked. */
def get_unmarked_ref (ptr *list_node) -> *list_node =
    cast *list_node (0xFFFFFFFFFFFFFFFEI64 & cast i64 (ptr));

/** Return whether the given pointer has its low bit marked. */
def is_marked (ptr *list_node) -> bool =
    cast bool (0x1I64 & cast i64 (ptr));
```

Figure 5-6: The destroy function, and functions for handling pointers with the low bit overloaded for marking.



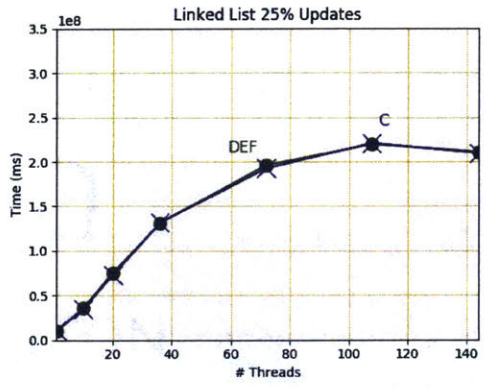


Figure 5-7: Performance of the DEF linked list versus C. Left: Execution with 10% update operations. Right: Same with 25% updates.

the ability to implement the code as presented in the literature, considering only where references are implicitly dropped. In principle, the functions could return pointers to internal nodes instead of booleans (they return booleans because that's what they did in the reference implementation), and the library's end users wouldn't have to worry about handling those pointers except for the constraints imposed by any conservative GC.

Fig. 5-7 shows performance results of the list in DEF versus C. Benchmarks were run on a 72-core machine with two NUMA nodes. With hyperthreading, I ran tests up to 144 threads. For consistency, both versions of the data structure were compiled down to LLVM IR and then converted to object files using the DEF compiler (using `-O3`), guaranteeing they both had the same optimizations performed on them. Without this consistency, the version of LLVM DEF uses is newer than the one Clang uses and performs better – but we're not interested in how well LLVM can optimize; we care about closeness to the machine. Clearly, DEF imposes no hidden costs on any operations, as designed, and the languages perform comparably.

5.2 Serial B-Tree

The B-Tree is a serial tree that's designed to maintain uniform height among all leaves. It isn't binary, so node sizes can be tailored to the application... in principle.

```
1  @[define
2     [make-btree keytype t]
3     [let* [[suffix [string-append keytype "_" [string t]]]
4            [leaf [string-append "leaf_" suffix]]
5            [node [string-append "node_" suffix]]
6
7            [serial_btree_create [string-append "serial_btree_create_"
      suffix]]
8            [destroy_node [string-append "destroy_node_" suffix]]
9            [serial_btree_destroy [string-append "
    serial_btree_destroy_" suffix]]
10           [search_node [string-append "search_node_" suffix]]
11           [serial_btree_contains [string-append "
    serial_btree_contains_" suffix]]
12
13 ...
14
15           [nt t]
16           [nt-1 [- t 1]]
17           [n2t [* t 2]]
18           [n2t-1 [- [* t 2] 1]]
19          ]
20      [parse-stmts
```

Figure 5-8: Macro for making custom parameterized B-Trees.

In practice, benchmarking in C to find the right dimension is hard because of code duplication. Whereas C++ has templates and high level languages have generics, C is required to use macros, which are hard to debug. And speaking from experience, the B-Tree is hard to get right – even with pseudocode.

DEFISMs, on the other hand, not only permit debugging, but allow a degree of freedom templates and generics don't since they allow a developer to programmatically generate code. I therefore present an easily benchmarked B-Tree in DEF, as implemented according to Cormen, Leiserson, Rivest, and Stein.[21] Unlike the linked list, and for the sake of brevity, only relevant snippets of the code are shown. But the purpose is to demonstrate the effectiveness of having a meta-language that's able to operate on its own data.

To begin, a programmer creates what is essentially a template of the types and functions. For our purposes, the B-Tree is parameterized on the key type and `t`, the minimum number of elements in a node. Fig. 5-8 shows the preamble for the `make-btree` ISM function. Line 3 shows the `suffix` added to each function and type

```
 1 @[define types '["i64" "f64"]]
 2 @[define dimensions '[2 3 4 5 6 7 8]]
 3
 4 @[concat-stmts
 5    [map
 6      [lambda [t]
 7        [concat-stmts
 8          [map [lambda [d] [make-btree t d]]
 9            dimensions]
10        ]]
11      types
12    ]]
```

Figure 5-9: Macro for creating a variety of B-Trees of different sizes and types.

to name-mangle in a human readable way. Most of the rest of the variables defined in the `let*` statement are names of types and functions that will be readable to C programmers (lines 4-11); the B-Tree has a lot of functions, so there are a lot of variables, and I've omitted many of them for brevity (line 13). Lines 15-18 are values that will be used repeatedly throughout the code regarding the dimensions. And line 20 opens the `parse-stmts` block where the DEF code will be written.

