Document Number: P0233R0
Date: 2016-02-12
Reply-to: maged.michael@acm.org, fraggamuffin@gmail.com
Authors: Maged M. Michael, Michael Wong
Project: Programming Language C++, SG14/SG1 Concurrency, LEWG

Hazard Pointers

Safe Resource Reclamation for Optimistic Concurrency

1. Introduction

Under optimistic concurrency, threads[1] may use shared resources concurrently with other threads that may make such resources unavailable for further use. Care must be taken to reclaim such resources only after it is guaranteed that no threads will subsequently use them.

More specifically, concurrent dynamic data structures that employ optimistic concurrency allow threads to access dynamic objects concurrently with threads that may remove such objects. Without proper precautions, it is generally unsafe to reclaim the removed objects, as they may yet be accessed by threads that hold references to them. Solutions for the safe reclamation problem can also be used to prevent the ABA problem, a common problem under optimistic concurrency.

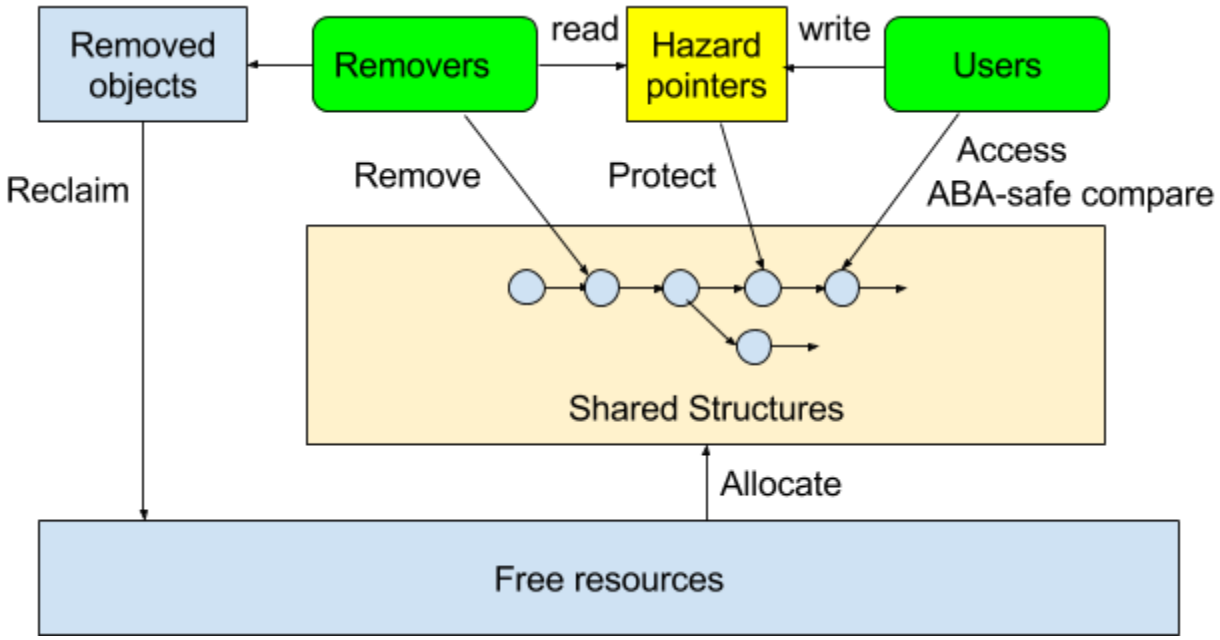
There are several methods for safe deferred reclamation. The main methods are reference counting, RCU (read-copy-update), and hazard pointers. Each method has its pros and cons and none of the methods provides the best features in all cases. Therefore, it is desirable to offer users the opportunity to choose the most suitable methods for their use cases. See paper P0232R0 (Concurrency ToolKit for Structured Deferral/Optimistic Speculation)[3] for a detailed comparative analysis of these methods along with atomic shared pointers which is based on an earlier paper by Paul McKenney [1]. This proposal focuses on the hazard pointer method [2].

We propose adding hazard pointers as a library as part of a collection of a Concurrency ToolKit methods (P0232R0).

2. Hazard Pointers

A hazard pointer is a single-writer multi-reader pointer that can be owned by at most one thread at any time. Only the owner of the hazard pointer can set its value, while any number of threads may read its value. A thread that is about to access dynamic objects optimistically acquires ownership of a set of hazard pointer (typically one or two for linked data structures) to protect such objects from being reclaimed. The owner thread sets the value of a hazard pointer to point to an object in order to indicate to concurrent threads--that may remove such object--that the object is not yet safe to reclaim.

[1] Throughout this document, we use to term *thread* to refer to any thread of execution, including language-level threads, processes, and signal handlers.



Hazard pointers are owned and written by threads that act as users (i.e., may use removable objects) and are read by thread that act as removers (i.e., may remove remove objects). The set of user and remover threads may overlap, so the same thread may write to its own hazard pointers when using objects and read the hazard pointers including those of other threads when reclaiming removed objects.

The key rule of the hazard pointers method is that **a removed object can be reclaimed only after it is determined that no hazard pointers have been pointing continuously to it from a time before its removal**.

In addition to the primary use cases for hazard pointers for memory reclamation, objects protected by hazard pointers could represent other reclaimable resources such as files, ports, and devices. Also, the method can be used by signal handlers and among processes as well as among language-level threads.

