Dynamic Atomicity: Optimizing Swift Memory Management

David Ungar
IBM Research, USA
ungar@mac.com

David Grove
IBM Research, USA
groved@us.ibm.com

Hubertus Franke
IBM Research, USA
frankeh@us.ibm.com

Abstract

Swift is a modern multi-paradigm programming language with an extensive developer community and open source ecosystem. Swift 3's memory management strategy is based on Automatic Reference Counting (ARC) augmented with unsafe APIs for manually-managed memory. We have seen ARC consume as much as 80% of program execution time. A significant portion of ARC's direct performance cost can be attributed to its use of atomic machine instructions to protect reference count updates from data races. Consequently, we have designed and implemented *dynamic atomicity*, an optimization which safely replaces atomic reference-counting operations with nonatomic ones where feasible. The optimization introduces a store barrier to detect possibly intra-thread references, compiler-generated recursive reference-tracers to find all affected objects, and a bit of state in each reference count to encode its atomicity requirements.

Using a suite of 171 microbenchmarks, 9 programs from the Computer Language Benchmarks Game, and the Richards benchmark, we performed a limit study by unsafely making all reference counting operations nonatomic. We measured potential speedups of up to 220% on the microbenchmarks, 120% on the Benchmarks Game and 70% on Richards.

By automatically reducing ARC overhead, our optimization both improves Swift 3's performance and reduces the temptation for performance-oriented programmers to resort to unsafe manual memory management. Furthermore, the machinery implemented for dynamic atomicity could also be employed to obtain cheaper thread-safe Swift data structures, or to augment ARC with optional cycle detection or a backup tracing garbage collector.

CCS Concepts **• Software and its engineering → Garbage collection**; *Software performance*; *Multiparadigm languages*;

Keywords Swift, ARC, Reference Counting

ACM Reference Format:
David Ungar, David Grove, and Hubertus Franke. 2017. Dynamic Atomicity: Optimizing Swift Memory Management. In *Proceedings of 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3133841.3133843

1 Introduction

The Swift programming language strives for safe and easy programming with good performance across a wide range of platforms and applications, including embedded applications, servers, and every Apple platform. These goals have inspired its designers to provide a wide range of features in multiple paradigms, and to implement them with a sophisticated, ahead-of-time compiler. Swift manages memory by automatically counting references, and provides a library-based concurrency model. But the intersection of these features impedes performance. For example, in one version of the Richards benchmark, Swift reference counting operations consume 80% of total execution time. These measurements were obtained with Swift 3, the latest version of Swift available when the work was done.

We have implemented *dynamic atomicity*, an optimization that reduces the costs of reference count updates without programmer intervention. The basic idea is a straightforward extension of prior work, and would be quite simple to implement in a pure object-oriented language with an unoptimizing interpreter. But, complications arise when extending the basic idea to a language that mixes object-oriented and functional programming constructs, a rich storage model, and a sophisticated, optimizing ahead-of-time compiler. This paper contributes: an extension of the technique of a thread-escape store barrier to a system with reference-, value-, and existential-types; a detailed explanation of the changes required to a large, sophisticated compiler and runtime for its implementation; and a preliminary performance evaluation.

Background on the Swift Language Swift is a very rich language [Apple 2016]. Its value types–structures, enumerations, and tuples–are passed by value and stored in the stack. Its reference types–closures and class-instances–are passed by reference and stored in the heap. A reference may point to either Swift or Objective-C data, either statically or dynamically determined; and may denote different ownership modes including strong, weak, and unowned. A value of any type can contain a value of any other type. For example, a structure could contain a reference to a class-instance which in turn contains an enumeration, which could have variously-typed values associated with each case. Swift's rich static type system includes generics to generalize over all of the above, and also includes protocols. The most dynamic type merely ensures that a variable's value will implement some set of protocols. Such entities are called *existentials.* In concert, these language features provide good support for both functional- and object-oriented programming. Swift's combination of language features and implementation technology makes Swift an attractive language for servers and cloud-computing because of its potential for achieving both high programmer productivity and high performance density (CPU throughput per MB of memory).

Language Implications for Its Implementation Swift's richness does entail significant implementation complexity. For example, virtual functions, enumerations, dynamic layouts of class-instances, and existentials all require dynamic dispatch but must use different mechanisms to achieve it. Tight interoperabiltiy with Objective-C and C and high performance depends on an industrial-strength compiler built on the LLVM compiler framework, and a non-compacting memory manager, *Automatic Reference Counting* (ARC), which must manage both Swift and Objective-C data, and the various ownership modes. Implemented mostly in C++, Swift 3's compiler, runtime, and libraries encompass about 1.5MLOC.

Concurrency and Memory Management Because Swift 3 supports noncooperative multi-threaded concurrent programming via the Grand Central Dispatch library [GCD 2017], its implementation must serialize updates to reference counts. Unsynchronized simultaneous updates produce incorrect counts, which then fail to recycle discarded memory or recycle memory which is still in use. The former failure leads to crashes caused by memory exhaustion, and the latter leads to bugs and security vulnerabilities caused by unintended memory sharing and heap corruption. Swift 3 ensures the correctness of reference counting operations in the presence of read-read data races by using atomic machine instructions to update reference counts. Like the Objective-C ARC specification [ARC 2012], Swift 3 specifies that read-write and write-write data races have undefined ARC behavior.[1]

The Problem Unfortunately, atomic instructions are significantly more costly than their nonatomic equivalents. Experimentally, we have observed that the majority of the direct cost of Swift 3's reference counting operations is attributable to the use of atomic machine instructions. However, many objects never become accessible to a second thread and thus cannot be part of a data race. If ARC could use cheap nonatomic instructions for these non-escaping objects, it could significantly reduce reference counting overheads. At the most extreme, the Swift 3 linked-list microbenchmark could triple its performance. The nbody and Richards benchmarks could double their performance. Swift would perform better if the expensive operations could be reserved for the cases in which they are needed.

Our Solution We designed and implemented a compiler-assisted dynamic optimization for reference counting in Swift 3, called *dynamic atomicity.* At runtime, it is possible to (conservatively) detect memory operations that might allow (a reference to) an object to escape the thread that created it. Such an escaping object can be marked, so that it uses the safer, slower, *atomic* reference counting operations. However, when one object escapes, every object that can be reached from it also escapes. Thus, our implementation includes compiler-generated support for reference tracing, similar to that used to recursively free objects.