After this, the implementation of the B-Tree follows, including types and functions, using these variables. `make-btree` is essentially a glorified C++ template (albeit with readable function and type names), but ISM can apply it as a function.

Fig. 5-9 shows how to generate a large number of B-Trees while treating the parameters according to which they're generated as data. Lines 1 and 2 define the parameters: the types and dimensions of the B-Trees.[1] Lines 4-12 are a common Lisp-like way of operating on all possible combinations along the two degrees of freedom: map a lambda onto each of the lists and call the `make-btree` function. Of course, `concat-stmts` is required (lines 4 and 7) since the output is an ISM list and DEF doesn't know what to do with that.

`defghi` will output all of the exported DEF types and functions generated by the macro to be used by C or other DEF modules. More than this, in benchmarking the different sizes and types, the same model can be used to generate driver code, configuration, and even the `-h` message for the application, with adding dimensions

[1]The dimension, called the t-value in CLRS, is the minimum number of keys a node can have.

```
1  @[define [big-if body-fcn]
2     [construct-if
3       [apply append
4         [map
5           [lambda [t]
6             [map [lambda [d] [body-fcn t d]]
7                  dimensions
8           ] ]
9           types
10    ] ] ]
11  ]
```

Figure 5-10: Macro for generating if-statements over the key types and dimensions of the B-Tree.

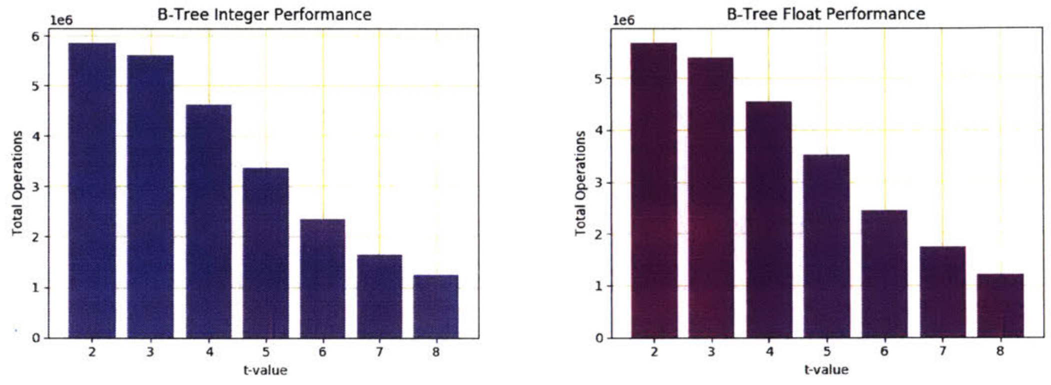


Figure 5-11: Performance comparison of the B-Tree for various widths. Left: Integer operations over a 10 second execution with 25% updates. Right: Floating point operations.

and types updating all of these regions of code automatically.

Fig. 5-10 is an example of a function that generates branches for all of the different types and dimensions. It takes a `body-fcn` parameter, a function that takes a key type and dimension and outputs a pair: an expression (a conditional) and a set of DEF statements (such as a loop for the benchmark, a `printf` for the help message, an insert call for the initialization, etc.). Lines 3-10 simply map the `body-fcn` onto all possible combinations and generate a big list of pairs. The `construct-if` function (line 2) takes this list and generates the if statement based on it. This is a useful example because it's a function that can be applied to other macros, and uses a number of common Lisp/Scheme functions that are built-in.

Fig. 5-11 shows performance results of the B-Tree generated in this way. It's a serial data structure, so all results are single-threaded. The results should not be surprising: for this workload, the most important thing is for nodes to use as few cache lines as possible. But the thing to take away from this example is that generating these results used DEFISMs in which little or no code required duplication, and for which some parameters for the code generation were treated as data by the macros.

5.3 Concurrent Multi Queue

Building on the B-Tree, a serial data structure, the Multi Queue (MQ) is a concurrent priority queue implementation that relaxes the correctness constraint in favor of performance. An MQ is a front-end to a collection of strict priority queues, such as those based on B-Trees, wherein a queue is randomly selected for insertion or removal. In the case of removal, if two queues are queried with a "peek min" method, and the lower value is selected, the MQ has probabilistic guarantees on the results. The particular properties of MQ's are beyond the scope of this thesis but the implementation is interesting for our purposes, particularly if the sub-queues are serial since they don't allow threads to access them simultaneously.

