Garbage Collection in the Next C++ Standard

Hans-J. Boehm, Mike Spertus

HP Laboratories
HPL-2009-360

Keyword(s):
C++, garbage collection

Abstract:

C++ has traditionally relied on manual memory management. Sometimes this has been augmented by limited reference counting, implemented in libraries, and requiring use of separate pointer types. In spite of the fact that conservative garbage collectors have been used with C for decades, and with C++ for almost as long, they have not been well-supported by language standards. This in turn has limited their use. We have led an effort to change this by supporting optional "transparent" garbage collection in the next C++ standard. This is designed to either garbage collect or detect leaks in code using normal unadorned C++ pointers. We initially describe an ambitious effort that would have allowed programmers to explicitly request garbage collection. It faced a number of challenges, primarily in correct interaction with existing libraries relying on explicit destructor invocation. This effort was eventually postponed to the next round of standardization. This initial effort was then temporarily replaced by minimal support in the language that officially allows garbage collected implementations. Such minimal support is included in the current committee draft for the next C++ standard. It imposes an additional language restriction that makes it safe to garbage collect C++ programs. Stating this restriction proved subtle. We also provide narrow interfaces that make it easy to both correct code violating this new restriction, and to supply hints to a conservative garbage collector to improve its performance. These provide interesting implementation challenges. We discuss partial solutions.

External Posting Date: November 6, 2009 [Fulltext] Approved for External Publication
Internal Posting Date: November 6, 2009 [Fulltext]

Presented at ISMM 2009, Dublin, Ireland, June 2009

Garbage Collection in the Next C++ Standard

Hans-J. Boehm
HP Laboratories
Hans.Boehm@hp.com

Mike Spertus
Symantec
mike_spertus@symantec.com

Abstract

C++ has traditionally relied on manual memory management. Sometimes this has been augmented by limited reference counting, implemented in libraries, and requiring use of separate pointer types. In spite of the fact that conservative garbage collectors have been used with C for decades, and with C++ for almost as long, they have not been well-supported by language standards. This in turn has limited their use.

We have led an effort to change this by supporting optional "transparent" garbage collection in the next C++ standard. This is designed to either garbage collect or detect leaks in code using normal unadorned C++ pointers.

We initially describe an ambitious effort that would have allowed programmers to explicitly request garbage collection. It faced a number of challenges, primarily in correct interaction with existing libraries relying on explicit destructor invocation. This effort was eventually postponed to the next round of standardization.

This initial effort was then temporarily replaced by minimal support in the language that officially allows garbage collected implementations. Such minimal support is included in the current committee draft for the next C++ standard. It imposes an additional language restriction that makes it safe to garbage collect C++ programs. Stating this restriction proved subtle.

We also provide narrow interfaces that make it easy to both correct code violating this new restriction, and to supply hints to a conservative garbage collector to improve its performance. These provide interesting implementation challenges. We discuss partial solutions.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management

General Terms languages, standardization, algorithms

Keywords garbage collection, C++

1. Introduction

Most recently designed programming languages such as Java and C# provide for automatic reclamation of the memory associated with unreachable objects, commonly referred to as *garbage collection*. However, languages such as C and C++ , which commonly still dominate for the development of systems software, traditional desktop applications, and embedded systems, do not intrinsically provide such a facility. Instead the programmer is expected to explicitly deallocate objects when they are no longer needed.

Such explicit memory management is widely recognized as error-prone. Especially in applications that manipulate complicated linked data structures, it may be difficult to identify the last use of an object. Mistakes lead to either duplicate deallocations and possible security holes, or memory leaks.

This problem becomes more severe in multithreaded applications. Consider a program that maintains an atomically updatable pointer P to a rarely modified object O. Concurrent threads frequently access O by simply dereferencing P, using suitable atomic pointer operations. This typically adds little overhead to accesses. And O can be updated by simply allocating a new copy, initializing it with the updated values, and atomically updating the pointer P to point to the new version. In a garbage collected setting, this just works.

However, there is no easy way to explicitly deallocate the old object O in this scheme. The thread performing the update cannot immediately deallocate the old object, since another thread may still be reading it. And indeed, adapting this technique to explicit memory management in an OS kernel setting constitutes the core of a recent Ph.D. dissertation [26].

Unlike C, C++ already simplifies memory management in a number of ways, none of which is a complete solution:

- Libraries are generally designed to export values that can be passed around and manipulated as simple values, without the explicit use of pointers or memory allocation. Any memory allocated by the implementation is implicitly deallocated as these value objects are destroyed. This may require techniques such as reference counting of shared objects, but they are hidden from the library users. Although useful, this does not help the library implementor, nor the user who has to invent new pointer-containing data structures.
- C++ implementations commonly support a reference-counted "smart pointer" library called `shared_ptr`[18]. This library facility is expected to become part of the next C++ standard, which is conventionally referred to as C++0x.[1] This effectively allows data structures that use `shared_ptr` pointer types to be garbage collected. The programmer is required to avoid `shared_ptr` cycles, possibly with the aid of a provided `weak_ptr` type.

  For small objects and multithreaded executables, this approach is often much more expensive than tracing garbage collectors [10], though this is expected to improve somewhat as a result of other C++0x language facilities. It is barely applicable at all in cases like our example above, where atomic pointer assignments are required.

[1] We are not deterred by the fact that it is now essentially impossible for the standard to be formally approved before 2010.

Perhaps more significantly, this approach is not really applicable if the existing code base involves libraries that already traffic directly in C++ pointers, as most existing code does.

- Although not part of the C++ standard per se, Microsoft's C++ /CLI supports both regular C++ pointers as well as a different class of pointers to a "managed", i.e. garbage-collected heap. Although this allows a wider variety of garbage collection techniques of the managed heap, existing code, even templatized generic libraries, can no longer be used for objects residing in this heap.

