Project Snowflake: Non-blocking Safe Manual Memory Management in .NET

MATTHEW PARKINSON, DIMITRIOS VYTINIOTIS, KAPIL VASWANI, MANUEL COSTA, and PANTAZIS DELIGIANNIS, Microsoft Research, U.K.
DYLAN MCDERMOTT, University of Cambridge, U.K.
AARON BLANKSTEIN and JONATHAN BALKIND, Princeton University, U.S.A.

Garbage collection greatly improves programmer productivity and ensures memory safety. Manual memory management on the other hand often delivers better performance but is typically unsafe and can lead to system crashes or security vulnerabilities. We propose integrating safe manual memory management with garbage collection in the .NET runtime to get the best of both worlds. In our design, programmers can choose between allocating objects in the garbage collected heap or the manual heap. All existing applications run unmodified, and without any performance degradation, using the garbage collected heap. Our programming model for manual memory management is flexible: although objects in the manual heap can have a single owning pointer, we allow deallocation at any program point and concurrent sharing of these objects amongst all the threads in the program. Experimental results from our .NET CoreCLR implementation on real-world applications show substantial performance gains especially in multithreaded scenarios: up to 3x savings in peak working sets and 2x improvements in runtime.

CCS Concepts: • **Software and its engineering** → **Memory management**; **Garbage collection**; Object oriented languages; • **Computing methodologies** → *Concurrent programming languages*; *Concurrent algorithms*;

Additional Key Words and Phrases: .NET, GC, Ownership, Memory Management

ACM Reference Format:
Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. 2017. Project Snowflake: Non-blocking Safe Manual Memory Management in .NET. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 95 (October 2017), 25 pages.
https://doi.org/10.1145/3141879

1 INTRODUCTION

The importance of garbage collection (GC) in modern software cannot be overstated. GC greatly improves programmer productivity because it frees programmers from the burden of thinking about object lifetimes and freeing memory. Even more importantly, GC prevents temporal memory safety errors, i.e., uses of memory after it has been freed, which often lead to security breaches.

Modern generational collectors, such as the .NET GC, deliver great throughput through a combination of fast thread-local bump allocation and cheap collection of young objects [Blackburn and McKinley 2008; Stefanovic et al. 1999; Tene et al. 2011]. At the same time several studies show

Authors' addresses: Matthew Parkinson; Dimitrios Vytiniotis; Kapil Vaswani; Manuel Costa; Pantazis Deligiannis, Microsoft Research, U.K.; Dylan McDermott, University of Cambridge, U.K.; Aaron Blankstein; Jonathan Balkind, Princeton University, U.S.A..

This work is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License.
© 2017 Copyright held by the owner/author(s).
2475-1421/2017/10-ART95
https://doi.org/10.1145/3141879

that GC can introduce performance overheads when compared with manual memory management [Hertz and Berger 2005; Hundt 2011; Zorn 1993]. These overheads are amplified in big data analytics and real time stream processing applications as recent work shows [Gog et al. 2015; Maas et al. 2016; Nguyen et al. 2016, 2015], partly due to the need to trace through large heaps. This trend is likely to continue as modern servers make use of ever larger memories – sizes of hundreds of gigabytes, or even terabytes, are already common.

Manual memory management addresses this problem: it avoids tracing the object graph to free objects and instead allows programmers to exploit their knowledge of object lifetimes to free objects at specific program locations. This improves throughput and also achieves better memory usage due to prompt deallocation. The downside is that manual memory management is typically unsafe and can lead to system crashes or security vulnerabilities, because freeing memory may create dangling pointers, i.e., pointers to memory that has been freed, and dereferences of dangling pointers lead to undefined behaviour. Requiring all memory to be manually managed is also unappealing because it negates the productivity benefits of GC.

In this paper, we show how to get the best of both worlds: combining a GC-based system – in our case the Microsoft open-source .NET runtime [CoreCLR 2017] – with a facility to manage memory manually, without compromising safety or performance. In our design, programmers can choose between allocating objects in the garbage collected heap or the manual heap. All existing applications run entirely unmodified using the garbage collected heap, and enjoy no performance degradation. Our design places no overhead on garbage collections or other operations like write barriers. Programmers who wish to optimize their applications need to incrementally change their code to allocate some objects from the manual heap, and to explicitly deallocate those objects. We allow allocation and deallocation of individual objects at arbitrary program locations, and we guarantee that manually managed objects enjoy full type- and temporal- safety, including in the presence of concurrent accesses. Programmers get dynamic managed-exceptions for use-after-free scenarios, but no crashes or security vulnerabilities.

Our novel programming model for manual memory management builds on the notion of unique *owners* of manual objects: locations in the stack or on the heap that hold the only reference to an object allocated on the manual heap. Our notion of *owners* is unique, compared to similar concepts in C++, Rust [rustlang.org 2017], and Cyclone [Swamy et al. 2006]: we allow arbitrary client threads to (a) share stack references to owners (but not to the underlying manual objects), (b) create arbitrary stack references to the actual underlying manual objects from these owner references, and (c) freely abandon the owner reference (which will eventually cause deallocation of the underlying manual objects) – while guaranteeing use-after-free exceptions. To allow safe concurrent sharing of manual objects we introduce the notion of *shields*. Accessing a manual object requires getting a reference from a shield, which creates state in thread local storage that prevents deallocation while the object is being used. Shields can only be created from the unique owning reference, thus when the reference is destroyed no more shields can be created and memory can be safely reclaimed once all previously active shields have been disposed.

We implement this model using a novel combination of ideas drawn from hazard pointer literature [Michael 2004] and epochs for memory reclamation [Bacon et al. 2001; Fraser 2004; Harris 2001] to provide an efficient lock-free manual memory management scheme, without having to scan large portions of the heap. We develop an epoch-based protocol for determining when it is safe to deallocate an object on the manual heap. The protocol accounts for weak memory model effects, but it is non-blocking. That is, it does not require stopping the world or the use of expensive synchronization. We introduce a mechanism that guarantees *liveness* of the epoch protocol by employing virtual memory protection.

We note that our manual memory management scheme and programming model is independent of the integration of manual memory management with garbage collection and could be applicable in a purely manually managed system too, although it would be more difficult to ensure end-to-end memory safety without an additional strong type system.

Our system is implemented as a fork of the Microsoft open-source .NET implementation. We have modified the .NET runtime (CoreCLR) and extended the standard libraries (CoreFX) with APIs that use manual memory. For manual heap allocations we have integrated jemalloc [jemalloc.net 2017], an industrial size-class-based allocator. Experimental results show substantial performance gains with this design: up to 3x savings in peak working set and 2x improvements in run time. In summary, our contributions are:

- A new flexible programming model for manual memory management where objects can be allocated and deallocated at any program point, and can be concurrently and safely shared amongst multiple threads.
- A set of rules at the C# frontend that ensure the safe use of the programming model.
- An efficient implementation of this programming model that does not require stop-the-world synchronization to safely reclaim manual memory.
- A design for safe interoperability with the garbage collected heap that does not adversely impact the write barriers. To keep the latency of Gen0 collections low, we use existing GC mechanisms to scan only the fragments of the manual heap that contain pointers to Gen0 objects, exactly as if those manual objects were in an older generation on the GC heap.
- An implementation and detailed evaluation on industrial use-cases from machine learning, data analytics, caching middleware, and a set of micro-benchmarks.

2 BACKGROUND AND MOTIVATION

Consider the code below, taken from `System.Linq`, a widely used .NET library for LINQ queries on collections that implement the `IEnumerable<T>` interface.

```
IEnumerable<TR> GroupJoinIterator(IEnumerable<TO> outer, IEnumerable<TI> inner,
    Func<TO, TKey> outerKey,
 Func<TI, TKey> innerKey, Func<TO, IEnumerable<TI>, TRes> res) {
  using (var e = outer.GetEnumerator()) {
    if (e.MoveNext()) {
      var lookup = Lookup.CreateForJoin(inner,innerKey);
      do {  TOuter item = e.Current;
            yield return res(item,lookup[outerKey(item)]);
      } while (e.MoveNext());
} } }
```

The code defines an iterator for the results of a join. We iterate through the outer `IEnumerable<TO>`, `outer`. If the outer enumerable is non-empty then we create a `Lookup<TKey,TI>` structure, which is – in effect – a dictionary that maps keys to groupings of elements from the inner enumerable `inner` that share the same key. We then iterate and apply the `res()` function through every `item` of the outer enumerable. The code uses the C# **`yield return`** construct and as a result will be compiled to a state machine that can return the results of `res()` one by one in successive calls.

The intermediate data structure `lookup` can potentially grow as large as the size of the inner enumerable and we have to hold-on to it throughout the iteration of the outer enumerable. It cannot be stack-allocated because **`yield return`** compilation has to save the current state of the iteration and pick it up in the next invocation of the iterator. This `lookup` is an object that is likely then to survive many Gen0 collections (which could happen as a result of allocations inside `e.MoveNext()` or `res()`), and possibly end up in the oldest generation before it can be collected.

It is pretty clear though that once the outer enumerable iteration completes, the interal arrays and data structures of the `lookup` dictionary are entirely safe to deallocate. A generational GC may, however, hold on to these objects until the next full heap collection (such as a Gen2 in the .NET garbage collector) which might happen much later, leading to a blowup of the peak working set; confirmed by our evaluation in Section 5.