The conventional implementation protects each sub-queue with a mutex. These mutexes aren't especially memory intensive since there's only one per sub-queue, but contention is a problem as thread count grows. As a consequence, on systems with more cores a MQ is implemented with more sub-queues. We consider a ratio of sub-queues to threads, c, for measuring the memory cost of a MQ implementation, and want to know whether lock ellision can achieve the same performance, possibly with less memory. As we'll see, it won't on this data structure, but DEF's transaction syntax combined with its macros is well suited to testing the problem. Walking through the code, consider what would be required to implement this in C or C++.

There are two versions of the code: the locked and the transactional MQs. In the case of the locked version, I implemented a quick test-and-test-and-set (TTS) lock which is fast because very little time is spent inside the critical sections, so we

```
1  @[define [make-mq keytype t is-locked]
2     [...
3       [parse-stmts
4  @[emit-stmts
5     [if is-locked
6       [parse-stmts
7  typedef @[emit-ident set] =
8      { lock    volatile u64,
9        btree   *@[emit-ident btree],
10       padding [48]char;
11     };
12      ]
13      [parse-stmts
14 typedef @[emit-ident set] = { btree *@[emit-ident btree] };
15      ]
16    ]]
17
18 export opaque
19 typedef @[emit-ident mq] =
20     { count i32,
21       sets  [1024]@[emit-ident set]
22     };
```

Figure 5-12: Macro code for the locked (blocking) MQ types. Some of the variable definitions have been omitted for brevity.

expect very little spinning. The locks are low-overhead, spacewise, though they're more than we'd like since the existence of the lock requires the sub-queues to be spaced out across cache lines.

Fig. 5-12 shows the DEFISM that generates the types based on the various dimensions and/or types of B-Trees. There's an interest, again, in testing B-Trees of different widths because the access pattern on a priority queue is different from that of random insertion and removal, and we don't know *a priori* that the same performance results will apply. Note that `make-mq` (line 1) takes an `is-locked` parameter that's used on line 5 to define the `set` type (lines 7 and 14).[2]

Why is this a valuable way to define the types? Virtually all of the code between the locked and transactional versions of the data structure is identical. Quite apart from bugs, we're interested in ensuring that our performance comparisons are apples-

[2]The `set` type has been defined as a struct with one member on line 14 because it makes accesses to the sub-queue the same in routines. But there is no space or performance difference versus setting the set type equal to a pointer to the `btree` type.

```
1  export
2  def @[emit-ident mq-add] (seed *u64, mq *@[emit-ident mq], val i64)
3      -> void
4  begin
5      var a = random_val(seed, mq.count);
6  @[emit-stmts
7     [if is-locked
8       [parse-stmts
9      tts_lock(&mq.sets[a].lock);
10     @[emit-ident btree-add](mq.sets[a].btree, val);
11     tts_unlock(&mq.sets[a].lock);
12      ]
13      [parse-stmts
14     atomic begin
15         @[emit-ident btree-add](mq.sets[a].btree, val);
16     end
17      ]
18     ]]
19 end
```

Figure 5-13: Naïve insertion code for the MQ, both for locked and transactional versions of the structure.

to-apples. Really, the only other code that's different between the two kinds of MQs is the initialization, insertion, and pop-min routines. The initialization is trivial: set the locks to zero, if they exist, and initialize the sub-queues. Insertion is interesting because there are a couple of options in how we implement the transaction.

Fig. 5-13 shows how to implement the `mq-add` routine. The caller passes a seed for the random number generator, since the struct is probabilistic, the MQ, and the value to be inserted. A random value is generated using the seed on line 5, and depending on which kind of MQ this is, it takes a lock (lines 9-11) or performs a transaction (lines 14-16). This code is somewhat naïve in the transactional case because if the transaction fails, it means that it collided with another transaction on the same random subqueue. The other thread may have been inserting, or it may have been performing a pop, but either way the other thread will try again. It behooves us to perform the random selection, again, in the case of failure so we don't wind up colliding with the other thread multiple times.

Consider the alternative presented in fig. 5-14. The failure clause on lines 4-5 chooses another random value if the transaction aborts for any reason. With or

```
 [parse-stmts
atomic begin
    @[emit-ident btree-add](mq.sets[a].btree, val);
xfail _:
    a = random_val(seed, mq.count);
end
 ]
```

Figure 5-14: Smarter insertion code for the transactional MQ. This is an alternative to the code presented in lines 13-17 above.

without this clause, it will try until it succeeds, but now it's increased its likelihood of success if the failure was caused by a collision.

It's worth pausing, here, and examining the difference between this kind of model versus C macros or C++ templates. For a single MQ, the most obvious benefit of a DEFISM over a C macro is the parser – the compiler understands what's in the DEFISM, which is quite different from simple text substitution. This isn't especially big in the case of the MQ since the code is pretty simple. More subtly, however, note that we've already mixed DEF data (keytype and dimension) with ISM data (`is-locked`). ISM can operate on the data in a way that C macros can't.