2.1. Hazard Pointer Domains

The hazard pointers method allows the presence of multiple hazard pointer domains, where the safe reclamation of resources in one domain does not require checking all the hazard pointers in different domains. It is possible for the same thread to participate in multiple domains concurrently. A domain can be specific to one or more resources, or can encompass all sharing among multiple processes in a system.

2.2. Main Structures and Operations

The main structures of the hazard pointers method are:

- **Hazard pointers:** pointer-sized variables.
- **Removed objects** awaiting reclamation.
- **Container structures** for hazard pointer records and removed objects.

The key operations are:

- **Allocate** a hazard pointer.
- **Acquire** ownership of a hazard pointer.
- **Set** the value of a hazard pointer.
- **Clear** the value of a hazard pointer.
- **Release** ownership of a hazard pointer.
- Request the **deferred reclamation** of a removed object.
- **Read** the value of a hazard pointer.

Design details are discussed in following sections.

2.3. Pros and Cons

The main advantages of the hazard pointers method are that:

1. The number of removed objects that are not yet reclaimed is bounded.
2. Readers do not interfere with each other or with writers
3. Cache friendly access patterns.
4. Constant time complexity for traversal and (expected amortized time for) reclamation
5. Its operations are lock-free (mostly wait-free), and therefore it is suitable for use in non-blocking operations that are required to be async signal-safe or immune to asynchronous process termination.

The main disadvantage of the hazard pointers method is that each traversal incurs a store-load memory order fence, when using the basic form of the method (without blocking or using interrupts).

3. Design Considerations

3.1. Progress Guarantees

Some use cases of hazard pointers require that all operations be non-blocking from end to end. An operation is non-blocking if it is guaranteed to complete in a finite number of its own steps, if it runs without interference from other operations, regardless of where other threads are blocked.

Lock-free progress is a stronger form of non-blocking progress, it further guarantees collective forward progress even in the presence of interference among threads. The hazard pointers method can have end-to-end lock-free implementations.

Non-blocking progress is an essential requirement for operations to be async signal safe. It is also essential for guaranteeing availability of resources in cases where processes may be killed asynchronously while sharing such resources.

The main trade-offs of guaranteeing lock-free progress are:

- Not using thread local storage (unless TLS is guaranteed to be non-blocking). This implies the need to implement non-blocking container structures for removed objects.
- Not using the default memory allocator, as it is unlikely to be completely non-blocking. This implies the need to design the library interface in a way that allows the specification of custom allocation and deallocation functions, as well as avoidance of memory allocation when possible.

3.2. Thread Types

Some use cases are by thread types other than typical language-level threads, in particular signal handlers and processes. Support for signal handlers requires implementation options that avoid thread local storage and that allow the use of non-blocking allocators. Support for processes require allowing custom allocation and deallocation functions that can operate on shared memory (and other shared system resources protected by hazard pointers).

3.3. Memory Allocation and Deallocation

There are several cases (as mentioned above) that require the use of custom allocators:

- The deferred reclamation of objects that are not allocated using malloc (e.g., new).
- End-to-end non-blocking progress is required.
- Sharing resources among processes.

Accordingly, the implementation must provide the capability to specify custom allocation and deallocation functions in various parts of the library interface.

3.4. Reclamation Frequency

There is a trade-off between:

- The upper bound on the number of removed objects that are not yet reclaimed.
- The time complexity of reclamation per object
- Using thread local storage.

For the purposes of this discussion, let N be the maximum number of hazard pointers (in a domain), and let M be the number of remover threads.

Using thread local storage (assuming wait-free TLS), the M removers can perform bulk reclamation after accumulating a number of removed objects that is at least $N+\Theta(N)$ (e.g., $2*N$). In such case the upper bound on the number of unreclaimed removed objects is O(M*N) and the amortized expected time per reclaimed object is constant. The progress is wait-free and contention-free.

Without using thread local storage, removed objects are inserted in shared lock-free structures. The worst-case unreclaimed removed objects can be bounded by O(N), but contention becomes possible and progress becomes lock-free instead of wait-free.

3.5. Number of Hazard Pointers and Thread caching

Using a fixed number of hazard pointers simplifies the implementation, but it restricts use and can be inconsistent with non-blocking progress if a larger number of hazard pointers is needed. For the sake of flexibility, the implementation must allow the dynamic allocation of hazard pointers.

Caching released hazard pointers between operations can minimize contention related to acquiring hazard pointers. Caching can be done transparently in the library implementation using TLS, however TLS is not always guaranteed to be non-blocking. Of course the programmer can cache hazard pointers explicitly at the cost of some inconvenience and taking responsibility for explicitly releasing hazard pointers instead of depending on their automatic release by the library.

3.6. Thread Local Storage

As discussed above the use of thread local storage has pros and cons. It reduces or eliminates contention in acquiring hazard pointers and allows wait-free progress (if TLS is wait-free). On the other hand, it is incompatible with async signal safety, and TLS implementations are not guaranteed to be non-blocking.

Due to the performance advantages of using TLS, the library implementation should allow the programmer to choose implementation paths that benefit from TLS when suitable, and avoid TLS when incompatible with the use case.

3.7. Exceptions

The sources of exceptions in implementations of the hazard pointers method are related to memory allocation, in particular the allocation of hazard pointers. All other operations can avoid memory allocation exceptions at some performance cost in the worst case when allocation is impossible.