Along with Demers, and Weiser, we originally developed techniques more than 20 years ago [15][2] that allow C programs to also rely on garbage collection. The same techniques can be used to identify memory leaks [15] in programs using explicit deallocation, an approach used by Purify[21] and other systems. The basic solution is to "conservatively" treat everything as a potential pointer, with a fast filtering mechanism that recognizes bit patterns that actually could be pointers to valid objects.

This approach was subsequently explored and refined over several decades (cf. [11, 7, 6, 29, 4, 19, 5, 27, 23, 20, 8]), and has worked sufficiently well to have enjoyed some significant successes. It has been used directly in some C or C++ applications, and has often been used in runtimes for garbage collected languages (e.g. in gcj, the GNU static Java compiler and in the Mono CLR implementation), both because it allows a faster initial implementation, and because it usually supports better interoperability with C.

Most Linux distributions include at least one version of our garbage collection library[3], along with a few C or C++ applications that rely on them.

The second author started a company, eventually indirectly acquired by Symantec, that used the approach to fix memory leaks in existing large applications.

In spite of its successes, most C and C++ applications clearly do not rely on such conservative garbage collection. In many cases, this is entirely appropriate, since libraries such as our garbage collector[3] often cannot satisfy some application constraint, e.g. guaranteed required bounds on response times or space usage [8].

But for many other applications, the main objection to adoption of garbage collection has been its poor support by language standards and vendor implementations.

There are rare C and C++ programs, arguably supported by existing standards, that will fail in the presence of a garbage collector like ours. These effectively "disguise" pointers sufficiently that the garbage collector is unable to recognize them, and hence mistakenly identifies the referenced objects as unreachable, recycling the memory while it is still in use. The most common instance of this are data structures that combine, usually by exclusive-oring them, two pointers in a single pointer-sized field.

Similarly, few existing vendor C or C++ implementations directly support garbage collection. (Notable exceptions are the Sun Studio and Digital Mars compilers[25, 16]).

Thus our goal was to provide for proper garbage collection support *in the language standard*, in this case the C++0x standard. Interestingly, in the process of trying to make this approach more palatable to the standards committee, and in developing a prototype implementation based on our garbage collection library[3], we ended up addressing a number of technical issues that we believe had previously been under-appreciated.

[2] Doug McIlroy independently explored similar approaches somewhat earlier. Less extreme approaches were also pursued in [28] and [1]. Much later, the initial Java implementations used a related, but much lower performance, approach.

2. Our Initial Proposal

We initially proposed comprehensive garbage collection support for C++0x, as a required feature of the implementation, but whose use by the programmer is optional [13]. Although this approach was repeatedly supported by a majority of the committee and was approved as part of the Registration Draft for C++0x, a subsequent vote removed it from the draft standard, largely based on concerns of integrating such a complex language feature on the tight schedule remaining in the C++0x time-frame. Open issues relating to the interaction between distinct libraries and ensuring invocation of C++ destructors that explicitly deallocate non-memory resources in a garbage collected environment figured prominently among the concerns.[3] Resolving these concerns would take more time, leading to a consensus that the comprehensive garbage collection proposal would be best addressed in an upcoming technical report containing extensions.

The proposal was designed to support transparent operation while simultaneously allowing programmer control.

2.1 Transparent Garbage Collection

Transparency enables easy use of garbage collection of existing source and object files. In typical cases, no changes to source code are required. Transparent garbage collection enables a number of important use cases.

Pure garbage collection We feel that if C++ is not an implementation option for the many programs that have no need for an explicit memory management, C++ will be excluded from many mainstream projects for which it is otherwise suitable [2]. We feel transparent support for the simple case of pure garbage collection is critical as garbage collection support that relies on distinct pointer types, allocators, and the like tends to result in opportunities for programmer error, incompatibility with existing libraries, and overall second-class support for pure garbage collected styles in C++ .

Standards support for leak detectors By providing a standardized definition of what constitutes a memory leak, leak detectors can produce more accurate and reliable reports on existing manually managed programs. For this reason, garbage collection is of considerable interest even to those programmers who do not intend to deploy garbage collected programs.[4]

Litter collection Since many existing programs leak memory, a common use for C++ garbage collection is to protect existing programs against memory leaks. Such "litter collection" cycles do not significantly affect application throughput since they only need to be run infrequently (often on the order of one per hour).

2.2 Programmer Control

While adding transparent support for pure garbage collection, we did not wish to impose garbage collection where it is not desired and so included explicit control mechanisms for disabling garbage collection or mixing it with explicit memory management. We did not include any support for scheduling the garbage collector explicitly, as this has proven problematic in languages such as Java, where explicit collection commands are typically treated as null operations.

[3] Statements here about C++ committee actions clearly reflect the understanding of the authors, who were proponents of this change, and active participants in discussions. Biases and misunderstandings are certainly possible.

[4] Indeed, leak detectors frequently use garbage collection as their underlying implementation technology

One of the most important control mechanisms is not a new one: Existing C++ memory reclamation commands always function as normal. In particular, the C++ `delete` command frees memory and calls destructors as expected. This ability to apply both garbage collection and explicit management to the same heap provides some notable benefits.

- Existing code can be easily used without modification in both garbage collected and non-garbage collected contexts.
- The amount of memory that the garbage collector needs to reclaim is greatly reduced, with the potential for greatly improving performance over pure garbage collected languages when space is at a premium or when the average object size is large.
- The Litter Collection scenario from the previous subsection relies on this, as does the Leak Detection scenario.

It is not specified whether garbage collection is enabled for unannotated source code. This allows existing programs to be easily used with leak detectors and litter collectors as described above. A program can explicitly request that garbage collection be enabled or disabled through use of specific annotation keywords.