Instead we would like to enable programmers to allocate ordinary .NET objects like `lookup` in their manually managed heap, and let them deallocate precisely when they wish. Note that `System.Linq` is a *generic* library that can manipulate not only collections of unboxed data (such as **`struct`**s of primitive types) but also of GC-allocated objects. To maintain this genericity and still be able to allocate internal dictionaries like `lookup` on the manual heap we must allow manual heap objects (like the internal arrays associated with `lookup`) to contain references to GC-allocated objects. Hence a key design goal is to maintain full GC interoperability, allowing pointers to and from the two heaps. This full interoperability is also essential to allow gradual pay-as-you-go migration of applications to use manual memory management for certain objects while others remain in the GC discipline, as performance requirements mandate.

2.1 The Challenge of Safe Manual Memory

Deallocation of manually managed objects, while preserving memory safety, is a challenging problem. We seek ways to ensure – statically or dynamically – that an object will not be accessed after it has been deallocated. The challenge is that references to the deallocated object could be remaining on the heap or the stack after deallocation. This might lead to access violations, memory corruption, accessing data that belongs to newer objects allocated in the same virtual address range etc. Scanning the roots and the heap upon any manual object deletion to discover and zero-out such remaining references can be expensive as each individual scan might have similar cost to a Gen0 collection.

Let us demonstrate the challenge of safety with an example from a multi-threaded caching component of the ASP.NET framework [ASP.Net 2017], a popular framework for web applications. The cache is defined simply as a dictionary of `CacheEntry` objects (for brevity throughout the paper we omit C# access modifiers like **`public`**, **`private`** etc.):

```
class MemoryCache {
  Dictionary<object, CacheEntry> _entries;
  bool TryGetValue(object key, out object res);
  void RemoveEntry(CacheEntry entry);
  void Set(object key, object value,
        MemoryCacheOptions options);
}
```

```
class CacheEntry {
    // cache entry metadata
    ...
    // actual entry
    Object m_Value;
}
```

Each cache entry object contains metadata about this cache entry (such as when it was last accessed) plus the actual payload value of this object `m_Value`. The cache exposes a method `TryGetValue()` that tries to fetch the payload object associated with a key in the cache. Occassionally the elements of the cache are checked for expiration and `RemoveEntry()` is called on those elements that have expired, which removes those entries from the shared dictionary. Client code can use a cache object `_cache` as follows:

```
if (!_cache.TryGetValue(key, out value)) {
    value = ... ;     // allocate a new object
    _cache.Set(key, value, _options); // put it in the cache
}
// perform a computation with value
```

The cache entry payload objects, accessed from the `m_Value` field above, are objects that survive into the older generations and may be collected much later than their removal from the dictionary. They also have a very clear lifetime that ends with a call to `RemoveEntry()`. They are, hence, excellent candidates for allocation on the manual heap. But how to deallocate those objects safely, when multiple threads are accessing the cache and one of them decides to remove an entry?

2.2 Key Insight: Owner References and Shields

In the caching example, the lifetime of the cache payload objects and access to them is controlled by the associated `CacheEntry`. The field `m_Value` acts as the only *owner* of those objects. These objects are only temporarily accessed by stack references in client code (while remaining in the cache) but all of those stack references have been first-obtained through the pointer `m_Value`. Consequently, if we are to zero-out `m_Value` in `RemoveEntry()` no *future* references to the payload object can be obtained.

But when can we actually deallocate and reclaim the memory of the underlying payload? In this example, client threads may have *already* obtained stack references to the object in question before `RemoveEntry()` has been called, and they may still be working on those objects after the corresponding cache entries have expired and have been removed from the dictionary. We cannot deallocate these objects while other code is accessing them.

Our solution to this problem is inspired by hazard-pointers [Michael 2004], a technique originating in the lock-free data structure literature. We introduce a mechanism to publish in thread-local state (TLS) the intention of a thread to access a manual object through one of these owner locations. This registration can be thought of as creating a *shield* that protects the object against deallocation and grants permission to the thread that issued the registration to directly access the manual object e.g. call methods on it or mutate its fields. At the same time no thread (the same or another) is allowed to deallocate the object and reclaim its memory. Once client code no longer needs to access this object, it can dispose the shield, that is remove the reference to this object from its TLS. It is not safe to directly access the object that has been obtained from a shield, after the shield has been disposed because, following this disposal of the shield, the actual deallocation is allowed to proceed (if some thread has asked for it, and provided that this reference does not exist in *any* TLS in the system). If the owner link has been zeroed-out in the meanwhile no new references can be obtained.

This is the key mechanism that we use to ensure manual memory safety. Next, we formally present our programming model and describe the rules that programmers must abide by to preserve memory safety.

3 PROGRAMMING MODEL

The Snowflake programming model is designed for .NET and we present it here in C# syntax. First, we remind the readers of some .NET features we depend on.

3.1 Preliminaries

C#/.NET introduce constructs that give programmers some control over memory layout. In our programming model we will use **`struct`** types, which – contrary to classes – can be allocated on the stack or directly inside another struct or class. Struct assignment amounts to memory copying and **`struct`** arguments are passed by value. In addition, C#/.NET allow to pass arguments (of **`struct`** or **`class`** types) by *reference* by explicitly using the **`ref`** keyword. The address of the corresponding struct will then be passed instead of a copy of the struct, and in the case of classes, the address where the object pointer lives instead of the object pointer.

```
struct Owner<T> where T : class {
  Shield<T> Defend();
  void Move<S>(ref Owner<S> x)
              where S:class, T;
  void Abandon();
}
```

```
struct Shield<T> : IDisposable
                    where T:class {
  static Shield<T> Create();
  void Defend(ref Owner<T> u);
  T Value;
  void Dispose();
}
```

```
class ManualHeap {
  void Create<T>(ref Owner<T> dst) where T:class, new();
  void CreateArray<S>(ref Owner<S[]> dst, int len);
}
```

Fig. 1. Core Snowflake API

3.2 Snowflake API

Figure 1 gives the public Snowflake API. To avoid any confusion we emphasize here that – by itself – the API does not guarantee safety. Safe use of the API relies on a set of language frontend checks, described in more detail in Section 3.4.

Owners. An `Owner<T>` encapsulates a (private, unmanaged) pointer to a manual object. The runtime implementation of our API relies for safety on the *unique owner condition*: No two `Owner<T>` structs should ever be allowed to refer to the same manual object. This condition is enforced by our C# language frontend (see Section 3.4). Note that `Owner<T>` is defined as a **`struct`** as opposed to a **`class`** to avoid an extra GC object allocation per manual object. For reasons explained in Section 3.4 we are only allowed to pass such structs as arguments to functions *by reference*.

Struct `Owner<T>` exposes three methods. The first **`Defend()`**, returns a `Shield<T>` and prevents deallocation of the manual object associated with this owner (by publishing this manual object pointer in thread-local state.) The second **`Abandon()`**, zeroes out the pointer to the manual object, so that no new `Shield<T>` can be obtained, and schedules the manual object for deallocation at some safe point in the future, when it is no longer protected by any shield in any thread. The final method **`Move(ref Owner<S> x)`**, corresponds to transferring ownership from `x` to the receiver struct. The pointer inside the `x` struct is moved to the receiver struct and the `x` struct is zeroed out. If the receiver struct was holding a manual object pointer prior to the call to `Move(`**`ref`**` x)` then that manual object will be scheduled for deallocation at some later safe point, since – by the unique owner condition – the receiver struct was the only owner of that object.

Shields. A `Shield<T>` acts as a stack-only access token to the underlying object. It can be obtained from the `Defend()` method of an `Owner<T>` and encapsulates a reference to thread-local state that records the underlying manual object as one whose memory cannot be reclaimed. It exposes the following members: **`Value`**, is a *property* that gives access to the underlying manual object; and **`Dispose()`** un-registers the manual object that this shield protects from thread-local state, making it thus a candidate for deallocation.

The lifetime of a shield is not tied to a specific access of a specific owner. Shields are only references to slots in thread-local state and can be created in *uninitialized form*, and be used to defend multiple objects. For this reason `Shield<T>` exposes two more methods: **`Create()`** which simply creates a new uninitialized shield that does not yet defend any object against deallocation; and **`Defend(ref Owner<T> u)`** which defends a *new* owner, and *un-defends* the owner it previously defended, if any. This is done by overwriting the TLS slot that corresponds to this shield with the new object pointer. This method is handy for avoiding frequent creation and disposal of shields on

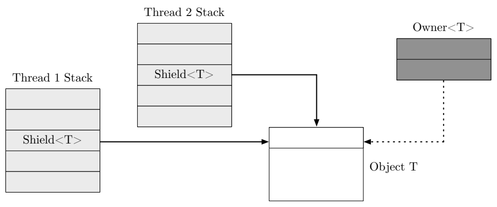

Fig. 2. Owners and shields.

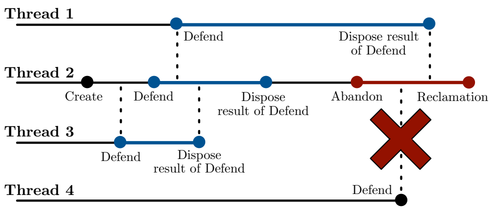

Fig. 3. Example of lifetimes of owners and shields. `Defend` refers to calling `Owner.Defend()` to get a shield; `Dispose` refers to disposing that shield.

every iteration of a loop that accesses some manual objects. We can instead create a shield before the loop (allocate a TLS slot) and dispose it in the end (deallocate a TLS slot), but continuously re-use it (overwrite the pointer in that slot) to defend each item in each iteration.