As compared with C++ templates, DEFISMs can generate code inline; not merely at the function or type levels. If there's a substantial overlap in code, and if the difference is hard to separate out into its own function because it requires significant parameterization, this leads to less code duplication. And, again, the ability to operate on the data, itself, can't be overstated. While one could write two versions of a function that explicitly instantiate an `is-locked` template parameter, that requires duplication of the whole function. Perhaps the MQ's insertion code is not such a big deal, since it's short, but popping the minimum value is somewhat more significant.

Popping the minimum value, again, requires a peek at two random sub-queues, and selection of the lesser. The boilerplate is in fig. 5-15. Line 6 shows the random selection of array indices, and `popped` (line 7) is the return value. The work of the function is implicit in line 9, and the only difference between the two versions is whether locks are taken or a transaction is performed. This code assumes the MQ isn't empty, even if individual sub-queues are, and keeps trying until it finds a

```
1  export
2  def @[emit-ident mq-pop-min] (seed *u64, mq *@[emit-ident mq])
3      -> @[emit-ident keytype]
4  begin
5    try_again:
6      var { a, b } = pick_two(seed, mq.count); // a < b
7      var popped @[emit-ident keytype];
8
9      // The business goes here.
10
11     return popped;
12 end
```

Figure 5-15: Pop-min boilerplate for the MQ.

value to pop. We could, in principle, return a tuple with `{ success, popped }`, but linearizability in that implementation is costly. Instead we use a `try_again` label (line 5) that it can jump to if both selected queues are empty.

If pop-min were serial, fig. 5-16 would solve the problem. The `a` and `b` B-Trees are checked for emptiness – if both are empty, try again (line 25). Peek the min value on each, and choose the lesser. If this code is in an atomic block, when it jumps out the transaction commits. But if locks are held, we have to be sure to release them (line 24).

This binding is defined in the initial `let` statement near the beginning of the overall ISM function. `release-locks` is shown in fig. 5-17. Very simply, if there are locks to be released, release them (lines 6 and 7); otherwise, do nothing (line 9).

The code in fig. 5-16 can, itself, be bound to a variable and placed in the original boilerplate code with two branches: one locked, and one transactional. Fig. 5-18 shows the versions with `pop-min-business` as the bound value. Lines 4-8 have the locked code that acquires the locks in order, and lines 11-16 use the atomic block. I've added a minor complexity to make the transaction retry without selecting new sub-queues in case of a spurious failure (line 13).

As with the serial B-Tree, any number of MQs can be generated programmatically and exported for use in C. Even types defined in C header files can be used in the MQ and benchmarked.

Fig. 5-19 shows performance results using this code on 72 cores. Observe that the

```
var abtree = mq.sets[a].btree;
var bbtree = mq.sets[b].btree;

if !@[emit-ident btree-empty](abtree) then
    if !@[emit-ident btree-empty](bbtree) then
        var amin = @[emit-ident btree-peek-min](abtree);
        var bmin = @[emit-ident btree-peek-min](bbtree);
        if amin <= bmin then
            @[emit-ident btree-remove](abtree, amin);
            popped = amin;
        else
            @[emit-ident btree-remove](bbtree, bmin);
            popped = bmin;
        fi
    else
        popped = @[emit-ident btree-peek-min](abtree);
        @[emit-ident btree-remove](abtree, popped);
    fi
else
    if !@[emit-ident btree-empty](bbtree) then
        popped = @[emit-ident btree-peek-min](bbtree);
        @[emit-ident btree-remove](bbtree, popped);
    else
        @[emit-stmts release-locks]
        goto try_again;
    fi
fi
```

Figure 5-16: The "business" part of the code. Peek both B-Trees and pop from the one that had the lesser value.

```
[let* [
        ...
        [release-locks
          [if is-locked
            [parse-stmts
              tts_unlock(&mq.sets[a].lock);
              tts_unlock(&mq.sets[b].lock);
            ]
            [parse-stmts 0; ]
        ] ]
        ...
```

Figure 5-17: `release-locks` is defined as releasing the locks if and only if this MQ is a locked data structure. This `let*` performs all of the bindings in the `make-mq` function.

```
@[emit-stmts
    [if is-locked
      [parse-stmts
     tts_lock(&mq.sets[a].lock); // Ordered to avoid deadlock. a < b
     tts_lock(&mq.sets[b].lock);
     @[emit-stmts pop-min-business]
     tts_unlock(&mq.sets[a].lock);
     tts_unlock(&mq.sets[b].lock);
      ]
      [parse-stmts
     atomic begin
         @[emit-stmts pop-min-business]
     xfail TF_SPURIOUS:
     xfail _:
         goto try_again;
     end
  ] ] ]
```

Figure 5-18: The new "business" part of the function with both locked and transactional versions of the code.