`gc_forbidden` This keyword specifies that the code may not be used in a garbage collected environment.

`gc_safe` This keyword specifies that the code is safe to use in a garbage collected program or in a program without garbage collection. It is the default, and can be loosely expressed as signifying that the code uses explicit memory management but does not hide pointers from a garbage collector (e.g., by exclusive or'ing two pointers).

`gc_required` This keyword specifies that the code relies on garbage collection to reclaim unused memory.

`gc_strict` This keyword specifies that the non-pointer types declared in the code, such as integers, will not be used to store pointers. This allows the collector to use type information to reduce conservatism. We believe that `gc_strict` C++ programs can make effective use of precise or nearly-precise collection algorithms.

`gc_relaxed` This keyword specifies that the code may store pointers in non-pointer types. `gc_relaxed` code generally requires conservative collection techniques. To maximize compatibility with existing programs, this is the default setting.

While these directives can, and typically will, be applied to entire programs in accordance with a preferred garbage collection policy, the `gc_strict` and `gc_relaxed` annotations can also be applied on a finer grain. In this case, the directive applies to all object definitions of integral type in its scope. Consider the following structure definition.

```
gc_strict struct A {
  A *next;
  B b;
  int video_data[100000000];
};
```

The garbage collector does not need to scan the `video_data` member of `A` objects for pointers, but will need to scan the `next` member and the `b` member. This is very nice because it allows the author of `struct A` to avoid needlessly scanning of 400MB of video data without worrying about whether `B` stores pointers in integral types internally.

By contrast, the `gc_forbidden` and `gc_required` directives apply to the entire program. This restriction arises because pointer chains may traverse memory created by arbitrary combinations of modules. Pointers hidden in one module can make it unsafe for a garbage collector to reclaim memory allocated in another. Therefore, the proposal does not allow the garbage collection decision to be encapsulated so that a module or library can use garbage collection internally without impact on the external program. Symmetrically, if a small garbage collected program loads a large library, such as `libjvm` that provides its own memory management, performance can suffer as the entire `libjvm` heap is needlessly scanned for pointers to the C++ heap.

While there do not appear to be any fundamental obstacles to developing suitable annotations that allow encapsulating the garbage collection decision within a single module or library, this would require more time than was available in the C++0x timeframe. This was a significant contributor to deferring the comprehensive proposal to a subsequent TR, which would likely include such annotations and appeal to the upcoming Modules Technical Report [30] to improve support in mixed environments.

2.3 Finalization

The support of finalization in the proposal was controversial. Hence finalization was eventually split into a separate proposal [12]. We believe it is especially important for C++ that garbage collectors support finalization for a variety of reasons.

- It is fairly common to use per-object locks for thread synchronization. Indeed, Java provides explicit syntax to support this. For some very common operating systems, the lock resource consists of more than just memory in the object. Thus, without finalization, we may lose much of the benefit of garbage collection, at least in multi-threaded programs.
- Opaque objects returned by a third-party library whose interface calls for them to be properly destroyed require finalization.
- Mixing garbage collected and explicitly managed memory may result in memory leaks without finalization. For example, if a `std::vector` member of a garbage class uses a non-garbage collected allocator, the data structures of the vector will be leaked.
- When supporting the "leak detector" use case for garbage collection, finalizers are useful for implementing reports on leaked memory and non-memory resources.

Although tempting, it is not desirable to equate finalizers and destructors because they may be called in different synchronization contexts in multi-threaded programs.[5] In addition, many destructors just release memory and would be wasteful to track with garbage collection enabled.

Finalizer methods are defined by prefixing the class name with `~~` to reflect the kinship with destructors. In many cases, they will just call the corresponding destructor after acquiring appropriate locks.

```
std::mutex m; // Mutex protecting global resource
              // required by destructor
class A {
public:
    ~A(); // destructor
    ~~A() { // finalizer
        std::lock_guard<std::mutex> protect(m);
        ~A();
    }
};
```

Unfortunately, finalization is difficult to support in a way that neither degrades performance of the vast majority of code that does

[5] This issue also arises in other C++ techniques for supporting automatic memory management, as in the use of destructors by C++0x `shared_ptr`s.

not use it, nor is likely to introduce subtle bugs into the majority of code.

As was pointed out in [9] and repeated in [12], there is a fundamental tension between compiler optimizations, and specifically elimination of dead pointer variables, on one side, and easily usable finalization on the other.[6]

To see this, consider a finalizable class that maintains some of the state of each object in an external data structure `E`, which is indexed with an index value stored in the object. Thus `E[`p`->index]` is the state associated with object p. Assume that p's finalizer destroys this state in some way that renders it unsafe to access, e.g. by explicitly deallocating part of it.

Now assume that the final method call on p executes the following code:

```
int i = index;
a:
foo(E[i]);
```

If a garbage collection takes place at point `a`, there is no guarantee that a pointer to p, i.e. to `this` will be visible to the garbage collector. The `this` pointer is not subsequently accessed and hence there is no reason for the compiler to preserve it. Modern calling conventions increasingly allow it to reside only in registers, which are fairly likely to be reused, e.g. to store `i`. Thus p may be finalized at point `a`, and the call to `foo` may fail unexpectedly.

We suggest a spectrum of solutions in [12]. These include

- Forbidding dead-variable elimination on essentially all pointers. Surprisingly to us, this was viewed as having very modest and acceptable cost by some of the compiler experts in the room, though actual measurements are lacking.
- Requiring the programmer to insert `keepAlive()` calls, as is essentially required by both Java and C#.[7]
- A hybrid based on an argument that it is almost always possible to identify when this might be an issue based on static type information.[8]

Unfortunately none of these is unambiguously the correct solution, and the uncertainty clearly contributed to the postponement of this proposal.