To avoid undue overhead for escape detection, our system makes approximations. These approximations are conservative, detecting escapes where there are none, so that concurrent programs will not fail because of missed escapes. To measure the performance that could be achieved were escape detection to be perfect, we performed a limit study by unsafely making all reference counting operations nonatomic. Measuring 171 Swift 3 microbenchmarks, 9 of the Computer Language Benchmarks Game programs, and the Richards benchmark, we observed potential speedups of up to 220% on the Swift 3 microbenchmarks, 120% on Benchmarks Game, and 70% on Richards. Our implementation of dynamic atomicity achieves a significant portion of this potential speedup: up to 150% on the Swift 3 microbenchmarks, 101% on Benchmarks Game, and 60% on Richards.

The next section summarizes the relevant features of the Swift language and gives a conceptual description of dynamic atomicity. Section 3 presents key aspects of our actual implementation of dynamic atomicity for Swift 3. Section 4 contains the experimental evaluation of our prototype. The paper concludes by discussing related work, avenues for future work, and possible additional applications in Swift of aspects of our underlying implementation.

2 Overview: The Swift Language and Dynamic Atomicity

Although we were familiar with automatic memory management for purely object-oriented languages, the extension of techniques such as store barriers and reference-tracing–both needed for dynamic atomicity– to Swift 3's multi-paradigmatic implementation required some thought. This section distills our conclusions by providing a conceptual overview of the

[1]See Jones et al. [2011] 18.1 for a discussion of why the basic atomic machine instructions that are actually provided by hardware are insufficient to protect reference counting from read-write and write-write races.

implementations of the various kinds of values and the additions required to support dynamic atomicity. For simplicity of exposition, we present the ideas as if Swift 3 were implemented via a simple interpreter and did not pack logically separate state variables into bit fields. Section 3 refines this conceptual presentation by describing aspects of the real implementation that are significantly different owing to complexities in the Swift 3 language or implementation.

2.1 Introduction to Swift Types

Swift's three varieties of types, reference-, value-, and existential-types require three different implementation strategies.

2.1.1 Reference Types

Class-instances and closures are represented by *reference types.* Two variables denoting the same class-instance contain distinct references to the same data, which we will call the *referrent.* This level of indirection permits two variables, each containing distinct references, to share state by both referring to the same referrent. A store operation, such as *var a = SomeClass()* stores a reference into *a*, so a reference is a *storeableValue* but a referrent is not. The referrent is a (heap-allocated) block of memory containing a reference count, an escaped flag, and some storeableValues. The escaped flag is added to support dynamic atomicity.

```
struct Reference: StoreableValue {
    var referrent: Referrent
}
class Referrent {
    var refCount = 0
    // newly-created object does not escape
    var escaped = false
    var contents: [StoreableValue] = []
 }
```

The ARC retain function increments refCount, and always uses atomic instructions in the base implementation. With dynamic atomicity, the *escaped* flag selects which flavor of reference count operation is performed. For example,

```
extension Referrent {
    func retain() {
        if escaped  {
            atomicallyIncrement(&refCount)
        } else  {
            nonatomicallyIncrement(&refCount)
        }
    }
}
```

The release function, which decrements *refCount* is similarly modified to check the *escaped* flag to select the appropriate reference counting operation. To conserve space, we elide the definitions of release throughout this section.

2.1.2 Value Types

In contrast to how variables of reference types provide for the sharing of state, variables of value types enforce isolation and confinement of side-effects. Each value-typed variable is allocated enough memory to contain its complete storeableValue. For example, the Swift structure is the value type analogue of the class instance for reference types.

```
struct Structure: StoreableValue {
    var contents: [StoreableValue] = []
    func retain() {
       contents.forEach { $0.retain}
    }
}
```

When one structure is assigned to another, each StoreableValue is copied. Thus, the entire structure is a storeableValue, and a Swift 3 StoreableValue may be more than one word.

Primitive types in Swift are represented as structures, so the simplest (non-vacuous) structure is an integer:

```
struct Integer: StoreableValue {
    var contents: Int
    func retain() {}
}
```

A Swift enumeration resembles a discriminated union (a.k.a. a sum type). It contains a tag that records which of a number of cases is represented. Each case may include a multiplicity of storeableValues.

```
struct Enumeration: StoreableValue {
    var tag: Int
    var cases: [[StoreableValue]]
    func retain() {
        cases[tag].forEach { $0.retain }
    }
}
```

Because it is a value type, enough storage must be reserved to hold the largest case (and the tag). When one enumeration is assigned to another, the tag and storeableValues in the appropriate case are copied.

2.1.3 Existential Types

A Swift protocol, like an interface in other languages, records the signatures of the operations, getters, and setters that a value might support. It need not state that a value must be a reference type or a value type. Thus if the only type associated with a variable is a (set of) protocol(s), it is not possible to statically choose between value- and reference-type semantics for assignment. Such a type, known as an **existential type,** is implemented with a dynamic check:

```
enum Existential: StoreableValue {
    case structure(Structure)
    case reference(Reference)
    case enumeration(Enumeration)
    var contents: StoreableValue {
        switch self {
        case .structure  (let s): return s
        case .reference  (let r): return r
        case .enumeration(let e): return e
        }
    }
    func retain() { contents.retain() }
}
```

2.2 Catching Escapees

To implement dynamic atomicity, the system must maintain the invariant that **every referrent that can be accessed from more than one thread is marked as escaping.** Since a referrent can only become accessible as the result of a store of a reference to it into an already globally accessible memory location, a *store barrier* must catch any store of a previously non-escaping storeableValue into a context that is accessible outside of its creating thread. Such a context may be a scope, if a storeableValue is stored into a simple variable, or it may be a larger storeableValue, if the storeableValue is stored into a component of another storeableValue:

```
indirect enum Context {
    case localScope
    case globalScope
    case referrent(Referrent)
    case structure(Structure)
    case enumeration(Enumeration)
    case existential(Existential)
}
```

The execution engine must be able to supply the context for each possible target of an assignment:

```
protocol HasContext {
    var context: Context {get}
}
extension Structure:   HasContext {}
extension Reference:   HasContext {}
extension Enumeration: HasContext {}
extension Existential: HasContext {}
```

This information forms the step in the recursion required to catch an escape:

```
func escapesWhenStoredInto(_ context: Context)
  -> Bool {
    switch context {
    case .localScope: return false
    case .globalScope: return true
    case .referrent(let r): return r.escaped
    case .structure(let s):
        return escapesWhenStoredInto(s.context)
    case .enumeration(let e):
        return escapesWhenStoredInto(e.context)
    case .existential(let x):
        switch x {
        case .reference(let r):
            return r.referrent.escaped
        case .structure, .enumeration:
            return escapesWhenStoredInto(x.context)
        }
    }
}
```