Programmers concerned about such exceptions (for example, in real-time code) can guarantee that hazard pointer operation will not throw if they meet certain conditions. Implementations of the

method can guarantee that the total number of hazard pointers never shrinks throughout the lifetime of the associated domain. Therefore, programmers can pre-allocate the needed number of hazard pointers and then release them, knowing that all these hazard pointer will remain available for reallocation throughout the lifetime of the associated domain, provided that care is taken in managing thread caching of hazard pointers. Alternatively, programmers can avoid creating hazard pointers ahead of time by creating a simple wait-free allocator that manages sufficient memory to allocate a large number of hazard pointers (and therefore is guaranteed not to throw) and provide this allocator as an argument to the hazard pointer constructor.

3.8. Primitives and Dependencies

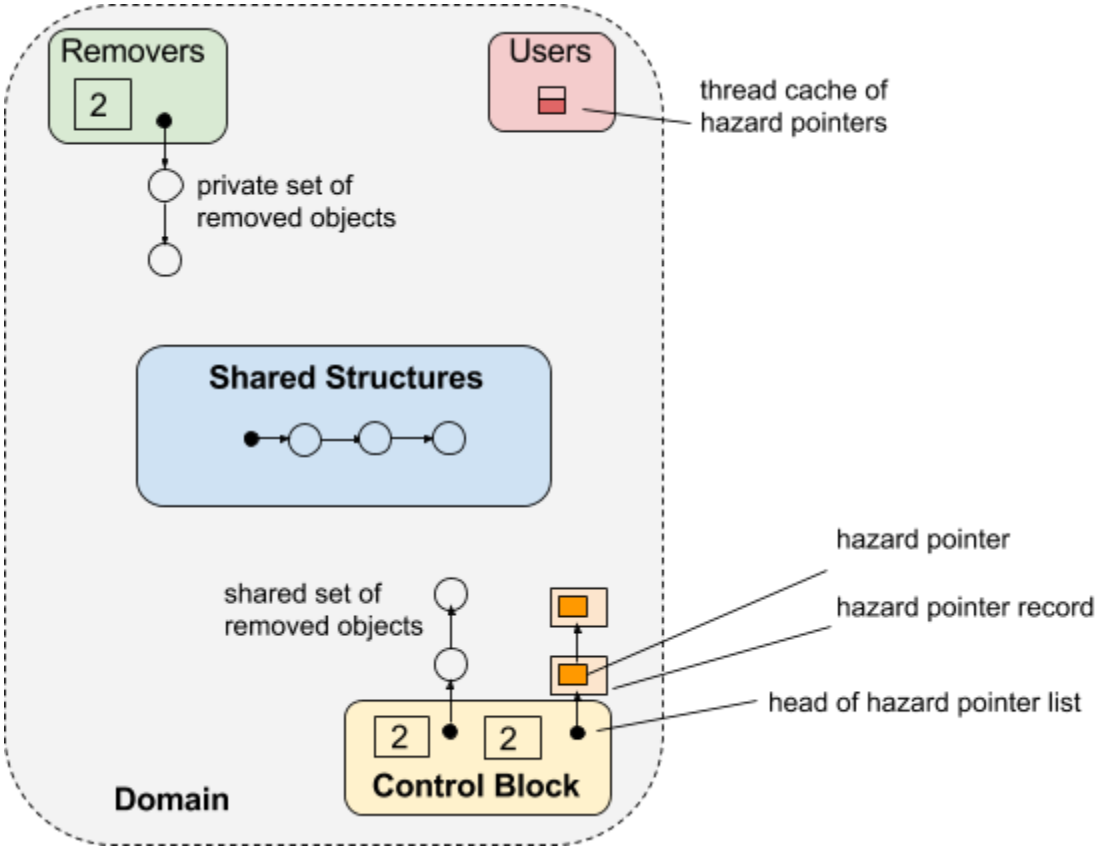
The hazard pointers method requires the use of atomic primitives on pointers and `size_t` variables and memory ordering primitives. The method has no direct dependencies on any system calls.

The method in its purely non-blocking form incurs a store-load fence. This gence becomes unnecessary if the method is used in more restricted cases such as inside lock critical sections, or by using interrupts to enforce ordering only when a remover thread is about to inspect the hazard pointers.

4. Design Overview

Based on the above considerations and with a goal of maximizing usability, we believe that hazard pointer implementations should have the following features or policies:

- Use TLS for performance but provide a path that is TLS-free.
- Provide an end-to-end lock-free path.
- Allow custom allocation and deallocation function objects.
- Support an end-to-end async signal safe path.
- Support multi-process sharing.
- Support multiple hazard pointer domains.
- Support dynamic hazard pointer allocation.
- Do not throw exceptions except in hazard pointer allocation, and provide use conditions that guarantee that hazard pointer constructors will not throw exceptions.
- Support hazard pointer caching.
- Support automatic release of hazard pointers.
- Support an interface that can avoid the store-load fence when not needed.



The above diagram shows the main components of the hazard pointer method's design:

- **Hazard pointer domains**: Multiple domains may be present concurrently. Threads may participate in multiple domains in different roles as users, removers, or both. There is one default domain per process.
- The **hazard pointer control block** is the defining component of a domain. It manages the **set of all hazard pointer records** in the domain, and the **shared set of removed blocks** (if any).
- A **hazard pointer record** contains a **hazard pointer** and an indicator of whether the hazard pointer is free or owned by a user thread. Hazard pointers may point to **removed objects** or **reachable objects in shared structures**.
- **User threads** (optionally) manage a small **thread cache** for hazard pointer records.
- **Remover threads** (optionally) manage a **private set of removed objects**.