3. A Much More Modest Proposal

Although adoption of the comprehensive proposal was moved beyond the next C++ standard, there remained a desire to improve support for garbage collection in the C++0x time frame. This gave rise to our current compromise proposal [14, 22], which has been adopted. Its goal is simply to allow garbage collecting C++ implementations to completely conform to the standard, while ensuring that existing applications either remain compatible with such an implementation or, in rare cases, have an easy path to become compatible with such implementations. It primarily makes three changes to the language and library:

1. It restricts the language so that only "safely derived" (undisguised) pointers to objects allocated with the built-in `new` operator may be safely dereferenced (or deallocated). For example, this prevents a program from storing the only copy of a pointer in a file on disk, and then dereferencing it. This effectively allows implementations to garbage collect objects that appear to the garbage collector to be unreachable, and yet remain fully conforming. However, it provides no guarantees that a garbage collector is provided.
2. It provides a `declare_reachable()` call to explicitly inform a garbage collector that an object must be treated as reachable although all pointers to it are hidden from the collector. Effectively this provides an escape from the above restriction to safely-derived pointers. An `undeclare_reachable()` call is also provided to undo the effect. In a non-garbage-collected implementation, both may be implemented as no-ops.
3. It provides a `declare_no_pointers()` call (along with its inverse) to tell the garbage collector that certain regions of memory never contain valid pointers, and hence do not need to be scanned.

The current proposal only allows garbage collection of objects allocated with the system-provided `new` operator. In particular it does not restrict pointers to `malloc`-allocated objects, although it would clearly be desirable for implementations to provide an option to collect those as well. These were excluded for now, both because the use of `malloc` by some low-level systems code imposes particular implementation challenges, and because we felt that such a change should really involve the C committee, not just the C++ committee.

Even this modest proposal breaks backwards compatibility with a few existing C++ programs. We impose new restrictions on pointer use. Past experience with conservative garbage collectors and leak detectors in C and C++ suggests that the vast majority of code satisfies these restrictions. But small amounts of existing code may need to be updated. This is fundamentally unavoidable: If we want to provide garbage collection, or even reliable leak detection, for existing unmodified code, we must make the assumptions necessary to guarantee that a tracing garbage collector can work.

As a practical matter, we expect such backwards compatibility issues to be minor. We would be very surprised if any implementations in the near term supported only a garbage collected runtime. Effectively, the programmer will be required to make changes, if any are needed, only in order to take advantage of newly provided garbage collection support.

4. Disallowing Access to Unreachable Objects

A major concern with retrofitting a garbage collector to existing C++ code is that the code might "hide pointers" such that the garbage collector cannot see those pointers, and hence reclaims the referenced object, in spite of the fact that the pointer may still be dereferenced. This might occur for two very distinct reasons:

1. The compiler may perform code transformations that temporarily hide pointers to an object. For example, the last reference to `a[i-1]` might be compiled as `a = a - 1, a[i]`. If a garbage collection were to take place just after decrementing the pointer to the beginning of the array `a`, it might reclaim the array before its last access.
2. The programmer may explicitly hide pointers. For example, linked list nodes occasionally store a single link field containing the exclusive-or of the pointers to the previous and next nodes, so that either pointer canbe recovered if the other is known, and hence the list can be traversed in both directions while only requiring a single pointer-sized link field. Unless the garbage collector is somehow informed of this convention, none of the

[6] There are also still ongoing discussions about better addressing this in Java, where we conjecture that a large number (majority?) of deployed classes using finalization exhibit the problem described below in some form.

[7] However few Java and C# programmers seem to understand this fully. And Java currently spells `keepAlive(this)` as `synchronized(this)`, with some added restrictions.

[8] This unfortunately appears to be false in Java, since `java.lang.ref` effectively allows any object to be registered for finalization-like processing.

nodes in the middle of the list will appear referenced to the collector, and hence may be reclaimed prematurely.

The first problem must be addressed by any implementation that supports garbage collection. The techniques for doing so are well-known [5].

The second problem is more interesting. Historically it has been somewhat unclear whether a strictly conforming program could hide pointers in a way that could not be handled through extremely conservative garbage collection techniques. But the recent introduction of the `intptr_t` type makes it clear that it is possible to convert pointers to integers and back, with arbitrary computation on the integers in the middle. Independent of standards conformance issues, such techniques have always been used by a few applications.

Our first attempts were to craft language requiring that all objects allocated with `new` must be reachable from recognizable pointers at all times. However, this would impose significant restrictions on optimization.

Consider the following function:

```
int f()
{
  int *p = new int();
  int *q = (int *)((intptr_t)p ^ 0x555);
a:
  q = (int *)((intptr_t)q ^ 0x555);
  return *q;
}
```

The object allocated at the beginning of the function is clearly reachable via `p` throughout `f()`. Nonetheless, a garbage collection at label `a` might reclaim it, since `p` is dead at that point, and would probably no longer be stored due to dead variable optimizations, while `q` contains only a disguised pointer to the object. This is closely related to the premature finalization problem described above and in [9].

Wholesale restrictions on dead variable elimination were regarded as too intrusive for (at least) the type of modest proposal adopted into the draft standard. C++0x implementations supporting garbage collection will still need to respect the more modest restrictions on optimizations described in [5].

Thus the correct restriction here is to first define a notion of a *safely derived* pointer, that has been derived from a pointer returned by `new`, and then modified only by a sequence of operations such that none of the intermediate results could have "disguised" the pointer. In addition we require that all of these intermediate values only be stored in fields in which they could be recognized as such by the garbage collector. We allow them to be stored in pointer fields, integer fields of sufficient size, and aligned subsequences of `char` arrays.[9]

Note that this is a property of pointer expressions, not values. At the end of the block in our example above `p` and `q` will typically contain the same binary value. Clearly `p` is safely derived, and hence would have been safe to dereference, while `q` is not.