2.3 Propagating Escapes

When a referrent is about to escape, all reachable storeableValues must be marked as escaped before the escaping store is actually performed. Doing the marking before the store prevents data races, because it ensures that referrents are marked as escaped before they become visible to other threads.

```
func store(_ value:   StoreableValue,
           into dest: StoreableValue) {
    if dest.container.escapes {
        value.markEscaped()
    }
    // do the actual store
}

extension Reference {
   func markEscaped() { value.markEscaped() }
}
extension Referrent {
    func markEscaped() {
        // Break recursion on cycles & avoid
        // reprocessing already escaped referrents
        if escaped { return }
        escaped = true
        contents.forEach { $0.markEscaped() }
    }
}
extension Structure {
    func markEscaped() {
        contents.forEach { $0.markEscaped() }
    }
}
extension Enumeration {
    func markEscaped() {
        cases[tag].forEach { $0.markEscaped() }
    }
}
extension Existential {
    func markEscaped() { contents.markEscaped() }
}
```

2.4 Summary

In summary, implementing dynamic atomicity requires that:

1. A reference counted object must maintain a bit of state to know if it has escaped its creating thread,
2. Every reference count operation must query that bit to select between atomic and nonatomic instructions,
3. The system must catch an escape before it happens and inform the escapees, and
4. In order to inform all the escapees, the system must find every object reachable from the transitive closure of the escaping value.

3 Details of Swift Implementation and Dynamic Atomicity

As described above, catching and marking escaping objects is conceptually simple. However the addition of this functionality to the implementation of the complete Swift language required non-trivial enhancements to both the Swift 3 compiler and runtime. Overall, our optimization added about 5,000 lines of code to the existing 1.5 million lines of code.

Swift 3 is ahead-of-time compiled, using an LLVM-based compiler. Compilation proceeds in stages: The parser generates an AST. The AST is walked to produce an intermediate form called raw SIL (Swift Intermediate Language). Raw SIL is type-checked and transformed to produce canonical SIL, or just SIL. This SIL can be optimized, and then is traversed to produce LLVM IR. Finally the LLVM optimizes the IR and generates machine code. Our changes were confined to the stages up through the SIL to IR translation stages; no LLVM modifications were needed.

To provide necessary context for explaining our implementation of dynamic atomicity, we first describe the implementation of ARC in Swift 3 and how the different kinds of Swift 3 values support reference tracing. With this background in place, we detail the key aspects of our implementation: maintaining the escape bit (3.2), using the escape bit in reference count operations (3.3), catching escapees via a store barrier (3.4), and recursively marking escaped objects (3.5).

3.1 ARC Implementation for Swift 3

Swift 3's Automatic Reference Counting (ARC) reclaims unused memory and calls user-supplied destructors when doing so. It is designed to provide predictable overheads and deterministic finalization. It reclaims unreachable memory eagerly to minimize peak heap size. Heap-allocated data includes a 32-bit (strong) reference count structure.[2] That structure includes a bit for *pinning*, a bit for *deallocating* and a 30-bit *count* of the number of extant strong references to the heap allocated data.

3.1.1 Basic Operations

The basic operations for ARC are the creation and destruction of references, called *retain* and *release*, respectively. When a new reference is created, the count is incremented, when a reference is destroyed, the count is decremented. When the count is decremented to zero, the allocation is freed. However, if the allocation itself contains references, then freeing it destroys those references and so, prior to freeing the allocation, the contained references must have their counts decremented. Thus a *release* can potentially cause a recursive cascade of releases and deallocations.

The Swift 3 compiler automatically inserts retain and release operations as it translates Swift source level statements into LLVM IR. For example consider the assignment of a reference from b to a, i.e. the statement $a = b$. First the current value of a is saved in a temporary location. Next b is retained. Then the reference to b is stored in a. Finally the old value of a that was saved away in the temporary location is released. If this release causes a reference count to go to zero, any references contained in the referenced object are also released and finally the memory is deallocated.

[2]There is also a structure to count weak references, but weak references are outside this paper's scope, as we expect them to be less frequent.

If two threads simultaneously try to increment or decrement a reference count without proper synchronization, the count could end up with an incorrect value. Synchronization is enforced by always using atomic machine instructions to access and update the reference count structure. Although the Swift 3 runtime includes nonatomic variants of all the reference count operations for experimental purposes, by default all operations use atomic variants.

3.1.2 Reference Tracing

How are the contained references found–either for assignment of value types or for recursive freeing? As much work as possible is done at compile-time, but dynamic dispatches are needed to handle runtime variation. These dispatches take different forms for reference types, enumerations, and existentials:

Reference Types A value of a reference type, for example an instance of a class, is stored in the heap, and has a header consisting of a reference count and a *metadata* pointer. The per-type metadata includes information about the layout of the object, which cannot always be statically-determined. It also contains (for heap-allocated objects) a word that points to a destructor function.

This destructor function handles the side-effects of destroying a class-instance. It is invoked when the object's reference count falls to zero, traverses all of the references contained in the object and decrements them. In addition, this function performs any custom finalization the programmer has specified. It must be dynamically-dispatched because of inheritance and genericity. This destructor function is created and massaged throughout the compilation pipeline. In addition to releasing external resources, programmer-supplied finalizers have been exploited to implement reclamation for core data structures such as Arrays. An Array contains an untyped *UnsafeMutablePointer* that points to the actual memory for the array. When the array is destroyed, a source-level *deinit* routine invokes a function on the *_elementPointer* that bottoms out in a call to a builtin to release the stored elements before deallocating the memory.

Enumerations A value of a value-type, such as an enumeration, is stored on the stack, up to a certain size, or in the heap if too large. A pointer to its metadata is carried along with its value if need be. Enumerations employ a variety of strategies, designed to optimize important cases. For example, an enumeration that represents a nullable reference can be stored in a single word, by employing unused bits in the pointer. Most operations on an enumeration require a conditional branch on the tag value to handle the appropriate case. In particular, copying an enumeration requires such a branch to generated code to (among other things) increase the reference counts of any references included in the particular case of the enumeration.

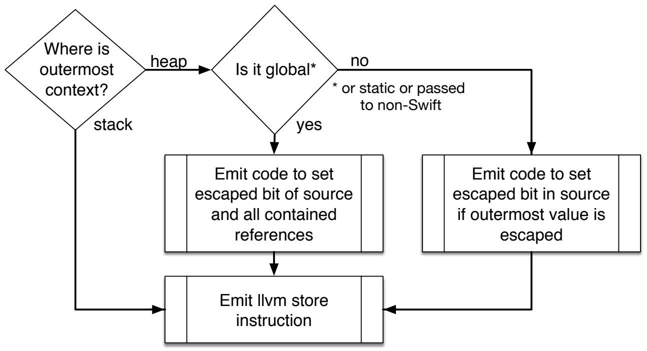
Where is outermost context?
heap
stack
Is it global*
no
* or static or passed to non-Swift
yes
Emit code to set escaped bit of source and all contained references
Emit code to set escaped bit in source if outermost value is escaped
Emit llvm store instruction

Figure 1. Compiler logic for determining what store barrier code to emit for every store to detect escaping references.