5. Impact on the Standard

Hazard pointers will be a pure Library addition (with no Language elements) likely to Clause 30 Thread support Library [thread], or a new Clause on Concurrency support Library [concurrent]. It does require Clause 29 Atomic operations library [atomics] for atomic operations and memory ordering.

6. Existing Implementations and Target Workloads

The hazard pointer method is used in several proprietary products that require high-availability and non-blocking progress for safe resource management. Other uses are in supporting lock-free access in key-value stores and applications with soft real-time requirements. The method is used in the MongoDB/WiredTiger open-source NoSQL database.

There are several open source implementations, such as Concurrency Kit, Concurrency Building Blocks, libcds, and Parallelism Shift. These implementations provide different interfaces that have their pros and cons. In this proposal we aim to maximize flexibility, and use variations of the flexible features of these interfaces and avoid restrictive features, such as supporting regular threads only, or requiring the numbers of hazard pointers to be fixed beforehand.

7. Comparison of Deferred Reclamation Methods

	Reference Counting	**RCU**	**Hazard Pointers**
Unreclaimed objects	**Bounded**	Unbounded	**Bounded**
Non-blocking traversal	**Lock-free**	**Wait-free**	**Lock-free.**
Non-blocking reclamation	**Lock-free**	Blocking	**Lock-free**
Contention among readers	Can be very high	**No contention**	**No contention**
Traversal speed	Atomic updates	**No or low overhead**	Store-load fence
Reference acquisition	Conditional	**Unconditional**	Conditional
Automatic reclamation	**Yes**	No	No

Advantages

8. Proposal for Adding a `haz_ptr` Library

The proposed library, `haz_ptr`, includes three public classes (See Appendix A):

1. `haz_ptr_control_block`
2. `haz_ptr_obj`
3. `haz_ptr`

8.1. `haz_ptr_control_block` class

This class is the root of all shared hazard pointer data structures in a domain. There is exactly one instance of this class in each domain. It is included in the library header in order to allow the programmer to create and control hazard pointer domains. The only public functions are the constructor and the destructor:

- **Constructor:** Default constructor.
- **Destructor:** Destroys all shared hazard pointer structures.
- Usage example (using a hypothetical shared memory allocator):
    - ```
      auto ptr = shared_mem_alloc(sizeof(haz_ptr_control_block));
      new (ptr) haz_ptr_control_block;
      auto cb = static_cast<haz_ptr_control_block*>(ptr);
      /* Pass cb as argument of hazard pointer operations in this
         domain. */
      ```

This class does not allow copy and move constructors and assignment operators:

```
haz_ptr_control_block(haz_ptr_control_block&) =delete;
haz_ptr_control_block(haz_ptr_control_block&&) =delete;
haz_ptr_control_block& operator=(haz_ptr_control_block&) =delete;
haz_ptr_control_block& operator=(haz_ptr_control_block&&) =delete;
```

8.2. `haz_ptr_obj` class

This is the base class for objects protected by hazard pointers. It has no public functions.
Usage example:

```
class Foo {
  class Node : public haz_ptr_obj {
    ...
  }
  /* Use hazard pointers to protect Node objects. */
  ...
```

This class contains two pointer-sized variables that are only used after removal, a link for inclusion in linked containers that do not require further memory allocation and a pointer to a function object for a reclamation function of the derived object.
```

8.3. `haz_ptr` class

This is the main hazard pointer class. It serves two purposes:

- Objects of this class manage all operations on individual hazard pointers (allocation, acquisition, setting, clearing, and explicit or implicit release).
- This class provides the interface for programmers to invoke all hazard pointer operations that are not specific to an individual hazard pointer.

The class defines two enum types:

- **`tc_policy`** for the thread caching policy for hazard pointer records.
  ```
  enum tc_policy {cache, nocache};  // To cache or not to cache
  ```
- **`rem_policy`** for private vs. shared handling of removed objects.
  ```
  enum rem_policy {priv, shared};   // Private vs. shared
  ```

The class supports the following functions::

- **Constructor:**
  ```
  haz_ptr(
    tc_policy tc = tc_policy::cache,
    std::function<void*(size_t)>* alloc = nullptr,  // Default malloc
    std::function<void(void*)>* dealloc = nullptr,  // Default free
    haz_ptr_control_block* control_block = nullptr);
  ```
  - Parameter **`tc`**: Controls the policy for thread caching for this hazard pointer. If set to avoid thread caching (e.g., used in a signal handler) a new hazard pointer record is acquired and possibly allocated through the control block. If set to use caching, then the thread cache is checked first for an available record. Thread caching is allowed by default.
  - Parameter **`alloc`**: A pointer to a function object for the function to use to allocate a new hazard pointer record if needed. The default value nullptr is associated with the default allocation function `malloc`.
  - Parameter **`dealloc`**: A pointer to a function object for the function to use to deallocate the acquired hazard pointer record if it needed to be allocated. The default value `nullptr` is associated with the default deallocation function `free`.
  - Parameter **`control_block`**: A pointer to the control block to use to manage this hazard pointer. The default value, nullptr, indicates using the default control block for the process. There is one default control block (and hence one default domain) per process. If thread caching is allowed for this constructor and the control block is different from the control block associated with cached hazard pointers, then the cached hazard pointers are released to their proper control block.
  - The constructor acquires a hazard pointer record (which contains a hazard pointer). The acquired record may be newly allocated or pre-owned.
  - May throw `bad_alloc`, unless the provided allocation function guarantees not to throw. This is the only function that may throw in the hazard pointer library.

- Usage examples:
  - `haz_ptr hptr1;`
  - ```
    haz_ptr hptr2(
      haz_ptr::tc_policy::nocache,
      loch_free_alloc,
      lock_free_dealloc,
      cb);
    ```
- The calling thread becomes the only owner of the acquired hazard pointer until the former releases the latter, either explicitly (by calling release()) or implicitly at the destruction of the haz_ptr object.
- We may add a nothrow version that requires pointers to function objects for nothrow allocation and deallocation functions.

