Precise Garbage Collection for C++ with a Non-Cooperative Compiler

Daniil Berezun
JetBrains
St. Petersburg, Russia
Daniil.Berezun@jetbrains.com

Dmitri Boulytchev
St. Petersburg State University
St. Petersburg, Russia
dboulytchev@math.spbu.ru

ABSTRACT

We describe a garbage collector for C++ implemented as a user-level library, which does not require a cooperative compiler but makes use of a custom garbage-collection-friendly heap implementation. We claim our garbage collector to be *precise*, i.e. capable to precisely identify all pointers to all managed objects, *tolerant* to the conventional C++ manual memory management, and *safe* in the sense that it does not affect the semantics of the program as long as a simple conventions, encouraged by library interface, are followed. We also discuss a performance penalties imposed by the usage of our library on an end-user program.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Processors; D.3.4 [**Memory management (garbage collection)**]: [C++, library, non-conservative]

1. INTRODUCTION

Garbage collection for C++ is a matter of a long-term discussion. Indeed, being positioned as a system-level language C++ provides features and constructs for a fine-tuning of various program traits and properties among which memory management often is of primary importance. So the ability to control memory management discipline is a vital property for C++ which virtually no one is dare to sacrifice. Many additional features of C++ aimed at providing better user-level control on a data representation and implementation of data transformations such as liberal type casts, pointers and pointer arithmetics etc. also make implementation of conventional garbage collection hard, if not impossible.

On the other hand garbage collection is often desirable for a components which are not performance-critical or hard and error-prone to implement using manual memory management. In addition some of the approaches used in garbage collection can be used for verification or debugging purposes as well.

We present a library which implements a precise garbage collection for C++. The library does not require any cooperation from the compiler and so potentially can be used with any compiler complying C++—11 Standard [16]. Our library provides minimalistic and easy-to-use interface, which makes it possible to organize garbage collection for managed objects which were created and operated with respect to that interface. The interface in question does not impose any artificial limitations on C++ constructs or the way they can be utilized. Our approach is safe in the sense that nothing can "go wrong" because of the garbage collection as long as no malicious or accidental intrusion into garbage collector's internal data structures is performed and a simple conventions are respected. Managed objects can coexist with unmanaged and can reference them with no additional cost or limitations; the referencing of managed objects from a non-managed is also possible but requires some user assistance to work. The presence of unmanaged objects and references, however, violates safety conventions which can make the reasonings about safety harder.

Our current implementation works on 64-bit Linux for single-threaded applications, but can be ported to other platforms and made thread-safe with relatively small efforts. We admit a performance issue with our implementation; nevertheless to our knowledge this is the first library-based garbage collector for C++ in its full bloom so we think the analysis of its properties can be beneficial even if it cannot be used yet in a real-world industrial projects.

We also express our gratitude to Maria Kren, Alexander Granovski and Mikhail Aristarkhov for their contribution on the earlier stages of this work, and to Oleg Pliss and anonymous reviewers who provided a precious feedback which helped us to improve the paper.

2. RELATED WORKS

Garbage collection [2, 3] is comprised of a number of principles, approaches and algorithms for automatic memory management.

One class of techniques is based on the idea of *reference counting*. In short, with reference counting every object is equipped with an integer value — reference counter — containing the number of existing references to this object. Pointer assignment operations are tracked in order to maintain the correct values for reference counters. An object is categorized as a garbage when its reference counter hits

zero. Reference counting in its original formulation is rather easy to implement; it can be used in various frameworks for different languages. There are, however, some well-known drawbacks:

- maintaining reference counters impose constant performance overhead on pointer operations;
- reference counting in its original form can not handle *circular references*: a set of mutually-referenced objects will never be categorized as a garbage;
- reference counting can potentially cause unpredictable pauses due to a massive deallocation of objects in a hierarchical data structures.

All these drawbacks were later addressed; however the improved versions of reference counting are not as simple and universally applicable as the original one.

Tracing garbage collection forms another category of approaches. The key observation for tracing garbage collection is that an object (block of memory) which can not be reached from other "useful" objects via references can not be used by a program. As this observation is apparently reflexive some axiomatic predefined *root set* of "useful" objects has to be defined. The exact definition of a root set depends on the architecture and runtime organization. In a typical case the root set consists of objects immediately referenced from the stack and static memory. Note that the "reachability" property conservatively approximates "usefulness" which is undecidable to determine in the majority of interesting cases.