Allocating on the manual heap. Our API exposes `Create()` and `CreateArray()` methods in Figure 1 for allocating objects and arrays. These methods allocate in the manual heap and transfer ownership of the newly allocated object to the destination owner. In our C# frontend we use syntactic sugar **`new`** `Owner<MyClass>(...)` for allocating in the manual heap *and* calling a constructor. Here we just focus on the low-level .NET APIs.

3.3 Putting It Altogether: Lifetimes

Figure 2 demonstrates how the stack and the heap may look during an execution of code that uses our programming model. An owner can live on the stack, the GC heap or manual heap and contains a link to the underlying manual object. Multiple threads may have used this owner to obtain stack-local shields, which internally refer to the manual object directly.

Figure 3 on the other hand describes owner and shield lifetimes in a left to right chronological order. Thread 2 creates an owner object and subsequently creates a shield through `Defend()` that allows it to access the underlying object up to the point of disposal of that shield. At the same time Thread 3 can also access the same object by obtaining another shield. At some point later on, `Abandon()` is called from Thread 2, and the object is scheduled for deallocation. Thread 1 has in the meanwhile *also* defended and obtained a shield, so is keeping the object from being deallocated. Deallocation can proceed once no thread holds an (undisposed) shield to the underlying object. Finally, Thread 4, cannot obtain a shield as it is issuing the `Defend()` call after Thread 2 has abandoned the owner.

3.4 Language Frontend and Safety

As mentioned in the beginning of the section, extra checks are needed at the C# frontend level to ensure safe use of the Snowflake .NET API. Although the emphasis of this paper is on the efficient lock-free runtime, in this section we describe those checks for completeness:

Ensuring the unique owner condition. If `Owner<T>` structs are allowed to be duplicated then we may be holding on to references into deallocated manual objects, because we have violated the unique owner condition. As an example, consider the unsafe code fragment in the left part of Figure 4. Similar dangers exist anywhere `Owner<T>` could be subject to copying. Concretely our frontend implements the following rules:

- No assignment or cloning of a heap location or stack variable of type `Owner<T>` is allowed.

```
class Foo {
    Owner<Object> mobj;

    void meth() {
       Owner<Object> mobj1;
       mobj1 = mobj;
     // violated unique owner condition
       mobj.Abandon();
       var sh = mobj1.Defend();
       sh.Value.ToString(); // unsafe!
}   }
```

```
var sh = mobj.Defend();
var sh1 = sh;
// referring to the same TLS slot!
... use sh.Value here ...
sh.Dispose();
// Object can go away now
... use sh1.Value here ... //
    unsafe!
```

Fig. 4. Unsafe owner (left) and shield (right) examples

- No passing or returning `Owner<T>` by value is allowed. If we were to pass `Owner<T>` by value then the callee and caller would effectively be holding two copies of the struct, compromising safety.
- No instantiation of generic methods with `Owner<T>` types is allowed. Generic code can involve arbitrary assignments where the type of the object we are assigning is a generic parameter, or function calls that pass arguments of generic parameter types, and hence can lead to potential duplication of `Owner<T>` structs if those generic parameters were to be instantiated to `Owner<T>`.

Effectively our frontend guarantees that `Owner<T>` is a *non-copyable* value type.

Ensuring shields are unique. Similarly to `Owner<T>`, `Shield<T>` should be a non-copyable value type. Consider the unsafe code fragment in the right part of Figure 4. By duplicating `sh` into `sh1` we kept a reference to the TLS slot that guarded our manual object past the point where that TLS slot stopped guarding the object, resulting in an unsafe access. For this reason, our C# language frontend enforces the same non-copyable value type conditions on `Shield<T>`, with a small improvement: a simple static analysis does allow a method to return by value a *fresh* `Shield<T>`. A fresh `Shield<T>` is one that is created within the scope of a method and is never stored or copied, but is allowed to be returned to the caller (and then freshness propagates). This improvement is useful to allow the line: `var sh = mobj.Defend();` above. That line – superficially – looks like a copy, but is actually innocuous. If we were not to have this improvement, our API would have to provide a `Defend()` method that accepted a `ref Shield<T>` argument.

Ensuring shields are stack-only. Shields cannot be stored on the shared heap. The reason is that a shield has a meaning only for the thread that created it – it holds a pointer to a TLS slot. By placing shields on the shared heap, we generate possibilities for races on state that was supposed to be thread-local.

Ensuring manual objects don't escape past their shields' lifetimes. Consider the unsafe fragment below:

```
var sh = owner.Defend();
Object mobj = sh.Value;
... use mobj here ...   // safe
foo.f = mobj;           // unsafe (heap escape)
sh.Dispose();
... use mobj here ...   // unsafe (use after shield dispose)
```

```
internal class CacheEntry : ICacheEntry {
 ...
 Owner<object> m_Value;
}

class MemoryCache : IMemoryCache {
  ...
  object Set(object key, ref Owner<object> value, MemoryCacheEntryOptions
       options) {...}
  bool TryGetValueShield(object key, ref Shield<object> result) {...}
  bool RemoveEntry(CacheEntry entry) {
    ...
    entry.m_Value.Abandon();
} }
```

Fig. 5. Memory cache API modification for Snowflake

Here, we've created a shield from `owner`, and we are allowed to access the underlying manual object `mobj`. However, we must ensure that the object does not escape onto the heap because this can result in an access after the local shield has been disposed of and the object has been deallocated. We must also ensure that we do not access the object locally after the shield has been disposed.

Our frontend enforces those restrictions with a conservative dataflow analysis that has proven sufficient for our examples. Whereas the escape past the lifetime of the shield is easier and more local to detect, the heap escape analysis can become more involved as we need to descend through potentially deep hierarchies of method calls. For this reason we also have experimented with an alternative implementation that enforces this restriction uniformly by throwing exceptions in the write barriers if the object we are storing belongs in the virtual address range of the manual heap. We have measured the impact to performance of this check to be negligible because this path is extremely rare and does not involve another memory access.

3.5 Examples of Snowflake In Action

We present here some examples of how Snowflake can be used to offload objects to the manual heap, safely.

Lists with manual spines. The C# collection library defines a `List<T>` collection for lists of `T` elements. Internally it uses an *array* `T[]` to store those elements, which gets appropriately resized when more space is needed. This array can grow quite large and persist many collections, but is completely internal to the `List<T>` object, and hence it is an ideal candidate for moving it to the manually managed heap. Here is the original (left) and the modified (right) code:

```
class List<T>{ T[] _items;  ...}     |     class List<T>{  Owner<T[]> _items;  ...}
```

Multiple methods of `List<T>` use `_items` and each requires modification. Here is the original `Find` method (slightly simplified for space) that finds the first element that matches a predicate `match` or returns a default value. To port this to our new definition for `_items` we have to obtain a shield on the owner struct.

```
T Find(Predicate<T> match) {
  for (int i = 0; i < _size; i++) {
    if (match(_items[i]))
      return _items[i];
  }
  return default(T);
}
```

```
T Find(Predicate<T> match) {
  using (Shield<T[]> s_items =
           _items.Defend()){
    for (int i = 0; i < _size; i++) {
      if (match(s_items.Value[i]))
        return s_items.Value[i];
  } }
  return default(T);
}
```

The **using** construct automatically inserts the `Dispose()` method on a `IDisposable` object (such as our `Shield<T>`) in ordinary and exceptional return paths and we use it as convenient syntactic sugar. Inside the **using** scope we can access `s_items.Value`, but it must not escape on the heap nor be used past the **using** scope.

Note though, that since the new `List<T>` has a manual spine, when it is no longer needed programmers have to explicitly deallocate it using the following method:

```
void Abandon() { _items.Abandon(); }
```

This is the only new method we added to the `List<T>` API. Finally note that `List<T>` itself can be allocated on the GC heap or on the manual heap. We stress that having owner structs inside GC objects is not problematic for memory safety. Whereas the GC object is shared by many threads, our API ensures there is still a unique owner into the manual spine of the list, the one that is stored inside that GC object. In addition to this distinguished pointer, many shield-registered stack references to the manual spine can also exist.

Moving ownership in lists with manual spines. Occasionally the internal manually allocated array of a list must be resized to increase its capacity. Here is how we can do that:

```
var new_items = new Owner<T[]>;
ManualHeap.CreateArray(ref new_items, new_size);
using (var s_items = _items.Defend(),
           s_new_items = new_items.Defend()) {
  Array.Copy(s_items.Value, 0,
             s_new_items.Value, 0, _size);
}
_items.Move(ref new_items);
```

We first allocate `new_items`, a new `Owner<T[]>` using our `ManualHeap.CreateArray<T>()` method. Once that is done, we obtain shields to both the old and new items, and copy over the contents of the old items to the new array. Finally, we transfer ownership of the `new_items` to `_items`, which schedules the original manual object for deallocation.

Collections of owner objects. `List<T>` is an example where the spine of a data structure can be moved over to the manual heap, but ASP.Net caching actually does store long lived data. For this reason we may port the `m_Value` field to be an owner of a manually allocated object. For convenience we will keep the dictionary spine on the GC heap. We first need to change the `CacheEntry` and the interface to the memory cache, as shown in Figure 5.