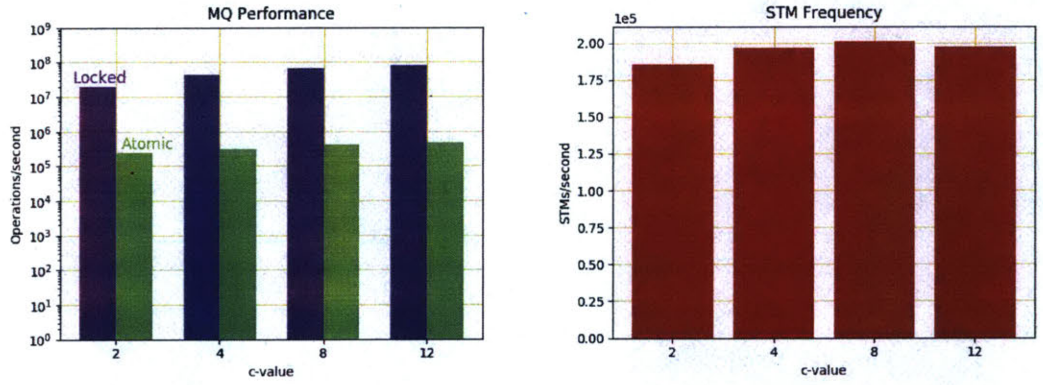


Figure 5-19: Performance comparison of the MQ using locks vs. atomic transactions. Left: Performance results versus c-value, and Right: the frequency with which transactions fell back to the STM slow path.

chart on the left uses a logarithmic scale to compare performance – the locked version of the structure is orders of magnitude faster than the transactional one. The chart on the right gives us some clue as to why: conflicts are rampant. Operations fall back to the software slow path *virtually every time.*

To understand why this is, if we think back to what a concurrent priority queue is fundamentally doing, we recognize that lots of threads are trying to get the smallest item in the set. Even with relaxed semantics, as the MQ has, no matter how big the set, there's a restricted range of likely candidates for any thread to select. One thread peeks at the minimum value of two queues and removes the lesser. Over the course of that operation, it's common for another thread to conflict on at least one of the cache lines involved.

No harm, no foul, though. The cost of comparing this code was low. Writing the hybrid transactional code was trivial after implementing the locked version – a few simple modifications, really. Not so in C. One might be tempted to ask whether writing a hardware transaction-only version of the program would have been simple in C; get a rough sense of the behavior of transactions, generally, before writing the hybrid code? The answer is: there is no hardware-only version of this data structure. The B-Tree allocates memory for which the allocator will call the OS if it needs more, causing any hardware transaction to abort. In reality, the software slow path is a necessary component in the B-Tree based MQ.

To test this data structure in C would require extensive work that's hard to get right... only to discover it was all for nought. Abstractions for certain kinds of concurrent operations are the answer to many concurrency questions, even in low level programming languages. It's certainly true that sometimes a programmer wants to program even those features close to the machine, but that's not restricted in DEF – the hybrid transaction library that DEF uses is written in DEF! Merely, the existence of the abstraction opens up possibilities to HPC programmers that were prohibitively expensive, before.

Chapter 6

Conclusions

6.1 Summary

Fundamentally, what I've done is to make a stab at designing the language I'd wished I'd had back when I was in industry. High performance programming is hard and it takes a lot of work. Performance engineering requires a lot of meticulous variable testing, and scalability... well, parallelism and concurrency are so fraught that our field is still at the stage where a person can get a paper with a single concurrent data structure accepted to a prestigious conference. This combination is legitimately hard. But it's harder than it needs to be.

One can do these things in a low level language and fight the language, itself, to accomplish them. Or one can simply give up on performance engineering and use a high level language. Or one can develop a new programming language to address the difficulties. This thesis represents the culmination of that third option.

The two fundamental difficulties we explored with parallel and concurrent programming are concurrent memory reclamation and atomicity, primarily in the form of transactions. The former is unique to low level programming languages because high level languages use GC to detect what threads can reach. This is a costly solution, both in performance/latency and in control over instructions and memory layout. Transactions are not, in principle, unavailable to high level languages, but it's uncommon even for such a language to support them. A high level language's advantage is that if it has good abstractions, it's possible to reimplement synchronization features using them. But conventional low level languages like C and C++ are unlikely ever to integrate either of these things, as both are contrary to their design philosophies.

Moreover, we observe that the intersection between languages that are designed for performance engineering, in that they are close to the machine, and ones that provide high level abstractions for parallelism and concurrency is the null set. Languages don't necessarily fit into either of these categories, but if they belong to one, they certainly don't belong to the other. This excluded middle isn't inherent. It took some work to devise a solution to the concurrent memory reclamation problem that would neither impede conventional performance engineering on a single core, nor impede the read operations that are common in many concurrent data structures. But such a solution exists, so there's no barrier to the existence of a language that fills that void.