Next, to make tracing garbage collection *precise* there has to be some way to identify references to other objects within given object's data. As a rule some meta-information has to be attached to all objects for this purpose. Precise garbage collector can safely distinguish references from non-references which guarantees that it can (in principle) reclaim all unreachable memory. In contrast, *conservative* garbage collector can only determine potential references using some heuristic conservative tests which makes it possible to overlook garbage.

Finally, garbage collection restricts the set of operations on pointers which are considered legitimate. In the simplest case only assignment and dereferencing are allowed; some approaches make it possible to support address arithmetic at the cost of performance penalty. With these requirements violated the garbage collection becomes *unsafe*, i.e. capable to reclaim utilized memory blocks.

The simplest algorithm for a tracing garbage collection is *mark-and-sweep*. Starting from the root set, all reachable objects are traversed and marked ("mark phase"). Then, during "sweep phase", a memory, occupied by all non-marked objects, is reclaimed. Another basic algorithm is *copying* garbage collection, in which all reachable objects are copied one by one into an area in the heap which was in advance deliberately left free; so after the collection the heap is compactified. More sophisticated approaches include "mark-and-compact", generational garbage collection as well as concurrent and parallel versions and mixtures of those.

An ability of a garbage collector to compactify heap is an important and often desirable property since it allows reducing heap fragmentation, simplifying heap implementation and speeding up memory allocation. However, compactification is possible only in precise garbage collectors since they have to update all relevant references to an object as it moves.

As we can see, in C++ virtually every requirement for a safe and precise garbage collection is violated. Thus, implementing garbage collection for C++ inevitably results in imposing some limitations on the programs of interest. There are three mainstream approaches to garbage collection for C++ [10], which we consider here.

The first approach is to utilize conservative garbage collection. Hans Boehm's garbage collector[1] is a renowned example for C/C++ and virtually the only one which is ready to use with real-world applications. It does not impose essential performance penalties during memory allocation and does not require source code modification to utilize. It is claimed that its performance is usually on a par with conventional garbage collectors. However, some drawbacks have to be mentioned:

- as it does not contain any user-level interface to garbage collection it is hard to justify its safety for a concrete program; the conventions for the safety of garbage collection are purely semantic and can easily be violated accidently;
- a special attention has to be payed to objects' destructors — they are not called by default when corresponding object is collected;
- with this framework all objects become managed — there in no way to create unmanaged objects at all, which in some cases can be undesirable.

Another interesting example is Bartlett's mostly copying garbage collector [7]. Like in the previous case, the root set here is identified conservatively, but an end-user program has to be annotated with certain primitives to help garbage collector identify intra-object pointers, which makes it possible to move objects and compactify the heap. As we can see, this approach requires "cooperative user" to work.

The second class of approaches utilizes the notion of *smart pointer*. Smart pointer [9] is a class which implements conventional pointer interface while performing some actions behind the scene. For example, a reference counting garbage collection can be implemented using smart pointers. Smart pointers are fairly widespread in C++ and are used in a number of frameworks. We can mention [14], where mark-and-sweep garbage collection is organized using smart pointer interface. However, unlike our library, reported garbage collector is not precise and does not support multiple inheritance. As a rule, smart pointers add an

[1] http://www.hboehm.info/gc

essential overhead to all pointer operations which can slow down the program in several times.

Finally, the third approach to a garbage collection for C++ deals with language extension. Strictly speaking, language extension is inevitable for precise and safe garbage collection to be possible [13]. The necessity for language extension can also arise in the context of C++ implementation for platforms which require garbage collection support or make this support beneficial [15]. However, as changing the substrate language, these approaches can not be categorized as garbage collection for C++ *per ce*; in addition all of them require cooperative compiler.

3. GARBAGE COLLECTION LIBRARY INTERFACE

Our garbage collector is implemented as a user-level library, which can be utilized both shared or statically-linked.

Two key entities of the library interface are smart pointer class `gc_ptr`:

```
template <class T> class gc_ptr
```

and memory allocation template function `gc_new`:

```
template <class T, typename ... Types>
gc_ptr<T> gc_new (Types ... types, size_t count = 1)
```

We claim that our garbage collector will correctly manage any program which uses `gc_ptr` instead of regular pointers and `gc_new` instead of regular operator `new`. Conventional operator `delete` can still be used; however, it will not reclaim memory occupied by managed objects. Moreover, we deliberately do not provide any way to make destructors of reclaimed objects be called during garbage collection phase since it would result in semantic flaws similar to those for Java finalizers[2]. With our approach destructors can still be used as conventional dispose primitives with explicitly specified behaviour for all versions of objects' allocation; the only difference between managed and non-managed objects is that for non-managed memory reclamation follows destructor invocation immediately while for managed it is postponed until garbage collection.