Method `Set()` now accepts a reference to a locally created owner value, and will `Move()` it in to the cache if an entry is found with the same key or will create a fresh entry for it. Method `TryGetValueShield()` is passed in a shield reference, and uses *it* to protect an object (if found) against deallocation. The client code can then just access the object through that shield, if `TryGetValueShield()` returns **true**. Finally `RemoveEntry()` abandons the owner, scheduling it for deallocation once it is no longer shielded. The client code can be as follows:

```
using (var res_sh = Shield<object>.Create()) {
   if (!_cache.TryGetValueShield(key, ref res_sh)) {
         Owner<byte[]> tmp;
         ManualHeap.CreateArray(ref tmp, size);
         res_sh.Defend(ref tmp);
         ... // populate tmp, through res_sh
         _cache.Set(key, ref tmp, _options);
   }
   ... // use res_sh to complete the request
}
```

First we create a shield that will protect manual objects against deallocation throughout the request. We create it uninitialized to start with, so it does not protect any object. Subsequently we try to make this new shield defend the object that this key maps to in the cache, if such a cache entry exists. If we do not find it in the cache, we create a new local owner and allocate something, and use the shield to protect it and finally exchange it into the cache. In the rest of the code we can access the manual object (coming from the cache or freshly allocated) through `res_sh`.

A set of simple changes in an application can offload many objects that survive into older generations to the manual heap and result in significant speedups and peak working set savings as we will see in Section 5.

4 IMPLEMENTATION

Next we describe the key parts of our implementation. We have extended CoreCLR with the shield runtime, integrated a modified version of jemalloc to manage the physical memory for the manual objects, and added interopability with the GC to allow the manual heap to be scanned for roots.

4.1 Shield Runtime

Our approach to implementing `Shield<T>` combines ideas from both hazard pointers [Michael 2004] and epoch-based reclamation [Bacon et al. 2001; Fraser 2004; Harris 2001]. We provide a comparison in the related work, and just explain our implementation here.

The core responsibility of the shield runtime is to safely enable access to (and deletion of) manually managed objects, without requiring expensive synchronisation on accessing an object (a "read barrier"). To achieve this we allow reclaiming manual objects to be delayed.

Each thread holds a thread-local array of slots, each of which protects a manual object (or 0x01 for an unused slot). A `Shield<T>` struct then holds a pointer (`IntPtr` in C#) capturing the address of the thread-local slot (`slot` field) that this shield is referring to. The TLS slot stores the address of the manual object. The same address – for efficient access that avoids TLS indirections – is also cached as a field (`value`) of type `T` inside `Shield<T>` and is what the `Value` property returns. Allocation of shields amounts to finding an unused TLS array slot and creating a shield that holds a pointer to that yet uninitialized slot. A call to `v.Defend(ref x.o)` is rewritten to `v.value = SetShield(ref v.slot, ref x.o)` where `SetShield()` simply sets the TLS slot to contain the address of the manual object and returns it.

For abandoning a manual object, we effectively do

```
void Abandon(ref Owner<T> o) {
    var x = InterlockedExchange(o, null);
    if(x != null) AddToDeleteList(x);
}
```

where `AddToDeleteList` adds the object to a thread local list of objects that need to be reclaimed. The call to `InterlockedExchange` is a compiler intrinsic that sets a location to the specified value as an atomic operation and returns the old value.[1]

Occasionally, when the objects in the list consume too much space we trigger `Empty()` to reclaim memory:

```
void Empty() {
  for (var i in DeleteList)
    if(IsNotShielded(i)) free(i.ptr);
}
```

`IsNotShielded()` needs to check that *every thread's* TLS shield array does *not contain* the object. However, during that check some thread may be calling `SetShield`. Correctness is predicated on the correct interaction between `IsNotShielded` and `SetShield`.

Naïve approach to synchronization. One way to solve the synchronization problem is to exploit the "stop-the-world" GC synchonization for collecting the roots. GC suspends all threads (which also causes all threads to flush their write buffers so that all memory writes are globally visible). At that point we can iterate through every thread and call `Empty()` on the thread-local `DeleteList`.

To make `Empty()` efficient, we first iterate through all threads and build a Bloom filter [Bloom 1970] that contains all shields of all threads. We use that Bloom filter to over-approximate the `IsNotShielded` check. This avoids the need to repeatedly check with the local shields of each thread.

However, there is a danger if we allow mutator threads to be suspended *during* a `SetShield()` call. In C notation, `SetShield()` executes the following:

```
Object* SetShield(Object** slot, Object** fld) {  *slot = *fld; return *slot; }
```

If a thread is suspended for GC right after reading `*fld;` but *before* publishing the object in the shield `slot` and another thread has already issued an `Abandon()` on the same manual object we can get into trouble: a call to `Empty()` from the GC thread will not see the object in the shield TLS array of the first thread and may deallocate it. When threads resume execution, the first thread will publish the address of a (now deallocated) object and continue as normal, leading to a safety violation. For this reason we prevent a thread that executes `SetShield()` from being suspended. This is supported in CoreCLR by executing `SetShield()` in "cooperative mode". This mode sets a flag that forces the runtime to back out of suspending the thread.

We have solved the problem by effectively forbidding calls to `IsNotShielded()` from occuring during `SetShield()`. However, calling `Empty()` this way comes at the cost of suspending the runtime. We want to maintain eager deallocation of manual objects, and avoid suspending the runtime too often, so we only use this approach if a GC is going to occur. Next, we develop another mechanism that does not require suspending any threads, and allows `Empty()` to be called independently by any mutator.

Epochs for concurrent reclamation. To enable more concurrency for manual object collection, we use epochs to synchronise the threads' view of shields, drawing from work on reference counted garbage collectors [Bacon et al. 2001]. Epochs allow thread-local collections without stopping any threads or requiring expensive barriers in the `Defend` code.

We use a 64-bit integer for a global epoch. Each thread has a local epoch that tracks the global. Adding an object to the `DeleteList` may trigger advancement of both this thread's local and/or the global epoch. If the running thread detects that it is lagging behind the global epoch, it just sets

[1]We use the Microsoft VC++ intrinsics – in `gcc` this would be `__sync_lock_test_and_set (o, null)`.

$$\begin{tikzcd}
L(0) \arrow[r, dotted] \arrow[dr, dotted] & A(o_1); A(o_2) \arrow[r, red] & L(1) \arrow[rr, dotted] \arrow[dr, red] & & L(2) \arrow[rr, dotted] \arrow[dr, dotted] & & L(3) \arrow[r, blue] & R(o_1) \text{ fails} \\
& G(1) \arrow[ur, dotted] \arrow[rr, dotted] \arrow[dr, dotted] & & G(2) \arrow[ur, dotted] \arrow[rr, dotted] \arrow[dr, red] & & G(3) \arrow[ur, blue] \arrow[rr, dotted] \arrow[dr, dotted] & & \ldots \\
L(0) \arrow[ur, dotted] \arrow[rr, dotted] & & L(1) \arrow[ur, dotted] \arrow[r, dotted] & D(o_1) \arrow[r, blue] & L(2) \arrow[ur, blue] \arrow[r, red] & D(o_2) \text{ fails} \arrow[r, dotted] & \ldots &
\end{tikzcd}$$

Fig. 6. Happens-before ordering of global (G), local (L) epoch events, abandon (A), defend (D), and reclaim (R) operations. Thread 1 in the top line, Thread 2 execution in the bottom line.

its local epoch to be equal to the global epoch. If the running thread detects that all threads agree on their local epochs – and epoch advancement is heuristically sensible – then it performs a CAS that sets the global epoch to be the agreed upon epoch plus one. It is okay for the CAS to fail, as that means another thread has advanced the global epoch.

This protocol guarantees that the global epoch is *never* more than one ahead of any local epoch. In Figure 6 we illustrate the ordering of such epoch events (arrows denote causal happens-before relations). $L(n)$ signifies a thread writing to its local epoch the value n; and $G(n)$ signifies the global epoch advancing to n. To advance the global epoch from n to $n + 1$, every thread's local epoch must be in epoch n.

When we add an object to the `DeleteList` we record which local epoch it was deleted in. Occasionally, mutators will call `Empty()`, and create the Bloom filter of all threads' shields, without stopping any thread. Unfortunately not all writes to the shield TLS may have hit the main memory so there is a possibility that an object will be residing in the `DeleteList` with its corresponding `SetShield()` write effects still in flight; we cannot actually deallocate those objects. However, all the objects in the `DeleteList` whose recorded epoch count is less than three epochs behind the local epoch, if they are not in the Bloom filter, are actually safe to deallocate.

We illustrate why three epochs is enough in Figure 6. First we define 3 types of *logical event*: *Abandon events*, $A(o)$ denote the exchange of the owner with *null* and scheduling a pending deallocation; *Defend events*, $D(o)$ denote the combined effects of `SetShield()` – that is reading of the object and storing it in the TLS shield array; and *Reclaim events*, $R(o)$ denote successful object deallocation. Assume that in epoch 0, Thead 1 abandons both o_1 and o_2 ($A(o_1); A(o_2)$). We have several possibilities for Thread 2:

- Assume a defend event $D(o_1)$ in epoch 1. The effects of this event (in particular the write to the TLS slot) are guaranteed to be visible by epoch 3 in Thread 1, as the blue solid arrows indicate. This ordering on x86 comes as part of the memory model; for ARM and Power barriers are required in the global and local epoch updates.
- Assume Thread 2 attempts to defend an object o_2 in a later epoch, epoch 2 ($D(o_2)$). However, as the red solid arrows indicate, the abandon event of o_2 ($A(o_2)$) must have become visible by now and hence we will be defending `null` – no violation of safety.
- If Thread 2 defends one of o_1 and o_2 earlier (e.g. in $L(0)$) then trivially less than three epochs suffice (not depicted in the diagram).

We conclude that in all cases we need only wait three epochs to safely delete an object if it is not contained in some TLS shield array. Effectively, and taking into account that `IsNotShielded()` may also be called with the runtime suspended, our `IsNotShielded` code becomes:

```
bool IsNotShielded (Node i) {
  return (runtimeIsSuspended || (i.epoch + 3 <= local_epoch) )
   &&  ... ;//check per thread shields
}
```

To efficiently represent the epoch an object was abandoned in, we use a cyclic ring buffer segmented into four partitions: three for the most recent epochs and one for the spare capacity.