We then insist that only safely derived pointers may be dereferenced or used for deallocation with `delete`.

5. Performance challenges for `declare_reachable`

A call to `declare_reachable(p)` is specified to ensure that the entire allocated object containing the object referenced by `p` be retained even if it appears to be unreachable. More precisely, a call to `declare_reachable(p)` requires that `p` itself be a safely derived pointer, but allows subsequent dereferences of pointers `q` to the same "complete" object as `p`, even if `q` is not safely derived.

This is undone by a call to `undeclare_reachable(r)`, where `r` points to the same complete object as a prior argument `p` to `declare_reachable()`.

`Undeclare_reachable()` returns a safely-derived copy of its argument. If the programmer wants to temporarily hide a pointer, it can be safely done through code such as

```
declare_reachable(p);
p = (foo *)((intptr_t)p ^ 0x5555);
// p is disguised here.
p = undeclare_reachable((foo *)
                        ((intptr_t)p ^ 0x5555));
// p is once again safely derived here and can
// be dereferenced.
```

In a non-garbage-collected implementation both calls turn into no-ops, and hence the resulting object code is very similar to what it would have been in the GC-unsafe version. In a garbage collected implementation `declare_reachable(p)` effectively adds `p` to a global, GC-visible data structure.

Multiple `declare_reachable()` and `undeclare_reachable()` calls on the same object may be nested. Thus the natural implementation keeps a multiset of all pointers that were declared reachable. In our prototype implementation, the multiset is implemented as a chained hash table, with each linked list node containing a small number m of pointers in a single node.

The implementation does however pose several performance challenges, addressed in the next two subsections.

5.1 Arguments need not be equal

The effect of a `declare_reachable()` call may be undone by a call to `undeclare_reachable()` with a slightly different pointer to the same complete object. Since these calls really affect complete objects, this yields a significantly less contorted specification. Since our prototype is implemented in the context of the our conservative garbage collector, which can map pointers to the base address of the object, this is not a fundamental problem. We could simply map every argument to either call to the base of the corresponding heap object.

We expect that in the vast majority of use cases the work to perform this mapping is useless. Even C++ programs typically traffic in pointers to the beginning of the object. Even when they don't, it is very likely that they will use the same pointer value for both calls.

Hence we instead include the original unmodified `declare_reachable()` argument in the multiset representation. If an `undeclare_reachable` call cannot find its argument in the multiset, we map the argument to the base address of the object. Thus `undeclare_reachable` proceeds by trying three increasingly time consuming approaches in turn:

1. We look for the specific argument `x` passed to `undeclare_reachable(x)` in the hash table. For most usage, we expect this to succeed roughly all the time.
2. We look up the base address of the object `x` in its hash bucket, looking up the base address of each object in the bucket as we go. This is not guaranteed to succeed, since we may be looking in the wrong hash bucket. However, the hash function is designed to ignore a few of the least significant bits in the

[9] Both C and C++ allow arbitrary data to be temporarily moved into `char` arrays and relax typing rules to facilitate that.

pointer value[10] Thus if x is close to the corresponding table entry, there is a high probability of the lookup succeeding.

3. We look through all other hash buckets corresponding to addresses within the object x, looking for an object with the same base address. Assuming $O(1)$ elements in each bucket, this process takes $O(s)$ time, where s is the size of the object. Thus its asymptotic complexity is still similar to that of allocating and collecting the object.

5.2 Concurrent access to multiset

We would like to maintain the efficiency of `declare_reachable()` and `undeclare_reachable()` in the presence of threads. We would like to minimize the chances that either becomes a scalability bottleneck when these functions are called from multiple threads.

Unfortunately, it is not easily possible to keep a separate multiset for each thread, since corresponding `declare_reachable()` and `undeclare_reachable()` calls may not be made by the same thread.

We currently address this issue by keeping lightweight per-chain locks for our chained hash table. We dynamically increase the size of the hash table, using the technique described in [17].

This has the disadvantage that the fast path for both calls still requires a lock acquisition and release, which is usually a significant cost. Techniques similar to those in the next section might alleviate that, but it is unclear that these calls will be performance critical enough to warrant that effort.

6. Performance challenges for `declare_no_pointers`

In most garbage-collected programming languages, objects in the heap contain type descriptors that can be used to locate pointers in the object to facilitate tracing during garbage collection. The type descriptors are derived from the type of the object when it was constructed.

In C and C++ , type information itself may be insufficient. For example, a `char` array may be used to temporarily hold a pointer-containing structure. Certain integral types may be used to hold pointers.

We address this issue primarily by assuming a conservative garbage collector that can determine at run-time whether a particular bit-pattern is a valid object address. Nonetheless, such collectors may perform much better, particularly in densely occupied address spaces, if the collector can disregard certain fields when locating pointers during tracing. Large arrays containing compressed, and thus seemingly random, data are the most egregious example of this.

Since we cannot rely on static type information, the C++0x draft instead provides a simple call `declare_no_pointers(`p`, `n`)`. It declares that no pointers are stored in the memory region consisting of n bytes starting at p. More precisely, any pointers that are stored there are no longer considered to have been safely derived, and hence may no longer be dereferenced.

The region passed to `declare_no_pointers()` will typically reside either in the heap, or inside a statically allocated object that would otherwise have been treated as part of the root set. This interface was chosen largely because of its simplicity.