Class `gc_ptr` represents a garbage-collection-safe pointer interface:

1. `gc_ptr ()` — default constructor;
2. `gc_ptr (const gc_ptr<T> &p)` — copy constructor;
3. `gc_ptr (const gc_ptr<T> &&p)` — move constructor;
4. `T& operator*() const` — dereference;
5. `T* operator→() const`, `operator T* () const` — access to a pointer;
6. `T& operator [](size_t index)` — array element access;
7. `bool operator== (const gc_ptr<T> &a)`, `bool operator== (const T* a)` — equality checks;
8. `bool operator!= (const gc_ptr<T> &a)`, `bool operator!= (const T* a)` — inequality checks;
9. `gc_ptr& operator = (const gc_ptr<T> &a)` — assignment;
10. `gc_ptr& operator = (const gc_ptr<T> &&a)` — move assignment;
11. `operator bool () const` — `NULL` check;
12. `void nullify ()` — `NULL` assignment.

Function `gc_new` allows us to express all use cases for memory allocation:

1. `gc_new<type> ()` — allocation for atomic types (`int`, `float` etc.);
2. `gc_new<C> ()` — allocation for a single instance of class `C` with default constructor invocation;
3. `gc_new <type> (len)` — allocation for an array of length `len` with elements of atomic type `type`;
4. `gc_new <C> (len)` — allocation for an array of length `len` with instances of class `C` with default constructors invocation for its elements;
5. `gc_new <C,` T_1, T_2, `..> (`a_1, a_2, `..)` — allocation for a single instance of class `C` with non-default constructor invocation with parameters $a_1, a_2, \ldots$ of types $T_1, T_2, \ldots$ respectively.

To make this possible we utilize ellipses and variadic templates — distinctive new feature of C++—11.

As we can see, the only legitimate ways to construct instances of `gc_ptr` are either using copy constructor or `gc_new`. Regular pointers can be exported from `gc_ptr`, but can not be imported into. As an example of this interface utilization we present an implementation of garbage-collected strings (see Fig. 1).

A careful reader may notice that we left overboard one important feature of C++ — it is impossible to acquire a `gc_ptr`, which points into the middle of some object, with presented interface. For example, it is impossible to take an address (in the form of `gc_ptr`) of a structure field or array element. To cope with this deficiency we introduce the following additional function:

```
template <typename F, typename B>
gc_ptr<F> derive (const gc_ptr<B> base,
                  const F* field)
```

[2]https://www.securecoding.cert.org/confluence/display/java/MET12-J.+Do+not+use+finalizers

```
class GCString {
  private:
    gc_ptr<char> pData;
    int length;
    GCString (gc_ptr<char> p, int l) :
      pData (p), length (l) {};
  public:
    GCString () : length (0), pData () {};
    GCString (const char *cString);
    virtual ~GCString () {};
    GCString (const GCString &s);
    GCString operator= (const GCString &s);
    GCString operator= (const char *cString);
    char operator[] (int i) {return pData [i];};
    GCString operator+  (const GCString &s);
    GCString operator+  (const char *cString);
    GCString operator+= (GCString& s);
};

GCString::GCString (const char *cString) {
  length = strlen (cString);
  pData  = gc_new<char> (length+1);
  strcpy ((char *) pData, cString);
}

GCString::GCString (const GCString &s) :
  length (s.length), pData (s.pData) {}

GCString GCString::operator= (const GCString &s) {
  length = s.length;
  pData  = s.pData;
  return *this;
}

GCString GCString::operator= (const char *cString) {
  return *this = GCString (cString);
}

GCString GCString::operator+ (const GCString &s) {
  gc_ptr<char> p = gc_new<char> (length + s.length + 1);
  strcpy (p, (char*) this->pData);
  strcat (p, (char*) s.pData);
  return GCString (p, length + s.length);
}

GCString GCString::operator+ (const char *cString) {
  return *this + GCString (cString);
}

GCString GCString::operator+= (const GCString &s) {
  return *this = *this + s;
}
```

Figure 1: Example: garbage-collected strings

This function allows us to acquire a derived `gc_ptr`; here `base` must point to the beginning of the object, `field` — to a some position "inside" `base`. To ensure safety, `derive` performs a runtime check of aforementioned contract.