Existentials To represent an existential (whose runtime size may not be statically determined), Swift 3 allocates a three-word buffer on the stack, plus another word to hold the pointer to the metadata. If the existential is a reference (at runtime), the *reference* is stored in the buffer; if it is an instance of a value-type that fits in the buffer, the *value* is stored there; otherwise the value is stored in the heap (but no reference count needed) and a *pointer* is stored in the on-stack buffer. Thus, a dynamic check is required merely to find the data. In addition, even basic operations such as assignment, retain, and release, require a dynamic dispatch. Thus, every existential has a *value witness table*, located via the metadata, which functions in a similar fashion to a vtable. It has entries at fixed offsets for such operations as assignment, retain, and release.

3.2 Maintaining Escaped State

Dynamic atomicity takes the lowest bit from the 30-bit reference count to track the escaped state of a heap-allocated object. As a result there are now three flag bits (*pinned, deallocating, escaped*) and 29 bits of count in the 32-bit reference count structure. Objects are allocated with the escaped bit clear, since the newly allocated object must be local to its creating thread. The Swift 3 standard library preallocates a few shared global objects as native static data; we modified the initialization of these objects to mark them as escaped.

3.3 Using Escaped State

Dynamic atomicity inserts a test of the escaped bit into each reference count operation, such as retain and release. If an object has not escaped, the nonatomic version is used, otherwise, the atomic version is used as in the base system.

3.4 Catching Escaping Values

The modifications required to catch escaping values were the most difficult aspect of dynamic atomicity: stealing a bit in the reference-count and redirecting operations to nonatomic variations was simple, and tracing all references in order to mark all escapees (3.5) involved a lot of code but we had the destructor routines to use as a model. In contrast, implementing a store barrier that caught every store and supplied the required context represented new functionality in the Swift 3 implementation. It was necessary to select the best stage in the compilation pipeline to intercept every store operation and identify its context. (See figure 1.)

We chose the compilation stage at which Swift Intermediate Language (SIL) is transformed into LLVM Intermediate Language instructions, which comes after Swift-specific optimizations have been performed. Had we chosen to intercede earlier, we would have needed to add additional SIL elements to carry the escape operations through the pipeline, and would have needed to modify all the Swift optimization stages accordingly. However, by interceding after the optimizations, our code must deal with their effects. For example, a SIL store operation has been lowered so that initialization cannot be distinguished from assignment.

To intercept conceptual store operations at this stage, our system adds code to the LLVM IR generation routines for the SIL *store* and *copyAddress* instructions. Since SIL, being an SSA form, contains backpointers to the SIL instructions producing values, our code recursively traces backwards from the destination of a store in order to find the outermost context needed to determine whether that store might create an escape. The pseudocode shown in Figure 2 illustrates backtracking through the SIL.

In addition to storing a value, passing it to a non-Swift function also causes it to escape. Our code intercedes at every such call site. This detection is straightforward: whenever a call is generated the compiler knows whether it is a call to Swift or to another language. However, Swift's *inout* parameter-passing convention poses additional problems.

3.4.1 Inout Parameters

Swift incorporates a facility for passing parameters by reference and allowing a callee to modify a caller's data, *inout* parameters. For example:

```
func increment(_ i: inout Int) { ++i }
var x = 0
increment(&x)
// now x == 1
```

At the call site, an *&* is used to indicate such parameters and the callee must declare them with the *inout* modifier.

Even forgetting about dynamic optimization, the aliasing potential of an inout parameter can create problems for both Swift 3 programmers and implementors. For example, the same value can be passed inout to two arguments to the same function, leading to unmaintainable code:

```
var i = 12
func crazy(a: inout Int, b: inout Int, c: Int) {
   a = 0
   return c / b
}
print( crazy(a: &i, b: &i, c: i) )
```

```
void func visitStoreInst(i: StoreInst) {
  emitStoreBarrier(i.src, i.dest, false)
  // do the store
}
void func visitCopyAddrInst(i: CopyAddrInst) {
  let isInitialization =
       getLoweredValue(i.dest)
         .isUnallocatedAddressInBuffer
    || i.isInitializationOfDest
  emitStoreBarrier(
    i.src, i.dest, isInitialization);
  // copy the address
}

void func emitStoreBarrier(
  src: SILValue,
 dest: SILValue,
  isKnownToBeInitialization: bool) {
  switch StoreTagCheck
           .backtrackForTagCheck(from: dest) {
  case .never:
    break
  case .always:
    emitMarkEscaped(src)
    break
  case .conditional(let valueToCheck):
    if valueToCheck.kind == .globalAddrInst
    || valueToCheck.isClassType
       &&  valueToCheck.classDecl.isReallyObjC {
      emitMarkEscaped(src)
    }
    else if !isKnownToBeInitialization {
     emitCheckEscapedBit(
      of: valueToCheck, thenMark: src)
    }
  }
}
```

```
enum StoreTagCheck {
  case never, always, conditional(SILValue)

  static func backtrackForTagCheck(from v: SILValue)
                    -> StoreTagCheck {
    if v.isAllocStack || v.isIndexRawPointer {
      return .never
   } else if v.isGlobalAddr || v.isProjectBlockStorage {
      return .always
   } else if v.isAllocBox || v.isRefTailAddr || v.isLoad {
      return .conditional(v)
    } else if v.isReferenceTypeElement {
      return .conditional(v.firstOperand)
   } else if v.isAllocValueBuffer || v.isUncheckedAddrCast
     || v.isUncheckedTakeEnumDataAddr
     || v.projectExistentialBox {
      return v.type.isReferenceCounted
        ? .conditional(v) : .never
    } else if v.isSILArgument {
        return v.isInOut && v.type.isReferenceCounted
        ? .conditional(v) : .never
   } else if v.isTupleAddrElement || v.isStructElementAddr
     || v.isMarkDependence || v.isInitEnumDataAddr
     || v.isIndexAddr || v.isProjectValueBuffer
     || v.isStructExtract || v.isPointerToAddress
     || v.isOpenExistentialAddr || v.isTypeExtract
     || v.isUncheckedEnumData {
     return backtrackForTagCheck(from: v.firstOperand)
    } else if v.isApply {
      if v.type.isReferenceCounted {
        return .conditional(v)
      } else if v.callee.isGlobalInit {
        return .always
      } else if v.callee.name == "materializeForSet" {
       return backtrackForTagCheck(from: v.lastArgument)
      } else {
        return .never
      }
    } else {
      return .always // don't know; conservative
    }
  }
}
```

Figure 2. Pseudo code for backtracking through SIL instructions to determine the outermost context for a Store operation.