- **Destructor:**
  ~haz_ptr();
  - The destructor clears and releases the owned hazard pointer.
  - The clearing and release of the hazard pointer is guaranteed to be ordered after prior loads and stores.
  - The released hazard pointer may be cached by the thread if thread caching was allowed in the constructor and there is an empty slot in the thread cache. Otherwise, the hazard pointer is released to the associated control block.
- **template <typename T>**
  bool protect(const T* ptr, const std::atomic<T*>* src);
  - Parameter **ptr**: A pointer to a block of type T.
  - Parameter **src**: A pointer to an atomic pointer to an object of type T.
  - This function sets the value of the owned hazard pointer to `ptr`, then checks if `*src` has the value `ptr`.
  - Return value: The function returns true if *src is found to have the value ptr *after* the setting of the hazard pointer. Otherwise, it returns false.
  - If this function returns true, then ptr is safe to dereference and comparisons with ptr are ABA-safe until the hazard pointer is released, cleared or it is used to protect a different pointer, provided that removers use only `haz_ptr::reclaim()` to reclaim the memory of `*ptr`.
  - Usage example:
    ```
    if (hptr.protect<Node>(pnode, &head))
      // Now, it is safe to dereference pnode
      // and comparisons with pnode are ABA-safe
    ```
- **void set(void* ptr);**
  - Parameter **ptr**: A pointer value representing a resource to be protected by the hazard pointer.
  - This function sets the value of the owned hazard pointer to `ptr`.
  - This function is similar to protect(), but without validating that ptr points to a reachable object (i.e., not removed).
  - This function does not provide any memory ordering guarantees.
    ```

- Usage example:

```
hptr.set(pnode);
```

- **`void clear();`**
  - This function clears the value of the owned hazard pointer.
  - The clearing of the hazard pointer is guaranteed to be ordered after prior loads and stores.
  - Usage example:

```
hptr.clear();
```

- **`static void reclaim(`**

```
haz_ptr_obj* ptr,
std::function<void(void*)>* dealloc = nullptr, // Default free
haz_ptr_control_block* control_block = nullptr,
rem_policy rem = rem_policy::priv,
size_t mult = 2);
```

  - Parameter **`ptr`**: A pointer to a removed object to be reclaimed.
  - Parameter **`dealloc`**: A pointer to a function object for the function to be used to reclaim the removed object. The default value `nullptr` is associated with the default deallocation function `free`.
  - Parameter **`control_block`**: A pointer to the hazard pointer control block associated with the removed object. The default value, `nullptr`, represents the default control block for the current process.
  - Parameter **`rem`**: The policy for accumulating the removed object in a private (i.e., thread local) or shared (in the control block) structures.
  - Parameter **`mult`**: The reclamation multiplier. The hazard pointers are checked after accumulating a number of removed objects that is at least `mult` times the number of hazard pointers in the control block.. This parameter can help a thread increase its chances of performing a higher or lower fraction of the reclamation work compared to other threads. For example, a high-priority thread can set this parameter to a higher value to increase its chances of doing less reclamation work than other threads.
  - This function sets in motion the reclamation (possibly deferred until safe) of the removed object.
  - The caller thread of this function need not own any hazard pointers.
  - Usage example:
    - `haz_ptr::reclaim(pnode);`
    - `haz_ptr::reclaim(pblock, deleteBlockFn, cb, haz_ptr::rem_policy::shared, 3);`

- **`static void swap(haz_ptr& a, haz_ptr& b);`**
  - Parameters **`a`** and **`b`**: References to haz_ptr objects.
  - This function swaps the hazard pointer records ownerships between the two objects. The function does not change the values of the owned hazard pointers themselves. Throughout the swap, each of the owned hazard pointers continues to protect the object that it is protecting (if any). See the linked list set example.

- **`static size_t count(haz_ptr_control_block* = nullptr);`**
  - Parameter **`control_block`**: A pointer to a hazard pointer control block.The default value, `nullptr`, represents the default control block for the current process.
  - Return value: The total number of allocated hazard pointer including free ones associated with the control block.
  - This function can be used by programmers to optimize the usage of hazard pointers.
  - The caller of this function need not own any hazard pointers.
  - Usage example:

```
if (haz_ptr::count() < THRESHOLD) ...
```

- **`static void release_cached();`**
  - This function releases any hazard pointers cached by the current thread.
- **`static void empty_private_removed();`**
  - This function moves any privately accumulated removed objects to the control block, and possibly performing reclamation, effectively relieving the current thread from the responsibility for reclaiming the remaining objects.