The effect of `declare_no_pointers(`p`, `n`)` may be undone by a call to `undeclare_no_pointers(`p`, `n`)` with the same values. These calls do not nest. A no-pointers range must be unregistered before an object is explicitly deallocated. When an object is garbage-collected, any corresponding no-pointers ranges must be automatically removed.

This interface makes it relatively convenient to exclude address ranges from scanning by invoking `declare_no_pointers()` in an object's constructor, and then to remove the information with `undeclare_no_pointers()` in its destructor. Since these calls apply to any region of memory, the constructor importantly does not need to know whether the object is allocated in the heap, on the stack, or statically. The object layout information does not leak to the client.

Unlike prior approaches, such as those supported by our existing collector (e.g `GC_malloc_atomic`), this call modifies the collector's pointer location information *after* the object has been allocated. This is precisely what makes the interface a much better match for C++ 's constructor and destructor facilities, but it also makes the implementation [11] considerably more challenging.

An implementation could accommodate this kind of dynamic information in one of two ways:

1. It could dynamically store the information in the object. In the normal case for our collector, there is no meta-information stored in the object. Objects allocated with explicit pointer location information are identified by segregating them into a separate "heap block" or chunk, roughly a page. Thus this appeared impossible without adding appreciable overhead, even for the case in which `declare_no_pointers` is unused. It may still be the most effective way to accommodate `declare_no_pointers` for large quantities of small heap objects, so if this interface becomes widely used, this decision may eventually have to be revisited.
2. It can store no-pointers information in a separate data structure, which is processed at garbage collection time. This also promises to add less overhead for information that is added and removed entirely between garbage collections. Since we expect calls to be made from constructors and destructors, we expect this to be the common case for stack allocated objects. We chose this route for now.

In order to accommodate fast addition and removal of no-pointers ranges by multiple independent threads, we go out of our way to allow threads to process the easy cases independently. This is likely to be even more important here than in the case of `declare_reachable()` since we expect no-pointers calls to be significantly more frequent.

6.1 The no-pointers data structure

The data structures required here are again complicated by the fact that matching `declare_no_pointers()` and `undeclare_no_pointers()` calls need not be made by the same thread. To accommodate this without requiring a lock acquisition on every call, we record the information in four separate data structures:

1. Per-thread arrays D_t containing a fixed number of recently added no-pointers ranges. We also store an associated index ID_t. Based on a tag bit, it stores either the index of what might be the next available slot (a hint) or the index of an entry currently being removed by this thread.
2. Per-thread arrays U_t containing a fixed number of no-pointers ranges that were recently removed by thread t, but could not be found in D_t.

[10] To avoid excessive collisions for consecutive allocations, the number of ignored bits shouldn't be much more than roughly $\log(m) + \log(g)$, where m is the number of pointers in a hash chain node, and g is the minimum allocation size.

[11] `Declare_no_pointers` may be correctly implemented as a no-op, but this is not useful.

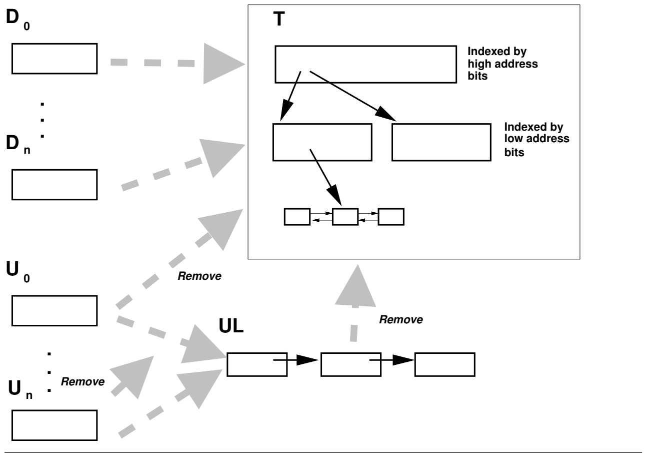

Figure 1. No-pointers data structures

3. A linked list of no-pointers ranges *UL* containing no-pointers ranges representing the combined overflow from all the U_t. The only reason for separating these is that *UL* is protected by a lock shared with the garbage collector, while U_t can be accessed using light-weight lock-free techniques. Since the entire array U_t is processed at once, the separation greatly reduces the number of lock acquisitions.
4. A more complex shared data structure T containing no-pointers ranges that either overflowed from D_t, or were in effect at the last garbage collection.

Figure 1 gives a pictorial representation of the data structure. Wide arrows indicate propagation of data between the different elements. Note that for U_t and *UL*, ”propagation” means that data is removed from T. Ranges are propogated from U_t to UL only if they cannot be found in T for removal. This may happen if they reside in another thread's D_t.

Our garbage collector already uses a two-level tree data structure, reminiscent of hardware page tables, to map each heap chunk to its meta-information. We abstracted this data structure out, so that we could use a separate instance of it to represent T. Since the data structure is indexed by addresses, independent of whether they are inside or outside the heap, it is easy to add addresses in the root set.

Each “leaf” in our two level tree contains a pointer to a sorted doubly-linked list of no-pointers ranges associated with the corresponding chunk-sized region (think page) of memory. No-pointers ranges spanning these regions are split and inserted separately into each list. The pointer at the leaf of the tree may point to anywhere in the list, and is left at the last accessed location, so that searches for ranges corresponding to adjacent memory regions are relatively fast.

6.2 Managing asynchronous thread suspension

Since the shared data structures *UL* and T are currently protected by the main allocator/GC lock, the trickiest “concurrency” issue arises with updates to D_t, the per-thread arrays of no-pointers ranges. Each D_t may be accessed in four different ways:

Insertion Thread t may add an entry by atomically replacing a null element of the array with a pointer to a no-pointers range. No other thread ever replaces a null entry with a non-null one, so this does not require an atomic compare-and-swap. We do insist that the update become visible atomically to a signal handler invoked as the result of a GC request to stop the thread.