The Law of Exclusivity proposed for Swift 4 would make such weird cases of aliasing brought about by inout parameters illegal. [Swift 2017]

If a reference type is passed as an inout parameter, the callee can check its escaped bit, as with any other store. However, if a value type is passed as an inout parameter, the callee cannot know, short of stack-walking, whether the argument value is contained in an escaping object or not. Our current implementation of dynamic atomicity adds code at every call site that passes an inout argument: After the callee returns, the outermost aggregate of each inout argument is checked in the same fashion as a store. If the outermost aggregate is global or a reference-type (e.g. a class-instance) with the escaped bit set, the argument is marked (recursively) as escaping.

Although this implementation suffices for all the multi-threaded programs we have tried, it does have a loophole: Consider the time interval before the callee returns but after the callee assigns a value type (containing a reference) to an inout parameter that is part of an escaped class-instance. The newly-assigned reference may not be marked as escaping, yet has escaped. The acceptability of this loophole depends on the granularity of the Law of Exclusivity: Does it operate at the level of a single variable, or at the level of an object subgraph? If the former obtains, we will have to revisit this solution. For example, the call site could create temporary, thread-private copies for inout value-typed arguments. Such a scheme would be safer, at the expense of copying overhead.

3.5 Recursively Marking Escaped Objects

When a value escapes, all references reachable from it must be marked as escaping. This operation resembles the destroy operation, in which all references reachable from a destroyed value must have their counts decremented. Both operations must trace every reachable reference and do something with each. Ideally, such tracing code would be factored, generalized, and reused. Since we are experimenting rather than putting dynamic atomicity into production, we chose to duplicate rather than generalize and reuse. Recall that heap-allocated objects have a header that points to their metadata which points to a destructor function. Our optimization extends the metadata with a pointer to a *markEscaped* function. Since we didn't expect to immediately implement the recursion for all kinds of things, we adopted the convention that a null-pointer in the metadata meant that the runtime should just use atomic reference counting operations. Accordingly, a check was added to heap object creation to set the reference-count escaped bit if the metadata contained a null pointer here.

For structures and enumerations, we duplicated and modified the existing code for destruction, which combines inline and out-of-line code generated by the compiler.

For existentials, we added entries to the value witness table pointing to appropriate functions for the concrete types.

4 A Limit Study and Preliminary Results

To assess the effectiveness of dynamic atomicity, we used a suite of 171 Swift 3 microbenchmarks from the Swift open source project, several Swift variations of the classic Richards benchmark [Richards 1999], and 9 Swift 3 programs drawn from the Computer Language Benchmarks Game. [CLBG 2008] The Swift 3 microbenchmark suite is one of the tools used by the Swift project to track performance and evaluate compiler, library, and runtime system optimizations. It contains a large number of small code snippets, each intended to test a specific feature of the language implementation or standard library. Richards is an OS kernel simulation, originally written in BCPL by Martin Richards and used for years to benchmark various languages. We have translated it to Swift 3 and created a number of variations, each taking advantage of different language features and their varying efficiencies. We have made our Richards implementations available on GitHub. [Ungar and Grove 2017c] The Computer Language Benchmarks Game consists of programs written in many languages that solve 11 programming problems. Since there is no canonical version of any particular program in this suite, we make the specific variants we measured in this paper available on GitHub. [Ungar and Grove 2017a]

Our experiments are designed to answer two questions:

- What is the potential benefit of using nonatomic machine instructions for reference count operations?
- Can dynamic atomicity obtain a significant fraction of that benefit, or do the overheads of maintaining and checking escape state overwhelm the gains of avoiding atomic machine instructions?

To answer these questions, we use four Swift 3 toolchains[3] built from the same code base forked from the master branch of the Swift GitHub repositories as of February 7, 2017. We have made our code publicly available on GitHub. [Ungar and Grove 2017b]

- *baseline*: A Swift 3 toolchain with no modifications.
- *nonatomic*: The baseline toolchain modified to unconditionally, and unsafely, use nonatomic machine instructions for all reference counting operations by patching `RefCount.h`. Although it is not safe in the presence of concurrency, we were still able to apply this toolchain to 75% of our multi-threaded benchmarks and obtain valid program runs.
- *dynamic atomicity*: The baseline toolchain enhanced with our implementation of dynamic atomicity as described in previous sections.
- *instrumented dynamic atomicity*: The dynamic atomicity toolchain with instrumentation enabled. The instrumentation includes individual counters for every atomic and nonatomic reference count operation, for true and false results for escape tests, and for both top-level and recursive escape marking.

Comparing the performance of *baseline* and *nonatomic* addresses the first question by establishing an overly optimistic upper bound on the potential benefit of dynamic atomicity. Comparing all three configurations addresses the second question. We expect the performance of *dynamic atomicity* to be strictly less than *nonatomic*, though it may still be somewhat more optimistic than a complete implementation for Swift 3 due to its treatment of inout parameters (3.4.1).

All experiments were performed on a MacBook Pro with a 4 core 2.5 Ghz Intel Core i7 processor and 16GB of RAM running MacOS 10.12.5. We compiled all programs using *-O -Ounchecked*. We ran each configuration and benchmark pair 20 times and report the median time. The error bars on the graphs indicate the standard deviation.

4.1 Swift 3 Microbenchmarks

Figure 3 compares the performance of *baseline* and *nonatomic* for all 171 Swift 3 microbenchmarks. Over half of the test cases show the potential for at least 5% speedups, with 24

[3] A Swift toolchain includes a Swift compiler, a Swift standard library compiled using that compiler, a runtime system, and other supporting tools.