This class does not allow copy and move constructors and assignment operators:

```
haz_ptr(haz_ptr&) =delete;
haz_ptr(haz_ptr&&) =delete;
haz_ptr& operator=(haz_ptr&) =delete;
haz_ptr& operator=(haz_ptr&&) =delete;
```

9. Sample Interface and Implementation

A C++ Standard Library sample interface code is in Appendix A. An implementation of the interface was tested using C++11 standard mode on PowerPC Little Endian Redhat 8 with gcc 4.8 and clang 3.9 (top of trunk), and x86_64 Ubuntu 14.04 with gcc 4.8.

10. Use Examples

Highlighted code uses the `haz_ptr` library interface.

10.1. Lock-Free LIFO List

This example demonstrates some variations of the basic operations of the hazard pointer method.

```
template <typename T> class LockFreeLIFO {
  /* Derived from base hazard pointer-protected object class */
  struct Node : public haz_ptr_obj {
    T value;
    Node* next;
```

```
    Node(T v, Node* n) : value(v), next(n) { }
  };

  std::atomic<Node*> head;

  /* Using malloc to match default haz_ptr deallocation function free. */
  Node* allocNode(T v, Node* n) {
    auto ptr = malloc(sizeof(Node));
    new (ptr) Node(v, n);
    return static_cast<Node*>(ptr);
  }

public:

  LockFreeLIFO() : head(nullptr) { }

  void push(T val) {
    auto pnode = allocNode(val, head.load());
    while (!head.compare_exchange_weak(pnode->next, pnode));
  }

  bool pop(T& val) {
    /* Acquire a hazard pointer, but don't use thread caching. */
    haz_ptr hptr(haz_ptr::tc_policy::nocache);
    Node* pnode;
    while (true) {
      if ((pnode = head.load()) == nullptr) break;
      /* Try to protect pnode using hptr. If protection cannot be
         validated, then skip the hazards. */
      if (!hptr.protect<Node>(pnode, &head)) continue;
      /* Now, pnode is protected. */
      /* Dereference of pnode is safe. */
      auto next = pnode->next;
      /* Comparison with pnode is ABA-safe. */
      if (head.compare_exchange_weak(pnode, next)) break;
    }
    /* Clear the hazard pointer, */
    hptr.clear();
    if (!pnode) return false;
    val = pnode->value;
    /* Reclaim *pnode only after no hazard pointers point to it. */
    haz_ptr::reclaim(pnode);
    return true;
```

```
    /* The hazard pointer is released automatically at the end of the
       scope */
  }

};
```

10.2.Single-Writer Multi-Reader Singly-Linked List

This example demonstrate the use of haz_ptr::swap for hand-over-hand traversal,

```
/**
 * Set implemented as an ordered singly-linked list.
 *
 * A single writer thread may add or remove elements. Multiple reader
 * threads may search the set concurrently with each other and with
 * the writer's operations.
*/
template <typename T> class SWMRListSet {
  struct Node : public haz_ptr_obj {
    T elem;
    std::atomic<Node*> next;
    Node(T e, Node* n) : elem(e), next(n) { }
  };

  std::atomic<Node*> head;

  Node* allocNode(T e, Node* n) {
    auto ptr = malloc(sizeof(Node));
    new (ptr) Node(e, n);
    return static_cast<Node*>(ptr);
  }

  /* Used by the single writer */
  void locate_lower_bound(T v, std::atomic<Node*>*& prev) {
    auto curr = prev->load(std::memory_order_relaxed);
    while (curr) {
      if (curr->elem >= v) break;
      prev = &(curr->next);
      curr = curr->next.load(std::memory_order_relaxed);
    }
    return;
  }
```

```
public:

  SWMRListSet() : head(nullptr) { }

  bool add(T v) {
    auto prev = &head;
    locate_lower_bound(v, prev);
    auto curr = prev->load(std::memory_order_relaxed);
    if (curr && curr->elem == v) return false;
    prev->store(allocNode(v, curr), std::memory_order_release);
    return true;
  }

  bool remove(T v) {
    auto prev = &head;
    locate_lower_bound(v, prev);
    auto curr = prev->load(std::memory_order_relaxed);
    if (!curr || curr->elem != v) return false;
    prev->store(curr->next.load(std::memory_order_relaxed),
                std::memory_order_release);
    /* Reclaim *curr only after no hazard pointers point to it. */
    haz_ptr::reclaim(curr);
    return true;
  }

  /* Used by readers */
  bool contains(T val) {
    /* Acquire two hazard pointers for hand-over-hand traversal. */
    haz_ptr hptr_prev;
    haz_ptr hptr_curr;
    T elem;
    bool done = false;
    while (!done) {
      auto prev = &head;
      auto curr = prev->load(std::memory_order_relaxed);
      while (true) {
        if (!curr) { done = true; break; }
        /* Alternative way to protect curr instead of protect().
         * First, Set hazard pointer value.
         * Then validate protection.
         * If validation fails skip the hazards.
         */
        hptr_curr.set(curr);
```

```
        std::atomic_thread_fence(std::memory_order_seq_cst);
        if (prev->load(std::memory_order_relaxed) != curr) break;
        /* Now, curr is protected. */
        /* Dereference of curr is safe. */
        auto next = curr->next.load(std::memory_order_relaxed);
        elem = curr->elem;
        /* Comparison with pnode is ABA-safe. */
        std::atomic_thread_fence(std::memory_order_acquire);
        if (prev->load(std::memory_order_relaxed) != curr) break;
        if (elem >= val) { done = true; break; }
        prev = &(curr->next);
        curr = next;
        /* Swap the haz_ptr objects. The swap does not change the
         * values of the owned hazard pointers themselves. After the
         * swap, The hazard pointer owned by hptr_prev continues to
         * protect the node that contains the pointer *prev. The
         * hazard pointer owned by hptr_curr will continue to protect
         * the node that contains the old *prev (unless the old prev
         * was &head), which no longer needs protection, so
         * hptr_curr's hazard pointer is now free to protect *curr in
         * the next iteration (if curr != null).
         */
        haz_ptr::swap(hptr_prev, hptr_curr);
      }
    }
    return elem == val;
    /* The hazard pointers are released automatically. */
  }

};
```