Managed by our library objects can coexist with regular unmanaged objects which can be allocated by `new` and freed by `delete` constructs. It is legitimate to keep pointers from managed objects to a non-managed; non-managed objects are completely invisible for the garbage collector. Moreover, it is possible to store a pointers to managed objects inside non-managed. However such pointers have to be manually registered/deregistered using functions

```
void register_object (void *);
void unregister_object (void *);
```

Registering an object immediately makes it an extra root; unregistering immediately removes it from the extra root collection.

Preciseness and safety are two important properties for a garbage collector which should be directly addressed. We postpone, however, the discussion of these properties and first provide an overview of the implementation.

4. IMPLEMENTATION

Our library implements one of the simplest kinds of tracing garbage collection: stop-the-world mark-and-sweep. However even for this case we had to solve the following problems:

- identify root set;
- construct and maintain meta-information;
- implement garbage collection invocation discipline.

An important component of our implementation is cooperative heap. Additionally we maintain a few separate pools. These pools are managed using Linux-specific functions `mmap`/`munmap`[3] which strictly speaking makes our library non-portable. We consider this non-portability as a minor drawback since a similar set of memory-management primitives is available in any general-purpose system.

In the following subsections we describe our implementation in details.

4.1 Cooperative Heap

Cooperative heap is an essential component of our solution. With cooperative heap we could easily implement the following important features:

- distinguish heap pointers from a non-heap;
- implement efficient sweep phase;
- store some garbage-collection-critical information for allocated blocks.

For a base heap implementation we took existing garbage-collection-cooperative heap [4] which in turn was derived from the well-known *Doug Lea's malloc*[4]. Our modifications

[3] http://man7.org/linux/man-pages/man2/mmap.2.html
[4] http://gee.cs.oswego.edu/dl/html/malloc.html

do not compromise the advantages of the original implementation.

Our heap maintains two bits in the header of each allocated block: *managed* bit and *mark* bit. Managed bit reflects the property of the block to be allocated for a managed object. This bit is set within `gc_new` function and cleared within regular `new` and `malloc` which are redefined in our library. Mark bit is unused for a non-managed blocks; for managed it represents conventional mark-and-sweep's mark bit. Two bits are available for free in the implementation of our heap for 64-bit platforms; we presume it is possible to tune it up for 32-bit version as well; however our current implementation is restricted to 64-bits so far.

Managed bit allows us to provide coexistence of managed and non-managed objects; in particular, garbage collector never reclaims memory occupied by non-managed objects; `delete` or `free` never reclaim managed memory.

4.2 Implementation of `gc_ptr`

Implementation of smart pointer class `gc_ptr` plays a crucial role in our library. In this section we consider only `gc_ptr`'s data representation; besides that an important subject is a semantics of its constructors and destructor; we, however, describe them in other sections.

Instances of `gc_ptr` come in two flavors: *simple* and *composite*. Composite instances can only be constructed via `derive` function, while simple — only by `gc_new`.

In either way an instance of `gc_ptr` occupies one machine word and contains an augmented pointer. We make use of two least significant bits (which are always available for free in both 32- and 64-bit models) to distinguish, first, root pointers from a non-root and, second, simple pointers from composite. These bits are masked out as pointer value is exported.

Simple pointers, besides two augmenting bits, hold a regular address. Composite pointers, however, point to an auxiliary two-word data structure — *trampoline* — which is stored in the heap. Two words of a trampoline hold, respectively, the starting address of the allocated object and the derived address within that object. This organization allows us to deal with derived pointers while preserving an ability to find the starting address of an allocated object easily. As trampolines reside in the regular heap, they are managed by the very same garbage collector: indeed, as it is capable to identify all live `gc_ptr`'s, it is equally can identify all live trampolines.

4.3 Constructing and Maintaining Root Sets

Conventional garbage collectors can safely identify root pointers among all other available data values; this identification is performed inside garbage collector and utilizes a certain contract which is respected by a compiler. However, as we do not claim a compiler to be cooperative, we generally can not rely on any assumptions of data layout it provides; moreover, our experience [1] shows that in case of optimizing compiler no safe assumptions can be made at all. Instead, we entrust a managed program with the burden to construct and maintain a root set for us. We do it using constructors and destructors of `gc_ptr`.