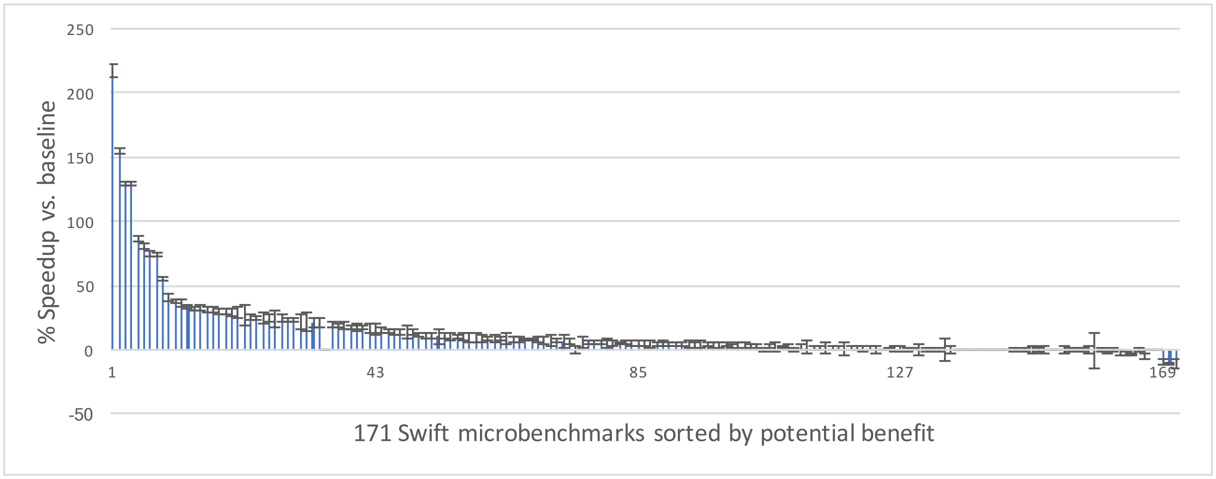

Figure 3. Limit study to evaluate the potential benefit of dynamic atomicity across the entire Swift 3 microbenchmark suite. The graph shows the speedup obtained over *baseline* by the *nonatomic* configuration that unsafely uses nonatomic machine instructions in all reference counting operations. Error bars indicate standard deviation.

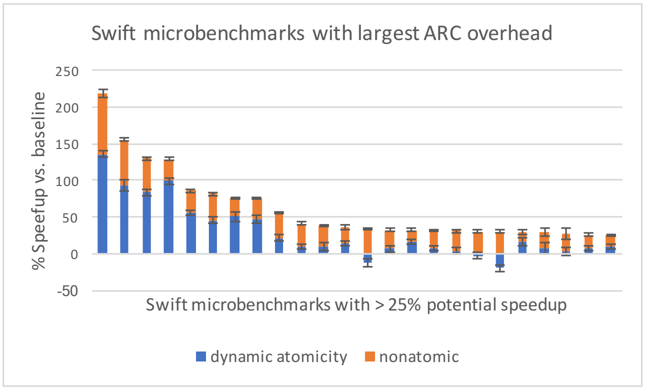

Figure 4. Comparison of potential to actual speedups for the highest impact Swift 3 microbenchmarks. The blue bars show the speedup (slowdown) obtained by *dynamic atomicity*, while the orange bars show the potential speedup represented by *nonatomic*. Error bars indicate standard deviation.

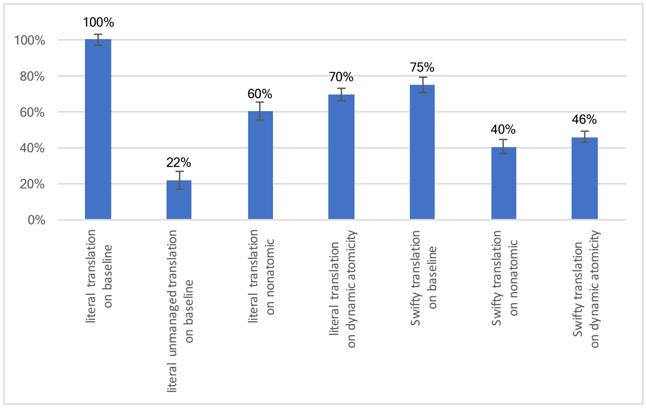

Figure 5. Normalized execution times for *literal*, *literal unmanaged*, and *Swifty* versions of Richards using the *baseline*, *nonatomic*, and *dynamic atomicity* toolchains. Smaller is better. Error bars indicate standard deviation.

of them having potential speedups over 25%. Surprisingly, 4 of the 171 show slowdowns between 5% and 10%; we have not yet determined why this occurred. Figure 4 compares all three configurations on the 24 microbenchmarks with the largest potential speedup. On the 9 with >50% potential speedup, dynamic atomicity achieved 61% of the potential. It achieved 30% of the potential averaged across all 24.

The two cases where dynamic atomicity noticeably degrades performance are measuring Set operations that box the data in class-instances and do a large number of insertions. Each insertion stores a reference to the box into the array holding the set. Because this core library circumvents the type system (possibly for the sake of speed), we had to add Swift code to the library to test to see if the set had escaped and mark the box as escaped if it had. A more polished implementation of Dynamic Atomicity could refactor the library so that the compiler could perform this check. Based on profile data, we believe that this quick-and-dirty library code is the primary source of slowdown in these benchmarks.

4.2 Richards

We report results for two variations of the benchmark: a very *literal* translation into Swift and our fastest translation, which is also a very idiomatic, or *Swifty*, translation. The uninstrumented toolchains supply the execution times, while separate runs on the instrumented chain supply counts of various operations. For example, instrumentation revealed that neither variation makes significant use of global or static variables, and so neither ends up using atomic counting operations in our implementation.

4.2.1 Literal Translation

The literal translation to Swift implements all functions and data as members of a single class. The time taken for this *literal translation on baseline* version, running on an unmodified Swift 3 toolchain is set to 100%, and all other times are relative to it, as shown in Figure 5. The smaller the amount of time remaining, the more effective the optimization.

To investigate how much time is consumed by reference counting, we modified the benchmark to use *Unmanaged* references. Unmanged references risk memory corruption and severely clutter up the code, but improve performance by eschewing reference counting. Compare normal code:

```
devpkt.p_a1 = Int(workpkt.p_a2[count])
```

to its unmanaged counterpart:

```
devpkt.takeUnretainedValue().p_a1 =
  Int(workpkt.takeUnretainedValue().p_a2[count])
```

This *literal unmanaged translation on baseline* version runs in 22% of the original time, suggesting that 78% of the baseline time is spent counting references.

The best possible outcome for dynamic atomicity would be to make all reference counting non-atomic without adding extra overhead. The (managed) *literal translation on nonatomic* version takes 60% of the original time. Thus, atomicity is responsible for 40% of the baseline time, or about 51% (40/78) of the reference counting time.

The *literal translation on dynamic atomicity* variation runs in 70% of the baseline time. It executes 92 million (all nonatomic) reference-counting operations and dynamic atomicity saves about 2.7 ns per operation. Our optimization saves almost one third of the baseline time.

4.2.2 Swifty Translation

The Swifty translation uses enumerations to dispatch functions, protocols to factor out common behavior, and a separate linked list data structure to encapsulate the queuing of tasks and packets. Fortunately, better factoring also yields better performance. This *Swifty translation on baseline* version runs in only 75% of the baseline time. Using Swift's features has saved a quarter of the time.