Finally notice that our reasoning relies on $A(\cdot)$ and $D(\cdot)$ events being atomic with respect to local epoch advancement. This holds as each thread is responsible for advancing its own local epoch.

Protocol ejection for liveness. A thread being responsible for advancing its own epoch can lead to liveness problems. If a thread blocks, goes into unmanaged code or goes into a tight computation loop, it can hold up the deallocation of objects. To solve this problem we introduce an additional mechanism to *eject* threads from the epoch protocol. If Thread A has too many objects scheduled for deallocation, and Thread B is holding up the global epoch, then Thread A will *eject* Thread B from the epoch consensus protocol and ignore its local epoch; Thread B must rejoin when it next attempts to use `SetShield()`.

Each thread has a lock that guards the ejection and rejoining process. When Thread A ejects Thread B, it makes the TLS shield array of Thread B read-only using memory protection (`VirtualProtect` in Windows, `mprotect` in Linux). It then marks Thread B's epoch as a special value `EJECTED` which allows the global epoch advancing check to ignore Thread B. As multiple threads may be simultaneously trying to eject Thread B, the ejection lock guarantees that only one will succeed (`TryAcquireEjectionLock()`).

```
void Eject(Thread *other) {
  if (other->TryAcquireEjectionLock()) {
    VirtualProtect(other->Shields,
                             READONLY);
    other->local_epoch = EJECTED;
    other->ReleaseEjectionLock();
}}
```

```
void Thread::Rejoin() {
  this->AcquireEjectionLock();
  VirtualProtect(this->Shields,
                       READWRITE);
  this->local_epoch = global_epoch;
  this->ReleaseEjectionLock();
}
```

Note that we *must* rejoin the protocol if we are to use shields, hence we must wait to acquire the ejection lock (`AcquireEjectionLock()`). We can then un-protect the TLS pages that hold the shield array, and set our local epoch back to a valid value to rejoin the consensus protocol.

```
void AddToDeleteList(Object *o) {
    Epoch curr = local_epoch;
    if (curr == EJECTED) curr =
        global_epoch;
    DeleteList->push(o, curr);
    ... // possibly call Empty()
}
```

To handle the ejection mechanism, we must adapt `Abandon()` slightly. Recall that `Abandon()` first exchanges **`null`** in the owner, and then calls `AddToDeleteList()`, with the current local epoch. However, due to ejection, the local epoch may be `EJECTED`. So if the thread is ejected, we use the global epoch. Note that for the argument in Figure 6 to be correct, it actually suffices that the epoch used to insert the object in the `DeleteList` be at least $(g - 1)$ where g was the global epoch at the point of the atomic exchange of **`null`** in the owner. If ejection happened between the exchange and `AddToDeleteList` then the new global epoch that we will read is guaranteed to be at least g.

The TLS shield array of an ejected thread will be read-only, hence we guarantee that any call to `SetShield()` from an ejected thread will receive an access violation (AV). We trap this AV, rejoin the protocol with `Rejoin()`, and then replay the `SetShield()` code. By replaying the `SetShield()` code after `Rejoin()` we make it atomic with respect to ejection, and thus local epoch advancement, as required earlier. You can view memory protection as an asynchronous interrupt that is only triggered if the thread resumes using shields.

4.2 GC Interoperability and Jemalloc

The core changes to the GC and jemalloc are: (1) provide a cheap test if an object is in the manual or the GC heap; (2) extend the GC card table [Jones et al. 2011] to cover the manual heap; and (3) allow iteration of GC roots from the manual heap.

We modify the OS virtual memory allocation calls both in the GC and jemalloc to use a threshold: 2^{46}. The GC allocates in pages directly above this, and jemalloc directly below. This allows for a cheap test to determine in which heap an object resides without any memory access. As CoreCLR has a generational GC, we need a card table to track pointers from the manual heap into Gen0 and Gen1. By growing both heaps away from a threshold, the used virtual address space is contiguous among the two heaps. Hence, we use a single card table to cover both heaps and we do not modify the write barriers. When jemalloc requests pages from the OS we notify the GC so that it can grow the card table. This requires delicate synchronization to avoid deadlocks.

Finally, we have implemented a trie that covers the manual heap and contains a bit for each 64bit word to represent if it is the start of an object that contains GC heap references. We iterate this trie when we need to find roots in the manual heap during a garbage collection. This trie is used in conjunction with the card table for Gen0 and Gen1 collections, and iterates all objects for a full Gen2 collection.

5 EVALUATION

We performed experiments to validate the the impact of our mixed-mode memory management approach to application runtime and peak working sets (PWS), as well as scalability on threads and heap sizes. To measure the PWS of a process we directly query the OS. Specifically, we query the "Working Set Peak" Windows Performance Counter for the process that we are testing, which includes both GC heap memory, manual heap memory, runtime metadata etc.

We have ported three industrial benchmarks from data analytics (`System.Linq.Manual` used to implement TPCH queries), caching (ASP.NET Caching), and machine learning (probabilistic automata operations on top of the Infer.NET framework [Minka et al. 2014]). We also present a small set of data structure specific micro-benchmarks on trees and graphs. The benchmarks were written originally in unmodified .NET but we ported some of the allocations to the manual heap based on profiling. For some benchmarks this was as easy as writing a few lines of code (e.g. ASP.NET caching, `System.Linq.Manual`), for some others it was more tedious due to the "by ref" style of owners and shields.

Our results generally demonstrate (i) better scalability of both runtime and PWS than the purely-GC versions, (ii) savings in PWS, though (iii) throughput results can be mixed, depending on the benchmark.

Experimental Setup. We performed all experiments on a 3.5GHz Intel Xeon CPU E5-1620 v3 (8 physical cores) with 16GB RAM running Windows 10 Enterprise (64-bit). We used our own branch of CoreCLR, CoreFX and jemalloc. We configured the CoreCLR in "Workstation GC" mode. Note that the CoreCLR GC does not give programmers control over generation/heap size so we do not experiment with space-time tradeoffs in garbage collection. However, CoreCLR additionally provides a "Server GC" mode, which allows for independent thread-local collections. We have only run preliminary results against this configuration and the relative trends are similar to our findings with Workstation GC, but the absolute numbers show that Server GC trades higher PWS for lower times to completion. We refer to the extended version of this paper for this data [Parkinson et al. 2017].

Table 1. GC Collections and pauses in ASP.NET Caching benchmark (8 threads). Mean and Max times are in milliseconds.

Config	Mode	%GC	#Gen0	Mean	Max	#Gen1	Mean	Max	#Gen2	Mean	Max
256-byte	GC	26.5%	530	5.2	15.3	197	16.2	31.5	21	27.5	181.8
256-byte	M+GC	18.1%	349	5.4	9.9	185	11.3	25.9	17	17.6	133.6
2048-byte	GC	32.1%	1333	2.0	6.8	698	8.5	74.4	37	40.8	321.6
2048-byte	M+GC	17.6%	370	5.4	14.8	201	10.9	26.1	20	21.6	143.3
4096-byte	GC	34.5%	2481	1.4	7.4	1387	7.0	67.7	46	43.5	409.8
4096-byte	M+GC	14.9%	397	5.3	15.0	199	11.4	27.2	27	17.1	132.8

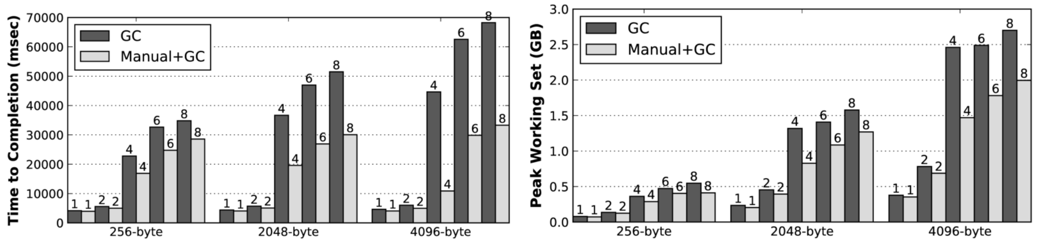

Fig. 7. Results for ASP.NET Caching.

ASP.NET Caching. In Section 3.5 we have presented a modification to a caching middleware component from ASP.NET. We have created a benchmark where a number of threads perform 4 million requests on a *shared* cache. Each request generates a key uniformly at random from a large key space and attempts to get the associated value from the cache. If the entry does not exist then a new one is allocated. We show here experiments with small (256 bytes), medium (2K), or large (4K) payloads and a sliding expiration of 1sec.

Figure 7 show the results. Thread configurations are labeled with a number on each bar. Both time to completion and PWS savings compared to the purely GC version improve substantially, particularly with bulkier allocations and multiple threads. Characteristically, time to completion halves with 8 threads and 4k cache payloads, whereas peak working sets improve up to approximately 25%.