We maintain two root sets which reside outside managed heap in a separate pools.

The first is organized as a regular balanced tree with insert-search-delete operations. This tree maintains a collection of *external roots* — those pointers which were explicitly registered using `register_object` primitive (see Section 3). When a non-managed object stores a reference to a managed, the referenced object becomes an external root and has to be explicitly registered. During the mark stage external roots are taken into account like the regular ones. It is left to a user completely to register external roots and unregister them when they are not needed anymore.

The second set of roots is implemented as a stack which holds a regular stack/static roots. Each instance of `gc_ptr` can categorize itself as a root or a non-root by analyzing its own `this`. Since no root can reside in a heap, an instance of `gc_ptr` categorizes itself as a root if it's `this` does not point into the heap (a cooperative heap interface is used for this purpose). The result of categorization is stored into the dedicated bit of `gc_ptr` thus making every instance of `gc_ptr` to remember it. If `gc_ptr` was categorized as a root then it's `this` is pushed onto the root stack. All this work is performed in `gc_ptr`'s constructors. In `gc_ptr`'s destructor root bit is inspected; if it is set then this root has to be removed from the stack. In short, every instance of `gc_ptr`, which categorizes itself as a root, self-registers in it's constructor and self-unregisters in it's destructor.

4.4 Constructing and Maintaining Meta-information

Meta-information in our framework is constructed and maintained in a tight cooperation between `gc_ptr` constructors and function `gc_new`. In addition we utilize Runtime Type Identification (RTTI) to establish a mapping between meta-information for a class and instances of that class.

Meta-information for a class is organized as an array of words of fixed (but specific for a given class) length. Each word of the array corresponds to a single field of type `gc_ptr` and contains an unsigned offset of that field in class' data. Our library maintains a collection of meta-data for each class indexed by its name, acquired via RTTI. This collection resides in a separate pool outside the heap.

Meta-data for a class is constructed when its first instance is created in the heap. The only conventional way to do it in our library is by using function `gc_new`. Moreover, each `gc_ptr` within class in question will eventually has its constructor called. These considerations can be reified into the following protocol:

- `gc_new` initializes some global data structure, allocates a necessary amount of memory in the heap and creates an instance of class of interest using placement new construct;

- a constructor of `gc_ptr` inspects that global data structure to decide if it was called in the context of `gc_new`; if so then it registers its own `this` in that data structure;
- after the execution of placement new operator is completed `gc_new` inspects that global data structure and retrieves a list of all `this`' for all `gc_ptr`s resided within just initialized class instance;
- `gc_new` calculates the offsets for all elements of retrieved list, constructs meta-information data structure and appends it into meta-data collection.

Several refinements and optimizations have to be performed to make this protocol work.

First, there is no need to construct meta-data for classes more then once.

Then, we have to take care of nested calls to `gc_new` which can occur, for example, when an instance of some class is constructed by a constructor of another class. To handle this case we make `gc_new` to save current content of the global data structure into stack in the beginning and to restore it before exiting.

Next, strictly speaking during the construction of a class not only those constructors of `gc_ptr`, which correspond to its fields, can be called. Any constructor may have instances of `gc_ptr` as its local variables or call functions which have local variables of type `gc_ptr`. These instances of `gc_ptr` *must not* register themselves into the global data structure. Fortunately, all these instances reside on a stack which allows us to filter them out easily.

Finally, an instance of a class can enclose instances of other classes (for example, via inheritance or aggregation). We have to make sure that this does not cause any problems with our protocol. Indeed, as these instances never created using `gc_new`, no problem arises.

As we pointed out earlier, no meta-information is created when an instance of a class is constructed on the stack. However this does not cause any problems since all its `gc_ptr`s resided on the stack as well and are categorized as roots.

On the final note, when an instance of a class or array is constructed in the heap we precede its data with two words: the first points to the meta-information for this class, the second holds array's length (1 for single class). Thus we avoid the need to search for a meta-information.

During the mark phase of garbage collection we examine each reachable `gc_ptr` in the following manner:

- if it is composite, we first retrieve the starting address of the object; otherwise it is already a starting address;
- if starting address points to the heap then we know both the number of elements to scan and the pointer to the meta-information which is the same for each element; we use this to retrieve all `gc_ptr`s for the next iteration of marking;
- if starting address points outside the heap we do not know anything else; however in this case all relevant `gc_ptr`s were categorized as roots and were taken into account in the very beginning of mark phase.