Flush Thread t may acquire the allocator/GC lock and flush D_t by copying them into the shared T. This happens when it cannot immediately find an empty slot.

GC Flush The garbage collector may scan and move the D_t contents into T, thus clearing D_t, while thread t is stopped in a signal handler. While doing so, it will ignore (and not clear) an entry marked by ID_t as currently being removed. (Since the removal call has not yet completed, it is acceptable to treat the corresponding no-pointers range as still being in effect.)

Removal Thread t may remove an entry from D_t in response to an `undeclare_no_pointers` call from the same thread on the same range. ID_t is set to the index of the element being removed before the actual removal, and reset afterwards. Suitable care is exercised to prevent compiler reordering of these assignments, since a signal handler must see a consistent state. Hardware memory fences are not required, since there is no real concurrency. The only accesses from outside t occur while t is stopped.

If ID_t were not used to guard against simultaneous removals by the garbage collector and `undeclare_no_pointers()`, the `undeclare_no_pointers()` call may effectively be lost. Unlike `declare_no_pointers()` calls, this is unsafe, since a pointer-containing region may no longer be scanned.

The treatment of U_t is fundamentally similar, but easier, since elements are only removed while holding the allocation/GC lock. Thus this coordination with the garbage collector is not an issue. Note that in the easy cases that we particularly want to optimize, U_t is updated much less frequently than D_t. If `undeclare_no_pointers()` can directly remove an element from D_t it does not touch U_t.

6.3 Handling no-pointers data while tracing

The actual garbage collection code in our collector was only minimally modified. As before, the garbage collector stops other threads during the GC mark phase by sending them signals. (We do not yet handle incremental GC mode well, but it is relatively rarely used.) Once threads are stopped, the collector invokes a routine that transfers all no-pointers information into the shared tree T. This is accomplished by first flushing all D_t arrays into the tree, and then processing all pending removals from both the U_t arrays and *UL*.

Note that this may temporarily result in duplicate or overlapping entries in T, if similar no-pointers ranges are repeatedly added and removed during the GC interval. This is fine; if we are asked to remove a range, we remove the first copy we find.

Since it would be very expensive to constantly check against no-pointers ranges during marking in our scheme, we currently limit the number of such checks in two ways. Both of these effectively cause us to ignore certain `declare_no_pointers()` calls:

1. We currently check for no-pointers regions only the last time we split large blocks in the marker. (Our collector normally does such splitting both in order to avoid mark stack overflows when pushing objects referenced from large blocks, and to facilitate tracing the pieces in parallel.) Currently we split blocks larger than 512 bytes. This normally causes no-pointers regions in the root set, and no-pointers regions containing substantial amounts of compressed data (e.g. images) to be respected. Empirically, these are probably the most important applications. It does render efforts to declare no-pointers ranges inside small heap objects ineffective.
2. We can explicitly ignore no-pointers regions below a certain size. We currently ignore requests of less than 2 words, though we have no real reason to believe that's an appropriate value.

When we do need to look up no-pointers ranges within an object, it is relatively efficient: It takes us as little as two memory references to retrieve the pointer to the doubly linked list of no-pointers ranges for that memory region. And that pointer will typically be positioned close to the address we are looking for. This no-pointers list is then used to trace only those regions outside any of the ranges.

The collector was already structured so that this same scheme handles both root segments and heap objects.

Finally, we added a post-pass to the garbage collector that removes no-pointers regions corresponding to unmarked objects.

7. Evaluation

We expect that nearly all code will pass matching arguments to `declare_reachable` and `undeclare_reachable` nearly all of the time. Thus the performance-critical implementation path reduces to that of implementing a concurrent map. Since there are good known solutions, we expect reasonable performance and processor scalability.

Our initial implementation of `declare_no_pointers()` from above also appears sufficiently useful to justify the facility. We expect it to be tolerant of aggressive use of the facility, and to perform quite reasonably, although it will not take advantage of all such calls. We expect that in many aggressive use scenarios it will not inhibit scalability to a reasonable number of processors, since many scenarios involving very large number of calls to these functions can be handled locally to the thread.

If `declare_no_pointers()` becomes heavily used, it is quite likely that further improvements will be necessary. Clear areas in which one might wish for improvement are reducing the use of the main GC lock when requests cannot be handled locally to the thread, and finding a way to better accommodate small `declare_no_pointers()` requests on heap objects, possibly by using compiler analysis or profile feedback to reserve space for layout information in certain heap objects.

We measured performance on some microbenchmarks to confirm the above observations. The benchmarks were run on a 16 core (4 sockets times 4 cores/socket) server blade using Intel Xeon E7330 processors running at 2.4 GHz. We measured overall completion time of a simple test running various numbers of threads each performing a total of 10 million `declare_` / `undeclare_` operation pairs each. Results are reported as nanoseconds per pair of `declare_` / `undeclare_` operations. Each value represents the average of three runs.

The results for higher thread counts on the `declare_reachable` tests were observed to vary by as much as about 30% from the average. We conjecture that this is attributable to some combination of differences in hash bucket occupancies, their impact on hash table growth, different maximum hash table occupancies due to phase differences between threads, and a locking scheme involving somewhat unpredictable back-offs.

For the `no_pointers` tests, we alternately declared N ranges, and then undeclared all of them, and then repeated the process. We measured with both $N = 10$ and $N = 100$. The former can be handled entirely by the per thread arrays D_t. The latter requires mostly accesses to the global tree, and hence performs much worse.