The key reason for the massive throughput speedups for this benchmark is that by off-loading the cache payloads to the manual heap, a lot of write barriers are eliminated entirely, and at the same time fewer GC collections are triggered (Table 1). For the 2048 and 4096 payload sizes, there are approximately half the Gen2 collections and $\frac{1}{6}$th of the Gen0 and Gen1 collections. There are still quite a few garbage collections remaining with the manual memory modifications, basically because the cache stores GC-based `CacheEntry` objects (which *in turn* store the actual manual payloads), and it is the allocation of those objects that triggers collections. Note also that the cost of collections for Gen0 and Gen1 is higher with manual memory, Gen0 goes from 1.4ms to 5ms with manual memory. This is possibly because our modifications have taken the bulky payload objects out of the ephemeral (Gen0 and Gen1) segments, and hence the ephemeral segments are now full with (many more) small `CacheEntry` objects. Ephemeral collections simply have to collect more objects and take more time. However, since in the purely GC version Gen0 collections collect fewer objects, more pressure is put on Gen2 collections. We can see this as the purely GC code takes 43.5ms for a Gen2 collection, where as with manual heap it is just 17.1ms (for the 4096-byte configuration).

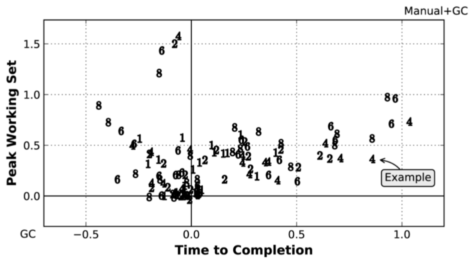

Fig. 8. PWS and runtime for TPCH queries.

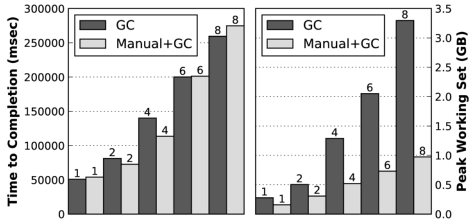

Fig. 9. PWS and runtime for the Infer.NET benchmark.

TPCH on `System.Linq.Manual`. Here we introduced a drop-in replacement for `System.Linq` (namespace `System.Linq.Manual`), which is a popular C# library for data processing in SQL-like syntax. It is designed to allocate little, but some operators must materialize collections, namely *group*, *join*, and *order-by* variants, as the example of a join iterator from Section 2 explained. We have moved these transient collections to the manual heap, and introduced deallocations when they were no longer needed. For evaluation we used 22 industry-standard TPCH queries [TPC.org 2017], ported them in LINQ C#, and ran them using `System.Linq` and `System.Linq.Manual`. We have scaled the TPCH dataset to a version where 300MB of tables are pre-loaded in memory, and another where 3GB of tables are pre-loaded in memory. We present here the results from the small dataset (the results are similar for the large dataset in the extended version of this paper [Parkinson et al. 2017]). To evaluate scalability in multiple threads we run each query in 10 iterations using 1,2,4,6,8 *independent* threads sharing the same pre-loaded dataset.

For reasons of space Figure 8 presents all 22x5 results as a scatter plot where the vertical (resp. horizontal) axis shows *relative* improvement in the peak working sets (resp. runtime). The labeling indicates number of threads. Positive numbers in both axes mean that we are faster and consume less memory. For example, the number 4 labeled "Example" in the figure indicates that on one query with 4 threads, the purely GC code used ~80% more time, and ~40% more memory. The majority of queries improve in PWS, some up to 1.5, meaning that the purely GC version uses 150% more memory than our modified version. For runtime the results are mixed, though many queries achieve significantly better runtime. Profiling showed that this is primarily due to the cost of TLS accesses during the creation and defending with shields. GC statistics that can be found in the extended version of this paper [Parkinson et al. 2017] reveal that we generally reduce the number and pauses of collections. For some queries though Gen2 pauses become higher – this may have to do with the more expensive scanning of the trie for GC roots, especially for *background* collections. We discuss this issue further in Section 6.

Machine learning on Infer.NET. We used a workload from a research project with known GC overheads based on the Infer.NET framework. The task was to compute the product of two large probabilistic automata for a small number of iterations. We converted various collections of automata states to use manual "spines" but kept the actual automata states in the GC heap to avoid too intrusive re-engineering. To avoid TLS access during iteration of these collections, a significant fragment of the product construction code was rewritten to re-use pre-allocated shields that were passed down the stack as by-ref parameters. This was a more invasive change, than those required in previous benchmarks. As with the TPCH queries, to understand scalability with multiple threads we run the tasks with 1, 2, 4, 6 and 8 independent threads. Figure 9 suggests that our modifications did not improve the runtime, in fact they did have a small negative effect (due to excessive use of shields). However, due to the immediate deallocation of the various intermediate collections we

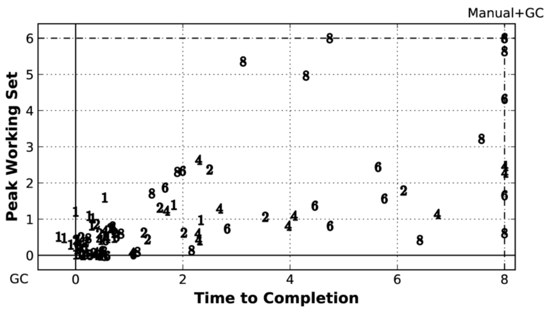

Fig. 10. PWS and runtime for micro-benchmarks.

were able to almost halve the peak working set in 1 thread, and get approximately 3x improvement when we scaled up this benchmark to 8 threads.

CoreCLR micro-benchmarks. Apart from the benchmarks discussed above, we also evaluated our mixed-mode memory management approach on 6 data structure specific micro-benchmarks adapted from the performance testsuite of CoreCLR: BinTree, BinTreeLive, BinTreeGrow, RedBlackTree, DirectedGraph and DirectedGraphReplace. Each of these was configured to run with 3 different input sizes (small, medium and large) and 1, 2, 4, 6 and 8 independent threads. Each benchmark involves typical operations with allocations on an underlying container. Details of each benchmark can be found in the extended version [Parkinson et al. 2017]. Figure 10 plots all our data points in all configurations, and shows massive gains both in runtime and memory that were obtained by porting selectively parts of these data structures to use the manual heap.

We collapse to the border the datapoints that are outside our bounding boxes.

6 DISCUSSION

6.1 Frontend

We have built a C# language front-end that guarantees the safe use of our API, enforces the correct use of owners and shields, and performs an escape analysis to ensure that manual objects do not escape their shields. It additionally makes programming easier, for example it adds implicit `Dispose()` calls for stack-based shields, implicit `Defend()` calls as coercions from owners to shields etc. However the focus of this paper is the runtime and hence we defer any source language modifications for presentation in future work.

6.2 Performance Considerations

Cost of shield allocation and setting. Shield allocation and setting is a thread-local operation (hence no interlocked barriers are needed); still frequent TLS access can have an adverse affect on performance, hence shield reuse and elimination of redundant shield setting can have beneficial effects. In future work we would like to explore tighter JIT integration for shield allocation, as well as analysis for thread-local objects for which cheaper synchronization can be used.

Allocator characteristics. Although jemalloc is a fast and scalable allocator with thread-local caches, it is difficult to match the throughput of the bump-allocator of the .NET generational GC. The situation reverses once surviving objects start to get stored in long-lived and growing collections. In some experiments that allocate a lot but survive some objects, we have found that it is beneficial to use the GC as nursery and only when we need to store an object *clone* it over to the manual heap. Despite the throughput disadvantage, using a size-class based allocator has the advantage of low fragmentation which means we can avoid compaction (fragmentation and compaction in older generations is a known issue in generational GCs.)

6.3 GC Integration

We highlight here some subtle performance aspects of the GC integration.

In GC allocation-intensive workloads that have numerous pointers from the manual to the GC heap, the collections will have to scan through portions of the manual heap. For ephemeral collections this amounts to a simultaneous iteration over the set cards for the manual heap and the trie, which becomes a hot code path. We have evaluated the cost of simultaneous card and trie scanning by designing the following experiment: First we allocate a big tree of small node objects, where nodes also store (initially **`null`**) pointers and ensure that the tree is in the oldest generation (by calling `GC.Collect(2)`). We then repeatedly run a big number of iterations: in every iteration we allocate a single (young) object and store it in a random subset of the tree nodes so that the cards corresponding to these nodes are set by the write barriers, and continue to call `GC.Collect(1)`. We measure the cost of that Gen1 collection, which will be based on the set cards for the Gen0 and Gen1 generations, but will do almost zero work aside from traversing those cards, because there is only one object to promote. We repeat the experiment but where the tree is allocated in its entirety in the manual heap. In that case, `GC.Collect(1)` will have the effect of using the trie and the cards to determine roots. Hence, measuring the cost of this call across many iterations gives us a precise account of the cost of trie-card scanning versus GC segment-card scanning. We find that when stores are sparse (1/500 tree nodes) – and so cards are sparsely set – then scanning through the trie is actually approximately 13% *faster* because a segment of the trie is loaded in the cache, whereas GC segment iteration loads method table pointers for consecutive objects to figure out the size of the object and move to the next object. When stores are denser (already at 1/50 tree nodes) scanning through the trie becomes approximately 10% slower compared to GC segment scanning, and when stores are very dense (1/5 tree nodes) then trie scanning is approximately 20% slower compared to GC segment scanning. That is because in these cases the cost is dominated by the bit manipulation that trie scanning involves. In a program with less than 15% of time spent in GC (already a pretty high number), these differences should be negligible.

CoreCLR GC employs concurrent background marking but our implementation at the moment only scans roots synchronously from the manual heap; addressing this is a matter of engineering but explains some of the larger Gen2 pauses in some benchmarks.

When bulky objects are off-loaded to the manual heap, the ephemeral segment budget will fill up with more and smaller objects, potentially increasing the pauses for Gen0 collections. This is often not a problem as the number of Gen0 collections is dramatically reduced but it reveals an opportunitiy to tune the various parameters of the GC to take the manual heap into account.