But atomicity still exacts a heavy price: the (unsafe) *Swifty translation on nonatomic* version takes just 53% (40/75) of the time (40% of baseline) as the same code run in the unmodified Swift 3 toolchain. This 35% (75-40) of baseline atomicity cost) is close to the 40% (100-60) seen for the same comparison with the literal version. This similarity suggests that the two sorts of optimizations, rewriting code to use Swift more effectively and avoiding atomicity, are complimentary.

The (safe) *Swifty translation on dynamic atomicity* version runs in 46% of baseline time. This is close to the limit study ideal of 40%. This version executes 77 million reference counting operations, all nonatomic, and dynamic atomicity saves 3.1 ns per operation. When applied to a translation of Richards designed to use Swift well and run fast, dynamic atomicity has saved 39% ((75-46)/75) of its time.

4.2.3 Summary of Richards

The linked-list intensive systems-programming in Richards is a tough challenge for Swift. Our current implementation of dynamic atomicity can save 35% to 39% of the running time of this sort of computation. Using the speedup metric of *(original time - optimized time) / optimized time,* dynamic atomicity obtains speedups of 43% and 60% for the two versions. The absolute number of milliseconds saved was fairly close for both versions of the code, suggesting that exploiting Swift's features does not obviate the need to optimize away atomicity. Preliminary instrumentation suggests the per-operation cost of atomicity is 3 ns.

Although it must be conservative in detecting escapes, dynamic atomicity's savings come close to those measured for a hypothetical perfect system. However there is an exception: global- or static variables. Such variables were not used in these versions of Richards. More work is needed for these variables.

4.3 Computer Language Benchmarks Game

The Computer Language Benchmarks Game is an ever evolving collection of small programs written in many languages to solve 11 programming puzzles. At the time of this work, there were one or more correct Swift solutions available for 10 of the 11 puzzles. For those benchmarks where multiple solutions were available, we selected the fastest version with the least use of unmanaged pointers to bypass reference counting operations (even with this bias, fannkuchredux, mandelbrot, fasta, and revcomp still make significant use of unmanaged pointers). We excluded pidigits from our evaluation because all Swift versions simply wrapped the GNU Multiple Precision Arithmetic Library which was not available on our experimental machine. With the exception of nbody, all of these codes are multi-threaded.

Figure 6 shows the normalized execution times (smaller bars are better). Even though *nonatomic* is unsound for multi-threaded programs, we were still able to gather data for 6 of the 8 multi-threaded programs (racy reference count updates did cause memory corruption and crashes for spectralnorm and knucleotide). The *nonatomic* configuration indicates that three programs, nbody, regexredux, and binarytrees, have a measurable potential benefit from dynamic atomicity. In all three cases, our implementation was able to capture almost all of the potential gains. Unfortunately, the additional runtime overheads of dynamic atomicity degraded performance for knucleotide.

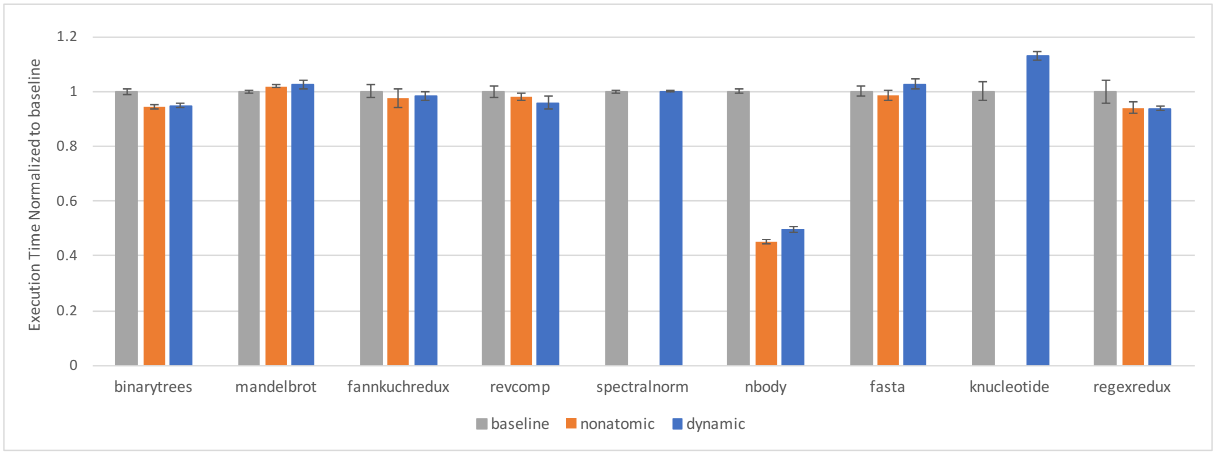

Figure 6. Normalized execution times for 9 Swift programs derived from the Computer Language Benchmarks Game. The gray bars show the normalized baseline time (always 1). The orange bars show the potential gains represented by *nonatomic* for those programs where this unsafe configuration did not result in runtime crashes. The blue bars show the performance obtained by *dynamic atomicity*, Error bars indicate standard deviation. Smaller bars are better.

5 Related Work

Many garbage collection algorithms rely on store barriers [Jones et al. 2011], going back at least as far as Ungar [1984]. Store barriers enable a variety of generational, partial heap, incremental, and concurrent garbage collection algorithms by augmenting the program's heap updating operations with additional semantics to preserve collector invariants.

Domani et al. [2002] first suggested combining a local/global bit in the object header with a store barrier to dynamically detect when a thread-local object may have become visible to other threads. This bit supported thread-local heaps and garbage collection for Java. Daloze et al. [2016] repurposed the store barrier idea of Domani et al. [2002] to optimize the Truffle object storage model for dynamically-typed languages to avoid costly synchronization operations on writes to thread-local objects. Dynamic atomicity extends this previous work to support a richer data model that in addition to classes also includes structures, enumerations with payloads, and existential types. We apply the barrier to enable a new optimization, eliminating atomic machine instructions in reference count updates on thread-local objects.

The direct cost of frequent reference count updates has long been recognized as one of the primary performance challenges of straightforward reference counting algorithms such as ARC. Buffered reference counting [DeTreville 1990] avoids requiring atomic reference count updates by logging reference count operations into thread-local buffers for later processing by a dedicated garbage collection thread. Since all reference count updates are performed by a single dedicated thread, it can perform them nonatomically. However, the algorithm requires cooperation from the application threads to allow the collector thread to construct a consistent snapshot of buffers to process and it delays reclamation of memory until the buffers are processed. Several prior systems have added additional phases to the basic reference counting algorithm to enable the elision of specific categories of reference count operations. Deferred reference counting [Deutsch and Bobrow 1976] elides reference count operations for references stored in stack locations; coalesced [Levanoni and Petrank 1999] and ulterior [Blackburn and McKinley 2003] reference counting extend deferred reference counting to also elide reference count operations for many references stored in slots in heap allocated objects. All three algorithms trade complexity and space for time: They delay the reclamation of unused memory, increasing peak memory requirements, to reduce the number of reference count operations. Such a tradeoff may be unsuitable for Swift's target applications.