For the `reachable` tests, we measured a case in which both `declare` and `undeclare` calls were passed pointers to the base of 100 byte objects, and the case in which they were given 40 and 80 byte offsets respectively. We expect the latter to represent a bad case that will rarely or never arise in practice. Somewhat surprisingly, its performance is not that much worse, in part because in our environment the hash function could safely ignore the bottom 8 bits of addresses, so that it was still rare that more than one bucket needed to be considered.

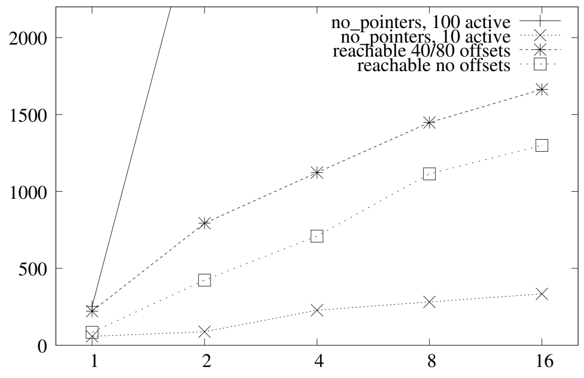

Figure 2. Performance of API implementations, nsecs/op-pair

The results are given in figure 2.[12] For reference, the `declare_reachable` times with different offset are roughly similar to allocation round-trip times. The results for `no_pointers` with 100 simultaneous live ranges are intentionally truncated; the value for 4 threads is almost 9 microseconds.

8. Current Status

The current C++0x committee draft[24] is intended as a vehicle to both allow feedback to the committee, and to make it easier for vendors to implement some of the more time critical features (notably thread support) sooner rather than later. It was not labeled by the committee as "final", and hence was not intended to be approved in its present form. Our current best guess is that an official standard, containing this minimal form of garbage collection support, will be approved in late 2010 or 2011. We hope and expect that a number of vendors will be encouraged to provide garbage collection support, perhaps initially based on our prototype implementation.

Acknowledgments

This work benefited significantly from many in-person and email discussions in the context of the C++ ISO standards committee. Particularly insightful contributions, not always in support of our proposal, were made by Sean Parent, Herb Sutter, and Dave Abrahams. Clark Nelson contributed substantially to the formal proposal accepted into the current draft standard. The anonymous reviewers made many useful suggestions for improving this paper.

References

[1] J. F. Bartlett. Compacting garbage collection with ambiguous roots. *Lisp Pointers*, pages 3–12, April-June 1988.

[2] D. Blake. Programming languages: Everyone has a favorite one. *Dr. Dobb's Journal*, 18(3), April 2008.

[3] H.-J. Boehm. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[4] H.-J. Boehm. Space efficient conservative garbage collection. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 197–206, June 1993.

[5] H.-J. Boehm. Simple garbage-collector-safety. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 89–98, June 1996.

[6] H.-J. Boehm. Fast multiprocessor memory allocation and garbage collection. Technical Report HPL-2000-165, HP Laboratories, December 2000.

[12] Due to a a temporary implementation restriction, these numbers assume only one collecting thread. This should have minimal impact.

[7] H.-J. Boehm. Reducing garbage collector cache misses. In *Proceedings of the 2000 International Symposium on Memory Management*, pages 59–64, 2000.

[8] H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proceeedings of the Twenty-Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 93–100, 2002.

[9] H.-J. Boehm. Destructors, finalizers, and synchronization. In *Proceeedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, pages 262–272, 2003.

[10] H.-J. Boehm. The space cost of lazy reference counting. In *Proceeedings of the 31st Annual ACM Symposium on Principles of Programming Languages*, pages 210–219, 2004.

[11] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, June 1991.

[12] H.-J. Boehm and M. Spertus. N2261: Optimization-robust finalization. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2261.html`.

[13] H.-J. Boehm and M. Spertus. N2310: Transparent programmer-directed garbage collection for c++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2310.pdf`.

[14] H.-J. Boehm, M. Spertus, and C. Nelson. N2670: Minimal support for garbage collection and reachability-based leak detection (revised). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2670.htm`.

[15] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, September 1988.

[16] W. Bright. Digital mars. `http://www.digitalmars.com/`.

[17] B. Cantrill and J. Bonwick. Real-world concurrency. *ACM Queue*, 6(5):16–25, September 2008.

[18] G. Colvin, B. Dawes, P. Dimov, and D. Adler. Boost smart pointer library. http://www.boost.org/libs/smart_ptr/.

[19] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6):527–547, 1994.

[20] J. R. Ellis and D. L. Detlefs. Safe, efficient garbage collection for C++. Technical Report CSL-93-4, Xerox Palo Alto Research Center, September 1993.

[21] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter Usenix Conference*, pages 125–136, 1992.

[22] H. Hinnant. N2771: Lwg issues. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2771.html`.

[23] M. Hirzel and A. Diwan. On the type accuracy of garbage collection. In *Proceedings of the International Symposium on Memory Management 2000*, pages 1–11, October 2000.

[24] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, Programming Language - C++ (Oct 2008 committee draft). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf`.

[25] M. Kapur. Using the c/c++ garbage collection library, libgc. `http://developers.sun.com/solaris/articles/libgc.html`.

[26] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating Systems Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health & Science University, 2004.

[27] G. Rodriguez-Rivera, M. Spertus, and C. Fiterman. Conservative garbage collection for general memory allocators. In *Proceedings of the International Symposium on Memory Management 2000*, pages 71–79, October 2000.

[28] P. Rovner. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, July 1985.

[29] M. Serrano and H.-J. Boehm. Understanding memory allocation of Scheme programs. In *Proceedings of the 2000 International Conference on Functional Programming (ICFP)*, pages 245–256, 2000.

[30] D. Vandevoorde. N2073: Modules in c++ (revision 4). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2073.pdf`.