4.5 Garbage Collection Invocation Discipline

Another important property of garbage collector is the discipline of its invocation. We reused existing results [5] and integrated them into our implementation.

In [5] four techniques were implemented to determine the moments of garbage collection invocation; a number of environment variables can be utilized by an end user to select certain technique and fine-tune its properties. To make a decision several characteristics of both heap configuration and memory allocation process are analyzed each time a memory management subsystem acquires a control. As original implementation deals with Doug Lea's heap its integration into our library was trivial.

5. PRECISENESS AND SAFETY

Preciseness and safety are two important and desirable properties of garbage collection; when garbage collector is considered as a native and mandatory component of runtime environment these properties are rather considered as requirements. In our case, however, it is impossible to justify neither of them unconditionally. However, as we implement our garbage collector as a library with rather simple interface, we can hope that necessary conditions can be formulated in the form of coding conventions. In this section we address such conventions.

We start from completely managed case — we assume that every object in heap is managed. Clearly nothing wrong can happen as long as we keep all pointers inside `gc_ptr`s. This means that at least

- no object is constructed using operator `new`;
- no address is taken using operator "&".

Both cases are justified by impossibility to turn pointer values into `gc_ptr`s. Unfortunately these two cases do not constitute a sufficient condition. Indeed, even if we can not turn pointers into `gc_ptr` we still can turn `gc_ptr`s into pointers. Two examples illustrate the potential danger. The first is

```
class C {
  X * x = gc_new<X> ();
};
```

Here we created a managed pointer with `gc_new` and immediately store it as a non-managed which well definitely be overlooked by the garbage collector since meta-information for class `C` is empty.

The second example is

```
void f (C& c) {...}

f (*gc_new<C> ());
```

This time we converted `gc_ptr` into a reference; this reference will never be taken into account by the garbage collector.

To eliminate the first case we may introduce a rather natural requirement not to have any pointer declarations besides `gc_ptr` at all. The second case is harder to handle; we can either prohibit using references completely (which is a simple but restricting condition) or prohibit to initialize them with dereferenced `gc_ptr`s (which is less restrictive but much harder to follow).

The situation becomes even worse if we allow unmanaged objects to exist. This time we can not prohibit neither operators `new` and "&" nor pointer declarations to appear in the program. We can only require not to convert managed pointers into unmanaged *and* properly register/unregister managed pointers stored into unmanaged objects.

6. PERFORMANCE

The performance of garbage collection is its vital characteristic as it manages a critical and extensively utilized resource. However, it is unsurprisingly tough to measure a performance impact imposed by a garbage collector to a managed application. Indeed, as a rule garbage collection is deeply integrated with a runtime environment, so it is generally not possible to get rid of its effects. Turning garbage collection off in many cases results in a crush due to running out of memory; increasing memory limits leads to extensive swapping, which does not occur in managed case and compromises performance evaluation results. Modification of a reference application to work without garbage collection (even if possible) also makes evaluation results questionable since strictly speaking the comparison is performed for a *different* programs.

This section presents some considerations and performance evaluation results for our library. We identify three categories of potential performance penalties:

1. an overhead imposed by smart pointer `gc_ptr` and root set maintenance;
2. an overhead imposed by memory allocation function `gc_new` and meta-information maintanance;
3. an overhead imposed by mark-and-sweep phase.

To evaluate the first category overhead we implemented two versions of a reference application which constructs a sets of binary trees in a bottom-up and top-down manner. The first version was written in pure C++ while the second utilized `gc_ptr` but did not use `gc_new` and, so, did not trigger garbage collection. So, we could evaluate the performance penalty imposed by smart pointers and root set maintenance. The results of the evaluation are presented on Fig. 2. As we can see, the first category of penalties results in approximately two-times slowdown, which agrees with a typical overhead for smart pointers.

Since it is impossible to evaluate the isolated overhead of the second category we performed an evaluation of common library overhead: smart pointers, root sets and meta-information construction and maintenance altogether, but with no garbage collection initiated. For this purpose we ran the same benchmarks, this time with `gc_new` as memory allocator and garbage collector switched off. The results are presented on Fig. 3. As we can see, the overall slowdown exceeds the order of magnitude. This evaluation gives us a rather pessimistic worst case behaviour since no operations are performed apart from constructing objects (which is costly with our approach).