Reference counting operations can also be targeted by compiler optimizations that move operations out of loop nests and eliminate redundant operations. [Cann and Oldehoeft 1988; Joisha 2006] The Swift 3 compiler includes ARC-specific optimization passes to look for these opportunities and to specialize calling conventions to reduce release/retain traffic at procedure call boundaries. All of these optimizations were enabled in our experiments, demonstrating that although static optimizations can be effective, they are complimentary to runtime techniques like dynamic atomicity.

6 Future Work

Completing Dynamic Atomicity Global and static variables prevent dynamic atomicity from yielding its full potential. We would like to implement a load-barrier that checks for loading a global into a new thread (a.k.a. thread-biasing). More testing and performance measurements are needed to better assess the costs and benefits of our approach.

Improving Swift Safety and Security For the sake of performance, Swift 3's concurrency model does not preserve heap integrity in the presence of data races. For example, if two threads each store a multi-word structure to two different elements of the same array, it is possible to overwrite freed (and potentially reallocated) memory. The machinery that powers dynamic atomicity could be reused to improve this situation, by tracking which data structures might be accessed concurrently and eliminating unneeded locking.

7 Conclusions

Dynamic atomicity improves the performance of Swift programs by replacing slow, atomic, reference-counting operations with faster, nonatomic ones. It preserves safety by performing this replacement only for reference counts belonging to objects that have not escaped their creating threads, and thus cannot be accessed concurrently. It tracks escapes at runtime by intercepting store operations, determining if the context being stored into is potentially visible to multiple threads, and, when necessary, propagating the escaped status to all references reachable as a result of the store.

The Swift language and its implementation pose challenges for dynamic atomicity. Swift's richness, ahead-of-time compilation, and many static optimizations result in a large source base, 1.5MLOC, which we had to understand and modify. We had to duplicate and repurpose the many different mechanisms required to trace through all values reachable from a given value, including inline code, out-of-line code, and dynamically-dispatched code.

The toughest challenge was the introduction of a store barrier with identification of the proper context to determine thread escapes, because there was nothing similar to it in the existing code. Our implementation intercepts stores near the end of the Swift 3 compilation pipeline, at the transition from Swift IL to LLVM IR. When a value is stored, our implementation backtracks through the Swift IL to recursively traverse the context of the store through the containing value-types in order to find the reference-type context of the store.

By adding a modest 5KLOC to the 1.5MLOC in the Swift 3 implementation, dynamic atomicity can optimize reference counting to slash the frequency of costly atomic operations. Measuring 171 Swift 3 microbenchmarks, 9 of the Computer Language Benchmarks Game programs (CLBG), and several variations of Richards, we observed potential speedups of up to 220% on the Swift 3 microbenchmarks, 120% on CLBG, and 70% on Richards. Our implementation of dynamic atomicity achieves a significant portion of this potential speedup: up to 150% on the Swift 3 microbenchmarks, 101% on CLBG, and 60% on Richards. The effort is still underway–more work may be needed for inout parameters and is definitely needed for global/static variables–but preliminary results look promising. We have open sourced our implementation [Ungar and Grove 2017b,c] and welcome collaboration in future exploration of the potential of dynamic atomicity in Swift.

References

Apple. 2016. *The Swift Programming Language* (Swift 3.1 ed.). Apple Inc.

ARC 2012. Objective-C Automatic Reference Counting (ARC). http://clang.llvm.org/docs/AutomaticReferenceCounting.html. (2012).

Stephen M. Blackburn and Kathryn S. McKinley. 2003. Ulterior Reference Counting: Fast Garbage Collection Without a Long Wait. In *Proceedings of the Conference on Object-oriented Programing, Systems, Languages, and Applications*. Anaheim, California, 344–358. DOI: http://dx.doi.org/10.1145/949305.949336

D. C. Cann and Rod R. Oldehoeft. 1988. *Reference Count and Copy Elimination for Parallel Applicative Computing*. Technical Report CS–88–129. Department of Computer Science, Colorado State University, Fort Collins, CO.

CLBG 2008. Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/. (2008).

Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and Thread-safe Objects for Dynamically-typed Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 642–659. DOI: http://dx.doi.org/10.1145/2983990.2984001

John DeTreville. 1990. *Experience with concurrent garbage collectors for Modula-2+*. Technical Report 64. DEC Systems Research Center.

L. Peter Deutsch and Daniel G. Bobrow. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM* 19, 9 (Sept. 1976), 522–526. DOI: http://dx.doi.org/10.1145/360336.360345

Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-local Heaps for Java. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM '02)*. ACM, New York, NY, USA, 76–87. DOI: http://dx.doi.org/10.1145/512429.512439

GCD 2017. Dispatch. https://developer.apple.com/reference/dispatch. (2017).

Pramod G. Joisha. 2006. Compiler Optimizations for Nondeferred Reference: Counting Garbage Collection. In *Proceedings of the 5th International Symposium on Memory Management (ISMM '06)*. ACM, New York, NY, USA, 150–161. DOI: http://dx.doi.org/10.1145/1133956.1133976

Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.

Joseph Levanoni and Erez Petrank. 1999. *A Scalable Reference Counting Garbage Collector*. Technical Report.

Richards 1999. Richards Benchmark. http://www.cl.cam.ac.uk/ mr10/Bench.html. (1999).

Swift 2017. Swift Ownership Manifesto. https://github.com/apple/swift/blob/master/docs/OwnershipManifesto.md. (2017).

David Ungar. 1984. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. ACM, New York, NY, USA, 157–167. DOI: http://dx.doi.org/10.1145/800020.808261

David Ungar and David Grove. 2017a. Swift Computer Language Benchmarks Game variants. https://github.com/davidungar/benchmarking-swift/tree/Swift_3.2/ComputerLanguageBenchmarkGame. (2017).

David Ungar and David Grove. 2017b. Swift Dynamic Atomicity Implementation. https://github.com/davidungar/Swift-dynamic-atomicity. (2017).

David Ungar and David Grove. 2017c. Swift Richards Benchmark. https://github.com/davidungar/benchmarking-swift/tree/Swift_3.2/Richards. (2017).