And we've established a motivation – a need for a language in that intersection. Quite apart from my own personal industry woes, insofar as the popular notion of Moore's Law has decoupled from the technical one (processors have stopped speeding up at the rate that more transistors fit onto a die), high performance programming demands scalability, even as the need for performance engineering grows.

DEF targets that void. It's close to the machine in the way that C or C++ is, but it includes non-intrusive high level abstractions for concurrency. Specifically, it supports the concept of `retire`, in addition to the common `new` and `delete` allocator primitives, and it supports hybrid transactions as part of the native `atomic` syntax.

`retire` is implemented by Forkscan, which incurs no performance penalties unless memory is actively being retired, and even those penalties are limited to the snapshot it makes of memory – an infrequent operation, on the whole. Clearly, a C program could (and did, in the original Forkscan paper) use Forkscan's API to retire memory... but at the cost of abstraction. The abstraction, itself, imposes no cost and the generality it provides means that as DEF is ported to other platforms, so are applications that use that feature even if Forkscan is unavailable on them.

Similarly, hybrid transactions with `atomic` syntax have the benefit of future-

Figure 6-1: Visualization of the solution to the question: performance engineering or scalability?

proofing in that the theory behind how to implement them is active. If nothing else, not requiring a programmer to implement hybrid transactions anew in each application means that as the compiler improves, so does performance. But the real win, here, is the compiler support, itself, independent of the progress of theory. Hybrid transactions are scary to do by hand, and completely unmaintainable. And for what benefit? The software transaction is the uncommon slow path. Are programmers eager to leverage their knowledge of the semantics of the application to optimize that beyond what a compiler can do without that knowledge? It's a fool's errand. And it's not easy to get right or maintain. Let the compiler optimize the uncommon slow path.

These features and the suite of other tools provided by DEF enable programmers to develop high performance, scalable code. No sacrifice of either is required. DEF is also transparently compatible with C, so integrating it into an existing C codebase (or vice versa) is low burden. Its feature set works to benefit C applications, and itself benefits from C libraries. What's good for DEF is good for C.

Coming full circle, DEF is the language I wanted every time I had to work in C/C++. Even coming back to school, working with Nir on a concurrency library for

C, it became apparent that the barriers to scalability were too high. DEF, though comparatively immature, enables this kind of a project. What remains, now, is to hone the language for actual applications – not just benchmarks.

6.2 Future Work

Numerous sundries remain unimplemented or unoptimized in the compiler. That work needs to be done, of course, but it isn't interesting from a research-y perspective and can be done in bits and pieces as the language develops and evolves to fit the HPC applications initially implemented in it.

More significant are the applications and tools built in and around DEF that push the boundaries of high performance computing.

6.2.1 Concurrency Library

As mentioned above, Nir brought me on to write a concurrency library for C, as Java has. This didn't come to fruition in quite the way we expected. But the goal is still valid. C lacks such a concurrency library, not least of all because it has nothing resembling generic types. So even when such an undertaking is attempted using conventional means of memory reclamation, the result looks more like a benchmark for or template of concurrent data structures. DEF's native support for memory reclamation and atomic operations and blocks makes it suitable for writing data structures that look good and perform well, and its macro language makes those data structures available to C programmers.

Our field is still in the Wild Wild West, as far as transactions are concerned, where people don't quite know what will work and low hanging fruit abounds. This is true not just for lock elision in existing data structures and algorithms (such as the set data structures used as examples in this thesis), but even in contexts where operations are relatively simple, but which happen to span multiple words. DEF is uniquely suited to explore this space, and for the benefit of legacy C applications that want to parallelize.

6.2.2 Scientific Computing or Machine Learning Application

Part of becoming a properly developed language is evolving to fit real world applications. Partnering with researchers in scientific computing or machine learning and getting a handle on the work they do, and practical experience programming DEF in an application setting is the necessary next step. I cite these particular areas because they're hot for HPC research, so it isn't simply developing applications to develop applications; that will happen on its own as DEF builds a following. These areas also have some human-oriented work that interests me, in medicine, climate change, astronomy, etc.

6.2.3 Windows Port

Of all the ports DEF can make, the transition to Windows will be the most challenging. The primary hurdle is the lack of Forkscan. Without a `fork` operating system primitive, it's not entirely clear how to create the snapshot quickly. Sure, `fork` exists in Cygwin, but it's slow – it's a (from a performance point of view) cheap facsimile designed to make Unix-like code compile for Windows.