10.3.Wide Compare and Set

This example demonstrates the use of a function object for deferred deallocation, and the use of a custom hazard pointer control block. In this case each object of this class constitutes its own domain and has its own control block.

```
/** A class that supports atomic load and compare and set on memory
 *  blocks of arbitrary width
*/
template <size_t M> class WideCAS {

  /* Clases of objects protected by hazard pointers must be derived
```

```
     from the base class haz_ptr_obj */
  class Block : public haz_ptr_obj {
    char a[M];

    friend WideCAS;
    Block(const char* b) { memcpy(a, b, M); }
    void get(char* b) const { memcpy(b, a, M); }
    bool cmp(const char* b) const { return memcmp(a, b, M) == 0; }
  };

  std::atomic<Block*> block;

  /* Special hazard pointers control block. Must be included in all
     related hazard pointers operations for consistency. */
  haz_ptr_control_block* cb = new haz_ptr_control_block;

  /** void(void*) destructor for use by hazard pointers deferred
   *  reclamation.
   */
  static void deleteBlock(void* ptr) {
    auto pblock = static_cast<Block*>(ptr);
    /* Delete the control block and all associated hazard pointers and
       structures. */
    delete pblock;
  }

  /* Function object to encapsulate the deleteBlock function */
  static std::function<void(void*)>* deleteBlockFn;

public:

  WideCAS(char* b) : block(new Block(b)) { }

  ~WideCAS() {
    delete block.load(std::memory_order_relaxed);
    delete cb;
  }

  void load(char* b) const {
    /* Acquire a hazard pointer, with special control block. */
    haz_ptr hptr(haz_ptr::tc_policy::nocache, cb);
    do {
      auto pblock = block.load(std::memory_order_relaxed);
```

```
    /* Try to protect pblock using hptr. If protection cannot be
       validated, then skip the hazards. */
    if (!hptr.protect(pblock, &block)) continue;
    /* Now, pblock is protected. */
    /* Dereference of pblock is safe. */
    pblock->get(b);
    break;
  } while (true);
  return;
  /* Ownership of the hazard pointer is released automatically at
     the end of the scope. */
}

bool compare_and_set(const char* expval, const char* newval) {
  /* Acquire a hazard pointer, with the same special control block. */
  haz_ptr hptr(haz_ptr::tc_policy::nocache, cb);
  auto newblock = new Block(newval);
  do {
    auto pblock = block.load();
    /* Try to protect pblock using hptr. If protection cannot be
       validated, then skip the hazards. */
    if (!hptr.protect(pblock, &block)) continue;
    /* Now, pblock is protected. */
    /* Dereference of pblock is safe. */
    if (!pblock->cmp(expval)) { delete newblock; return false; }
    /* Comparison with pblock is ABA-safe. */
    if (block.compare_exchange_weak(pblock, newblock)) {
      /* Clear the hazard pointer. */
      hptr.clear();
      /* Request reclamation of the removed block. Must use the
         special control block and a deallocation function object
         that matches the allocation function. Also, set the
         reclamation multiplier to 3 (i.e., check the hazard
         pointers after accumulating a number of removed objects
         that is at least 3 times the number of hazard pointers in
         the domain of the control block). */
      hpns::haz_ptr::reclaim(pblock, deleteBlockFn, cb,
                             haz_ptr::rem_policy::shared, 3);
      return true;
    }
  } while (true);
  /* Ownership of the hazard pointer is released automatically at
     the end of the scope. */
```

```
  }

};

/* Initialize the function object for custom deallocation function. */
template<size_t M>
std::function<void(void*)>* WideCAS<M>::deleteBlockFn =
  new std::function<void(void*)>(WideCAS<M>::deleteBlock);
```