Another interesting phenomenon happens when we resize a manual array with GC references to ephemeral objects – if that array was GC then the newly allocated array would be allocated in Gen0 hence no card setting would be required. However in our case, we have references from the manual heap to ephemeral objects and hence card setting is required. Although we have observed this to happen, we have not seen any major performance problems, as the card range is much smaller than the array itself.

6.4 Migrating Away from the GC Heap

How does a programmer know which objects should be allocated on the manual heap? Our methodology is to use a heap profiling tool (e.g. we have modified `PerfView`, a publicly available tool from Microsoft), to determine if the GC has a significant cost, then identify objects of reasonable size that have survived and have been collected in the older generations, get the stack traces associated with such objects and look for sensible candidates amongst those with clearly defined lifetimes.

6.5 Programming Model

By-ref style of owners and shields. Owners and shields cannot be duplicated on the stack or returned from arbitrary functions, and hence can only be passed "by-ref", complicating the porting of an application. For some applications this is not a problem (e.g. `System.Linq.Manual`) but for others it may require redesign of APIs and implementation.

Sharing references on the heap. A well known limitation of owner-style objects (and unique references in general) is that sharing of multiple references to the same manual object from the (GC or manual) heap is not allowed, thus making them unsuitable for data structures with a lot of pointer sharing. Snowflake allows shareable pointers to be built around owners. For instance, we have introduced a *class* called `Manual<T>` (as opposed to a struct), that encapsulates an `Owner<T>` and supports similar methods but can be stored on the heap and passed to or returned by other functions like every other GC object. In exchange for greater programming flexibility, object `Manual<T>` incurs extra space cost per manual object, so calls for a careful profile-driven use. Finally, we have also built shareable reference counted objects, `RefCount<T>`, but we are considering API extensions in this space as important future work.

Shields on the heap. Shields may accidentally escape on the (shared) heap, resulting in complaints in our C# frontend, predominantly due to the subtle C# *implicit boxing* feature or closure capture. For other examples, we may actually prefer to temporarily store a shield on the heap, as long as we can guarantee that only the very same thread that created that shield will access it and the underlying object (otherwise the reference to the TLS state is bogus). How to enable this without sacrificing performance is an open challenging problem.

Finalization and owners. When owner objects get abandoned our runtime abandons recursively their children owner fields. To do this efficiently we extend the method table with an explicit run-length encoding of the offsets that the owner fields exist in an object and use that to efficiently scan and abandon. An avenue for future work is to modify the layout of objects to group together owner fields to improve this scanning, similarly to what CoreCLR is using for GC pointers. .NET also allows for *finalizers*, which are functions that can be called from a separate thread when objects are no longer live. There is design space to be explored around finalizers for manual objects, e.g. should they be executed by the mutator or scheduled on the GC finalization thread.

Asynchronous tasks. In the *async and await* [MSDN 2016] popular C# programming model a thread may produce a value and yield to the scheduler by enqueuing its continuation for later execution. This continuation may be eventually executed *on a different thread.* This decoupling of tasks from threads makes the use of the thread-local shields challenging and it is an open problem of how to allow a task that resumes on a different thread to safely use a shield.

7 RELATED WORK

Several previous studies compared garbage collection with manual memory management [Hertz and Berger 2005; Hundt 2011; Zorn 1993]. Like us, they concluded that manual memory management can achieve substantial gains in both memory usage and run time, particularly when the heap size is not allowed to grow arbitrarily.

Memory management for managed languages. Several systems have proposed optimizing garbage collection for specific tasks – for instance for big data systems [Gidra et al. 2015; Maas et al.

2016], taking advantage of idle mutator time [Degenbaev et al. 2016], specializing to real-time and embedded systems with very low latency constraints [Bacon et al. 2003]. Other work suggests arena-based allocation for moving bulky data out of the GC heap. Scala off-heap [github.com/densh 2017] provides a mechanism to offload all allocations in a given scope onto the unmanaged heap but no full temporal and thread safety. Stancu et al. [2015] introduce a static analysis that can infer a hierarchy of regions and annotate allocation sites so that at runtime a stack-like discipline of regions can be enforced. The analysis ensures that pointers only exist from newer regions to older regions or the GC heap but not vice verca. Broom [Gog et al. 2015] introduces region-based allocation contexts, but relies on type system for safety. Other work attempts to offload data allocations to the manual heap through program transformations [Nguyen et al. 2015]. Recent work proposes hints for safe arena allocations [Nguyen et al. 2016], through more expensive write barriers and potential migration of objects when programmer hints are wrong. Our proposal is complementary to these other techniques, and should be seen as yet another tool available to programmers that guarantees memory safety in single-and multithreaded scenarios. For instance, allocating the intermediate dictionaries from our `System.Linq` example in an arena would be cumbersome and less efficient as those allocations are intertwined with other allocations of objects that need to survive, so copying out of the arenas would be required.

Kedia et al. [2017] propose a version of .NET with only manually managed memory. Their programming model just exposes a "free object" method, and a dangling pointer exception for accessing reclaimed memory. The runtime reclaims physical pages associated to an object at some point after they are "freed", and relocates other collocated objects. When an object is accessed an access violation can be triggered due to the object been deallocated or relocated: the runtime determines which and either surfaces the exception, or patches the memory and execution to allow the program to continue. The approach gets surprisingly good results (comparable to the results we report here). The approach has not been integrated with the garbage collector and thus does not provide the pay-for-play cost of our solution. The exceptions for accessing deallocated objects are unpredictable, and dependent on the runtime decisions for when to reclaim memory: some schedule can work by allowing access to a freed but not reclaimed object, while other schedules may not. Finally, their solution requires the JIT/Compiler to be modified to enable stack walking on every read and write to memory, whereas we do not require any JIT level changes.

Although .NET does not support it, manual or automatic object pre-tenuring [Cheng et al. 1998; Clifford et al. 2015; Harris 2000] would be another way to directly push long lived objects onto a less frequently scanned generation. It is certainly the case that a lot of the performance gains of Snowflake are related to not having to compact and promote between generations, and that is where pre-tenuring would also have a similar effect. However, the significant reduction in the number of Gen2 collections would not be given by pre-tenuring. Using Table 1 for the 4096-byte case we can measure total GC pause times of Gen0 = 3473.4, Gen1 = 9709, Gen2 = 2001, whereas Manual+GC are Gen0 = 2104.0, Gen1 = 2268, and Gen2 = 459. Pre-tenuring could potentially match the Gen 0/1 cost reduction but would not produce savings from not having to scan the older generation.

Safe memory reclaimation. Our shields are based on the hazard pointers [Michael 2004] mechanism developed for memory reclamation in lock-free data structures. Hazard pointers require the access to the data structure, in our case defend, to use a heavy write barrier (mfence on x86). We found this was prohibitively expensive for a general-purpose solution. By combining the hazard pointer concept with epochs we can remove the need for this barrier.

Epoch-based reclamation (EBR) has been used for lock-free data structures [Fraser 2004; Harris 2001; Hart et al. 2007], and has been very elegantly captured in the Rust Crossbeam library [docs.rs 2017]. To access a shared data structure you "pin" the global epoch which prevents it advancing (too much), and when you finish accessing the data structure you release the pin. As epochs advance the older deallocated objects can be reclaimed. We could not directly use epochs without hazards, as EBR requires there to be points where you are not accessing the protected data structure. When EBR is protecting a particularly highly-optimised data structure, this is perfectly sensible. We allow pervasive use of manually allocated objects and thus enforcing points where the manual heap is not being accessed is simply impractical. One could view shields as pinning a single object, rather than a global pin as in Crossbeam.

Our use of epochs is actually closer to those used in reference counted garbage collectors such as Recycler [Bacon et al. 2001]. Recycler delays decrements, such that they are guaranteed to be applied after any "concurrent" increments, thus if the reference count reaches zero, then the object can be reclaimed. We are similarly using epochs to ensure that the writes to shields will be propagated to all threads, before any attempt to return it to the underlying memory manager. It is fairly straighforward to extend our mechanism to handle reference counted ownership rather than unique ownership using delayed decrements. Recycler uses stack scanning to deal with stack references into the heap.

Alistarh et al. [2015] take the stack scanning approach further and use this instead of hazard pointers for a safe memory reclaimation. They use signals to stop each thread and scan its stack. These stack scans are checked before any object can be deallocated. We think our approach may scale better with more threads as we do not have to stop the threads to scan the stacks, but the sequential throughput would be lower for us as we have to perform the shield TLS assignment for what we are accessing.

There are several other schemes that use hazard pointers with another mechanism to remove the requirement for a memory barrier. Balmau et al. [2016] extend hazard pointers in a very similar way to our use of epochs. Rather than epoches they track that every thread has performed a context switch, and use this to ensure that the hazard pointers are sufficiently up to date. If the scheduling leads to poor performance, they drop back to a slow path that uses standard hazard pointers with a barrier. Our mechanism for ejection means we do not to have a slow path even when the scheduler starves particular threads, we believe our ejection mechanism could be added to their scheme. Dice et al. [2016] also develop a version of hazard pointers without a memory barrier. When the hazards need scanning, virtual memory protection is used to ensure the hazards are read only, which forces the correct ordering of the checking and the write. A mutator threads that attempts to assign a hazard will take an access violation (AV) and then block until the scan has finished. This means any reclamation is going to incurr two rounds of inter-processor interrupt (IPIs), and mutators are likely to experience an AV. We only use the write protection in very infrequent cases of threads miss behaving, and rarely have to deal with the AVs.