Windows presents a unique difficulty in that the source code for the OS isn't available to modify. It seems likely that OS support is necessary, so the question is what can a driver provide in the way of corralling a process's threads and mucking with the page table? Can a Copy-on-Write driver implementation compete with Linux-native `fork`, for example?

6.2.4 Heterogeneous and Distributed Computing

These two areas feel like they have something in common; like there's a common abstraction to be discovered. Ultimately, I don't believe that GPUs are the future. GPUs will turn into light-weight cores on the same die as the main CPU. This phenomenon of farming work out to a special-purpose processor that gets iteratively more capable, year by year, until it winds up as another fully-featured processor had already been observed multiple times by 1968, leading to the coining of *wheel of reincarnation* to describe it by Myer and Sutherland.[75] Nevertheless, distinct GPU coprocessors exist now, and even after they are reintegrated, the wheel will come again.

The interesting thing, to me, looking at CUDA, is that there seems to be a fundamental overlap of concepts between GPU computing and distributed (across a network) computing. Primitives for marshalling and unmarshalling data, shipping tasks, distribution of work (possibly even load balancing) are common to both, and it may be that there is a generic way of expressing these concepts that makes it easy to do either (or both simultaneously). Again, this is another kind of high level feature that's unlikely to be adopted by conventional low level languages for philosophical reasons, yet need not impinge on programming close to the machine.

More unexplored territory exists in this area, too, since distributing references across systems in a low level language is difficult to do automatically. This is an exciting time to be in high performance computing.

Bibliography

[1] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM.

[2] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 11–20, New York, NY, USA, 2015. ACM.

[3] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. Forkscan: Conservative memory reclamation for modern operating systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 483–498, New York, NY, USA, 2017. ACM.

[4] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. *ACM Trans. Parallel Comput.*, 4(4):18:1–18:18, May 2018.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, AFIPS FJCC '67, pages 483–485. AFIPS, 1967.

[6] Arm. Arm a64 instruction set architecture. Technical Report DDI 0596 (ID032719), Arm Limited, 2019.

[7] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, New York, NY, USA, 2016. ACM.

[8] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGOPS Oper. Syst. Rev.*, 34(5):117–128, November 2000.

[9] Tim Blechmann. Boost.lockfree documentation. `https://www.boost.org/doc/libs/1_69_0/doc/html/lockfree.html`, Accessed: 2019-02-15.

[10] 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 Not.*, 30(8):207–216, August 1995.

[11] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[13] OpenMP Architecture Review Board. Openmp api, v.5.0. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`, November 2018.

[14] Hans-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 93–100, New York, NY, USA, 2002. ACM.

[15] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 197–206, New York, NY, USA, 1993. ACM.

[16] Hans-Juergen Boehm. Boehmgc, 2015. Available at http://www.hboehm.info/gc/.

[17] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. ACM.

[18] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM.

[19] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Maurice Herlihy, and Gilles Pokam. Invyswell: a hybrid transactional memory for haswell's restricted transactional memory. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 187–199. IEEE, 2014.

[20] Linux Community. Linux 3.13, 2014. Available at http://kernelnewbies.org/Linux_3.13.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[22] Intel Corporation. Intel cilk runtime source. `https://bitbucket.org/intelcilkruntime/intel-cilk-runtime`, Accessed: 2019-02-26.

[23] Microsoft Corporation. Microsoft docs: Fibers. `https://docs.microsoft.com/en-us/windows/desktop/procthread/fibers`, Accessed: 2019-02-28.

[24] NVIDIA Corporation. Cuda toolkit documentation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, Accessed: 2019-04-29.

[25] CPPReference. C atomic operations library. `https://en.cppreference.com/w/c/atomic`, Accessed: 2019-02-15.

[26] CPPReference. C++ atomic operations library. `https://en.cppreference.com/w/cpp/atomic`, Accessed: 2019-02-15.

[27] CPUWorld. Release dates of desktop microprocessors. `http://www.cpu-world.com/Releases/Desktop_CPU_releases_(2015).html`, Accessed: 2019-09-10.

[28] Robert D. Blumofe and Charles Leiserson. Space-efficient scheduling of multithreaded computations (extended abstract). *SIAM Journal on Computing*, 27:202–229, 02 1998.

[29] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *SIGPLAN Not.*, 47(4):39–52, March 2011.

[30] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[31] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

[32] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. The JCilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2005.

[33] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. Cpu db: Recording microprocessor history. *Commun. ACM*, 55(4):55–63, April 2012.

[34] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM.

[35] David Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[36] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[37] Jason Evans. Jemalloc, Retrieved 2018-08-06. Available at https://github.com/jemalloc/jemalloc.

[38] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. ACM.

[39] Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 124:5, August 2004.