11. Appendix A: Draft Library Header

```
class haz_ptr;  /* Main hazard pointer class */
class haz_ptr_rec;  /* Hazard pointer record */
class haz_ptr_control_block;  /* Control block - One per domain */

class haz_ptr_user;  /* One thread local per thread */
class haz_ptr_remover;  /* One thread local per thread */

/** haz_ptr_obj
 *
 *  Base class for objects protected by hazard pointers.
 */
class haz_ptr_obj {
  /* Pointer used in constructing lists of removed objects awaiting
     reclamation, without requiring additional allocation. */
  haz_ptr_obj* next_removed_;

  /* Pointer to a destructor function object. */
  std::function<void(void*)>* dealloc_;

  friend haz_ptr;
  friend haz_ptr_control_block;
};

/** haz_ptr_control_block
 *
 *  Control block for hazard pointers. One per domain.
 */
class haz_ptr_control_block {
  /* Head of a linked list of hazard pointer records. */
```

```
  std::atomic<haz_ptr_rec*> hptr_head_ {nullptr};

  /* Upper bound on number of hazard pointer records in list. */
  std::atomic<size_t> hptr_count_ {0};

  /* Head of linked list of removed objects. */
  std::atomic<haz_ptr_obj*> removed_ {nullptr};

  /** Estimate of number of removed objects in list. Intended to be a
      cumulative lower bound. Overestimations are always offset by
      prior underestimations. */
  std::atomic<size_t> removed_count_ {0};

  haz_ptr_control_block(haz_ptr_control_block&) =delete;
  haz_ptr_control_block(haz_ptr_control_block&&) =delete;
  haz_ptr_control_block& operator=(haz_ptr_control_block&) =delete;
  haz_ptr_control_block& operator=(haz_ptr_control_block&&) =delete;

  friend haz_ptr;

  haz_ptr_rec* hptr_head_load();
  bool hptr_head_cas(haz_ptr_rec*&, haz_ptr_rec*);
  size_t hptr_count_load();
  void hptr_count_increment();
  void push_removed(haz_ptr_obj*, haz_ptr_obj*, size_t);
  void pop_removed(haz_ptr_obj*&, size_t&, size_t);

public:
  haz_ptr_control_block() =default;
  ~haz_ptr_control_block();
};

/** haz_ptr_rec
 *
 *  Hazard pointer record.
 */

class haz_ptr_rec {
  /* Hazard pointer. The key structure. */
  std::atomic<void*> hptr_ {nullptr};
  /* Pointer to next hazard pointer record. */
  haz_ptr_rec* next_;
  /* True if owned. */
```

```
  std::atomic<bool> valid_ {true};
  /* Function to reclaim the memory of this record. */
  std::function<void(void*)>* dealloc_;

  haz_ptr_rec(haz_ptr_rec&) =delete;
  haz_ptr_rec(haz_ptr_rec&&) =delete;
  haz_ptr_rec& operator=(haz_ptr_rec&) =delete;
  haz_ptr_rec& operator=(haz_ptr_rec&&) =delete;

  friend haz_ptr;
  friend haz_ptr_user;
  friend haz_ptr_control_block;

  haz_ptr_rec(haz_ptr_rec* next, std::function<void(void*)>* dealloc);

  template <typename T>
  bool protect(T* ptr, const std::atomic<T*>* src) {
    hptr_.store(ptr, std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_seq_cst);
    return src->load(std::memory_order_relaxed) == ptr;
  }

  bool acquire();
  void set(void* ptr);
  void clear();
  void release();
};

/** haz_ptr
 *
 * Wrapper class for RAII automatic allocation and release of hazard
 * pointers, and interface for user calls to hazard pointer functions.
 */

class haz_ptr {
public:
  enum tc_policy {cache, nocache};
  enum rem_policy {priv, shared};

private:
  /* Pointer to owned hazard pointer record. */
  haz_ptr_rec* rec_;
  /* Pointer to control block. */
```

```
  haz_ptr_control_block* cb_;
  /* To cache or not to cache. */
  tc_policy cache_ {tc_policy::cache};

  haz_ptr(haz_ptr&) =delete;
  haz_ptr(haz_ptr&&) =delete;
  haz_ptr& operator=(haz_ptr&) =delete;
  haz_ptr& operator=(haz_ptr&&) =delete;

  void release();

  friend haz_ptr_rec;
  friend haz_ptr_user;
  friend haz_ptr_remover;

  static void drain(haz_ptr_control_block*);
  static void scan_private(haz_ptr_control_block*);
  static void scan_shared(haz_ptr_control_block*, size_t);

  static std::function<void*(size_t)>* mallocfn;
  static std::function<void(void*)>* freefn;

public:
  haz_ptr(tc_policy tc = tc_policy::cache,
          haz_ptr_control_block* control_block = nullptr,
          std::function<void*(size_t)>* alloc = nullptr,
          std::function<void(void*)>* dealloc = nullptr);
  ~haz_ptr();

  template <typename T>
  bool protect(T* ptr, const std::atomic<T*>* src) {
    return rec_->protect<T>(ptr, src);
  }

  void set(void* ptr);
  void clear();

  static void reclaim(haz_ptr_obj* ptr,
                      std::function<void(void*)>* dealloc = nullptr,
                      haz_ptr_control_block* control_block = nullptr,
                      rem_policy rem = rem_policy::priv,
                      size_t mult = 2);
  static void swap(haz_ptr& a, haz_ptr& b);
```

```
  static size_t count(haz_ptr_control_block* = nullptr);
  static void release_cached();
  static size_t private_removed_count();
};
```

12. Acknowledgement

Thanks to Paul McKenney for reviewing this draft.

13. References

[1] Paul E McKenney. "Structured deferral: synchronization via procrastination." *Communications of the ACM* 56.7 (2013): 40-49.
[2] Maged M Michael. "Hazard pointers: Safe memory reclamation for lock-free objects." *Parallel and Distributed Systems, IEEE Transactions on* 15.6 (2004): 491-504.
[3] P0233R0,, P. McKenney, M. Wong, M. Michael, A Concurrency ToolKit for Structured Deferral or Optimistic Speculation