Morrison and Afek [2015] use time as a proxy for an epoch scheme: effectively they wait so many cycles, and have observed empirically that the memory updates will have propogated by this point. This leads to a simple design, no thread can stop time advancing, hence they do not require an Ejection mechanism like we have. However, to achieve this they make assumptions on the hardware that are currently not guaranteed by the vendors about timing.

There are other approachs to reclaimation that use data-structure specific fixup [Brown 2015] or roll-back [Cohen and Petrank 2015a,b] code to handle cases where deallocation occurs during an operation. This would not be practical for our setting as we are using manual memory management pervasively and do not have nice data structure boundaries that these schemes exploit.

Techniques for unsafe languages. Several systems use page-protection mechanism to add temporal safety to existing unsafe languages: [Dhurjati and Adve 2006; elinux.org 2015; Lvin et al. 2008; support.microsoft.com 2012] these approaches are either probabilistic or suffer from performance problems. Some systems propose weaker guarantees for safe manual memory management. Cling [Akritidis 2010] and [Dhurjati et al. 2003] allow reuse of objects having same type and alignment. DieHard(er) [Berger and Zorn 2006; Novark and Berger 2010] and Archipelago [Lvin et al. 2008] randomize allocations to make the application less vulnerable to memory attacks. Several systems detect accesses to freed objects [Lee et al. 2015; Nagarakatte et al. 2010; Younan 2015], but do not provide full type safety.

Type systems for manual memory management. The Cyclone language [Hicks et al. 2004; Swamy et al. 2006] is a safe dialect of C with conservative garbage collector [Boehm and Weiser 1988] and several forms of safe manual memory management, including stack and region-based allocation [Grossman et al. 2002; Tofte and Talpin 1997]. Capability types [Walker et al. 2000] can be used to verify the safety of region-based memory management. Unlike Cyclone, we do not use regions as a basis for safe manual memory management. Furthermore we allow eager deallocation at arbitrary program locations. Several languages have proposed using unique pointers to variables or objects [Hogg 1991; Minsky 1996; Naden et al. 2012] based on linear typing [Baker 1995; Wadler 1990]. Our owners are a form of unique pointers, but we allow stack sharing of the references using shields. This is similar to the concept of borrowing references to temporarily use them in specific lexical scopes [Boyland 2001; Clarke and Wrigstad 2003; Swamy et al. 2006; Walker and Watkins 2001]. Rust [rustlang.org 2017] incorporates several aspects of the Cyclone design, including integration of manually managed memory with reference counting, unique pointers, and lexically-scoped borrowing. Finally, languages with ownership types [Boyapati et al. 2003; Clarke and Wrigstad 2003; Clarke et al. 1998] and alias types [Smith et al. 2000] can express complex restrictions on the object graphs that a program can create. Though owners cannot express cycles, we do allow sharing through the GC, and permit cross-heap pointers in both directions.

8 CONCLUSION

We have presented a design that integrates safe manual memory management with garbage collection in the .NET runtime, based on owners and shields. Our programming model allows for stack-based sharing of owners potentially amongst multiple threads, as well as arbitrary deallocations while guaranteing safety. We show that this design allows programmers to mix-and-match GC and manually-allocated objects to achieve significant performance gains.

REFERENCES

Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers.. In *USENIX Security Symposium*. 177–192.

Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *SPAA*.

ASP.Net. 2017. ASP.Net/Caching: Libraries for in-memory caching and distributed caching. https://github.com/aspnet/Caching. (2017).

David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. 2001. Java Without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. *PLDI* (2001).

David F. Bacon, Perry Cheng, and V.T. Rajan. 2003. The Metronome: A Simpler Approach to Garbage Collection in Real-time Systems. In *In Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*.

Henry G. Baker. 1995. Use-once variables and linear objects–storage management, reflection, and multi-threading. *SIGPLAN Notices* 30, 1 (January 1995), 45–52.

Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *SPAA*.

Emery D Berger and Benjamin G Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, Vol. 41. ACM, 158–168.

S. M. Blackburn and K. S. McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*.

Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970).

Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an uncooperative environment. *Software – Practice and Experience* 18, 9 (1988), 807–820.

Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. 2003. Ownership types for safe region-based memory management in real-time Java. In *PLDI*.

John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software – Practice and Experience* 31, 6 (2001), 533–553.

Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *PODC*.

Perry Cheng, Robert Harper, and Peter Lee. 1998. Generational Stack Collection and Profile-driven Pretenuring. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 162–173. https://doi.org/10.1145/277650.277718

Dave Clarke and Tobias Wrigstad. 2003. External uniqueness is unique enough. In *ECOOP*. 176–200.

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *OOPSLA*.

Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2754169.2754181

Nachshon Cohen and Erez Petrank. 2015a. Automatic memory reclamation for lock-free data structures. In *OOPSLA*.

Nachshon Cohen and Erez Petrank. 2015b. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *SPAA*.

CoreCLR. 2017. CoreCLR: the .NET Core runtime. http://www.github.com/dotnet/CoreCLR. (2017).

Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. 2016. Idle Time Garbage Collection Scheduling. In *PLDI*.

Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *DSN*.

Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. 2003. Memory safety without runtime checks or garbage collection. *ACM SIGPLAN Notices* 38, 7 (2003), 69–80.

Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *ISMM*.

docs.rs. 2017. Rust CrossBeam library. https://docs.rs/crossbeam/0.3.0/crossbeam/. (2017).

elinux.org. 2015. Electric fence malloc debugger. http://elinux.org/Electric_Fence.

Keir Fraser. 2004. *Practical lock-freedom*. PhD Thesis UCAM-CL-TR-579. Computer Laboratory, University of Cambridge.

Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: a garbage collector for big data on big NUMA machines. In *ASPLOS*.

github.com/densh. 2017. Scala-Offheap: Type-safe off-heap memory for Scala. https://github.com/densh/scala-offheap. (2017).

Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *HotOS*.

Dan Grossman, Greg Morrisett, and Trevor Jim. 2002. Region-based Memory Management in Cyclone. In *PLDI*.

Timothy L. Harris. 2000. Dynamic Adaptive Pre-tenuring. In *Proceedings of the 2Nd International Symposium on Memory Management (ISMM '00)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/362422.362476

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*.

Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing* 67 (May 2007), 1270–1285.

Matthew Hertz and Emery D. Berger. 2005. Quantifyng the Performance of Garbage Collection vs. Explicit Memory Management. In *OOPSLA*.

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience With Safe Manual Memory-Management in Cyclone. In *ISMM*.

John Hogg. 1991. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*.

Robert Hundt. 2011. Loop Recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011*.

jemalloc.net. 2017. JEmalloc. http://jemalloc.net. (2017).

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

Piyus Kedia, Manuel Costa, Matthew Parkinson, Kapil Vaswani, and Dimitrios Vytiniotis. 2017. Simple, fast and safe manual memory management. In *PLDI*.

Byoungyoung Lee, Chengyu Song, Yeongjin Jang, and Tielei Wang. 2015. Preventing Use-after-free with Dangling Pointer Nullification. In *NDSS*.

Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2008. Archipelago: trading address space for reliability and security. In *ASPLOS*.

Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *ASPLOS*.

Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504.

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. (2014). Microsoft Research Cambridge. http://research.microsoft.com/infernet.

Naftaly Minsky. 1996. Towards alias-free pointers. In *ECOOP*. 189–209.

Adam Morrison and Yehuda Afek. 2015. Temporally Bounding TSO for Fence-Free Asymmetric Synchronization. In *ASPLOS*.

MSDN. 2016. Asynchronous Programming with async and await. https://msdn.microsoft.com/en-us/library/mt674882.aspx. (2016).

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A Type System for Borrowing Permissions. In *POPL*.

Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS Compiler-Enforced Temporal Safety for C. In *ISMM*.

Khan Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High Performance Big-Data-Friendly Garbage Collector. In *OSDI*.

Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *ASPLOS*.

Gene Novark and Emery D Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 573–584.

Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. 2017. *Project Snowflake: Safe Manual Memory Management in .NET*. Technical Report MSR-TR-2017-32. Microsoft Research. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/07/snowflake-extended.pdf

rustlang.org. 2017. Rust programming language. https://www.rust-lang.org.

Fred Smith, David Walker, and Greg Morrisett. 2000. Alias types. In *European Symposium on Programming (ESOP)*.

Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Safe and Efficient Hybrid Memory Management for Java. In *ISMM*.

D. Stefanovic, K. S. McKinley, and J. E. B. Moss. 1999. Age-based garbage collection. In *OOPSLA*.

support.microsoft.com. 2012. How to use Pageheap utility to detect memory errors. https://support.microsoft.com/en-us/kb/264471.

Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe Manual Memory-Management in Cyclone. *Science of Computer Programming* 62, 2 (October 2006), 122–14.

Gil Tene, Balaji Iyengar, and Michael Wolk. 2011. C4: The Continuously Conucrrent Compacting Collector. In *ISMM*.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2 (February 1997), 109–176.

TPC.org. 2017. The TPC Benchmark^{TM}H. http://www.tpc.org/tpch. (2017).

Philip Wadler. 1990. Linear types can change the world!. In *IFIP TC 2 Working Conference*.

David Walker, Karl Crary, and Greg Morrisett. 2000. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems* 24, 4 (2000), 701–771.

David Walker and Kevin Watkins. 2001. On regions and linear types. In *ICFP*.

Yves Younan. 2015. FreeSentry: protecting Against User-After-Free Vulnerabilities Due to Dangling Pointers. In *NDSS*.

B. G. Zorn. 1993. The measured cost of conservative garbage collection. *Software – Practice and Experience* 23, 7 (1993), 733–756.