[40] Keir Fraser and Timothy L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[41] Sanjay Ghemawat and Paul Menage. Tcmalloc, Retrieved 2018-08-06. Available at http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[42] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8):1173–1187, 2009.

[43] Google. Go language faq. `https://golang.org/doc/faq`, Accessed: 2018-08-03.

[44] Google. Go syncmap package. `https://godoc.org/golang.org/x/sync/syncmap`, Accessed: 2019-02-15.

[45] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. Java language specification (java se 8 edition). `https://docs.oracle.com/javase/specs/jls/se8/html/`, 2015-02-13.

[46] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45(9):1563–1581, Nov 1966.

[47] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench. In *Proceedings of the 20th Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[48] Andreas Haas, Thomas Hütter, Christoph M. Kirsch, Michael Lippautz, Mario Preishuber, and Ana Sokolova. Scal: A benchmarking suite for concurrent data structures. In Ahmed Bouajjani and Hugues Fauconnier, editors, *Networked Systems*, pages 1–14, Cham, 2015. Springer International Publishing.

[49] Tim L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

[50] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The cilkview scalability analyzer. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 145–156, New York, NY, USA, 2010. ACM.

[51] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 339–353, 2002.

[52] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[53] Richard L. Hudson. Go: The journey of go's garbage collector. `https://blog.golang.org/ismmkeynote`, Accessed: 2019-02-17.

[54] Intel. Intel 64 and ia-32 architectures software developer's manual: Volume 2. Technical Report 325383-060US, Intel Corporation, September 2016.

[55] Michael Kerrisk. *The Linux Programming Interface.* No Starch Press, Inc., San Francisco, CA 94103, 2010.

[56] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[57] Bradley C. Kuszmaul. Supermalloc: A super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 41–55, New York, NY, USA, 2015. ACM.

[58] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[59] I-Ting Angelina Lee and Tao B. Schardl. Efficiently detecting races in cilk programs that use reducer hyperobjects. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 111–122, New York, NY, USA, 2015. ACM.

[60] Charles E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[61] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems*, pages 206–220. Springer, 2013.

[62] Robert Love. *Linux System Programming, 2nd Edition.* O'Reilly Media, Sebastopol, CA 95472, 2013.

[63] Redis Labs Ltd. Memtier benchmark, Retrieved 2016. Available at https://github.com/RedisLabs/memtier_benchmark.

[64] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. *Ada Lett.*, 34(3):103–104, October 2014.

[65] Alexander Matveev and Nir Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. *SIGARCH Comput. Archit. News*, 43(1):59–71, March 2015.

[66] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, New York, NY, USA, 2015. ACM.

[67] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, , and M. Soni. Read-copy update. In *In Proc. of the Ottawa Linux Symposium*, page 338?367, 2001.

[68] Paul McKenney. Whatis rcu, fundamentally? `https://lwn.net/Articles/262464/`, Accessed: 2019-07-29.

[69] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[70] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[71] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[72] Microsoft. .net fundamentals of garbage collection. `https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals`, 2017-03-29.

[73] Microsoft. Windows virtual memory functions. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366781(v=vs.85).aspx#virtual_memory_functions`, Accessed: 2017-02-28.

[74] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), 1965.

[75] T. H. Myer and I. E. Sutherland. On the design of display processors. *Commun. ACM*, 11(6):410–414, June 1968.

[76] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. *SIGPLAN Not.*, 49(8):317–328, February 2014.

[77] Oracle. Java concurrency package. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`, Accessed: 2018-08-06.

[78] Chuck Pheatt. Intel&reg; threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[79] The Clang project. Clang: a c language family frontend for llvm. `https://clang.llvm.org`, Accessed: 2019-04-15.

[80] Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras - non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 367–369, New York, NY, USA, 2017. ACM.

[81] Dennis M. Ritchie. The development of the c language. *SIGPLAN Not.*, 28(3):201–208, March 1993.

[82] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. The cilkprof scalability profiler. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 89–100, New York, NY, USA, 2015. ACM.

[83] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 249–265, New York, NY, USA, 2017. ACM.

[84] Rainer Schuetze. Concurrent garbage collection in D. `http://rainers.github.io/visuald/druntime/concurrentgc.html`, Accessed: 2018-08-06.

[85] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 121–139, 2014.

[86] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 263–268, May 2000.

[87] Bjarne Stroustrup. Stroustrup faq. `http://www.stroustrup.com/bs_faq.html`, Accessed: 2019-04-29.

[88] The Rust Docs Team. The rust reference. `https://doc.rust-lang.org/reference/`, Retrieved 2019-02-25.

[89] Robert Virding, Claes Wikström, Mike Williams, and Joe Armstrong. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[90] WIKI, Accessed: 2019-03-20. `http://en.wikipedia.org/wiki/Unix_signal`.