We can note that the performance of managed implementation sometimes is better in a bottom-up case. This is due to the fact that in our implementation top-down case requires more pointer operations (first current node is build with a default constructor and then pointers to its subtrees are assigned).

We did not evaluate the performance with garbage collection switched on since our primary interest was to evaluate the overhead of a library-based approach.

7. CONCLUSIONS AND FUTURE WORK

We presented an implementation of garbage collector library for C++. As this library does not require any assistance from a compiler it can potentially be used with many standard compilers and utilized as a non-intrusive component in many frameworks. Despite our current implementation works only on 64-bit Linux it can with a reasonable effort be ported to other platforms as well. Our work yet again demonstrates the expressive power of C++ and its ability to provide enough features to implement system-level functionality on a user level.

Implemented in our library garbage collection is precise in the sense that it can correctly identify all managed pointers and thus capable to reclaim all unreferenced memory. We also describe a conventions which provide safety property of our garbage collection. Finally, our library is compatible with conventional manual memory management at the price of more complex safety requirements.

The performance of garbage collection is our primary concern since the slowdown imposed by our library on an end-user programs sometimes becomes unreasonable. As this implementation is virtually our first attempt we presume it can be optimized to be on a par with performance of other smart pointer libraries.

8. REFERENCES

[1] Daniil Berezun. Root Set Identification and Maintenance for Garbage Collection (in Russian) // JetBrains Programming Languages and Tools Laboratory Reports, Issue 1, 2013.

[2] Richard Jones, Rafael Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, Inc., 1996.

		top-down		bottom-up	
number of trees	tree height	non-GC time (msec)	GC time (msec)	non-GC time (msec)	GC time (msec)
1	26	6582	14671	6798	13649
4	25	14594	31276	14163	28964
16	22	6630	15926	6833	13765
32	21	6578	15280	6870	16311
128	20	13961	13619	14414	28344
512	18	14235	32564	14269	28391
4096	15	14336	32653	14862	28734
1048576	4	1568	3579	1640	3299

Figure 2: Smart pointers and root set maintenance overhead evaluation

		top-down		bottom-up	
number of trees	tree height	non-GC time (msec)	GC time (msec)	non-GC time (msec)	GC time (msec)
1	26	6582	56667	6798	57810
4	25	14594	78024	14163	77853
16	22	6630	54142	6833	62863
32	21	6578	71816	6870	70776
128	20	13961	101172	14414	114022
512	18	14235	141012	14269	126422
4096	15	14336	135879	14862	117410
1048576	4	1568	14671	1640	14744

Figure 3: Library overhead evaluation

[3] Richard Jones, Antony Hosking, Eliot Moss. The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman & Hall/CRC, 2011.

[4] Alexander Samofalov. Mark-and-Sweep Garbage Collector Implementation for LLVM (in Russian). Term paper, Department of Software Engineering, Faculty of Mathematics and Mechanics, St. Petersburg State University, 2014.

[5] Liana Bakradze. Garbage Collection Invocation Discipline (in Russian). Term paper, Department of Software Engineering, Faculty of Mathematics and Mechanics, St. Petersburg State University, 2014.

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Pearson Education, 1994.

[7] Joel F. Bartlett. A Generational, Compacting Garbage Collector for C++ // ECOOP/OOPSLA Workshop on Garbage Collection in Object-Oriented Systems, 1990.

[8] Hans-J. Boehm, Mike Spertus. Garbage Collection in the Next C++ Standard // Proceedings of the International Symposium on Memory Management, 2009.

[9] Igor Semenov. Smart Pointers in C++ (in Russian) // RSDN Magazine, No. 1, 2008.

[10] Herbert Schildt. The Art of C++. The McGraw-Hill Companies, Inc., 2004.

[11] Bjarne Stroustrup. Programming: Principles and Practice Using C++. Addison-Wesley, 2009.

[12] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001.

[13] John R. Ellis, David L. Detlefs. Safe, Efficient Garbage Collection for C++ // Proceedings of the 6th Conference on USENIX Sixth C++ Technical Conference, Vol. 6, 1994.

[14] William E. Kempf. A Garbage Collection Framework for C++. http://www.codeproject.com/Articles/912/A-garbage-collection-framework-for-C.

[15] Herb Sutter. A Design Rationale for C++/CLI // http://www.gotw.ca/publications/C++CLIRationale.pdf, 2006.

[16] Information Technology — Programming Languages — C++. ISO/IEC 14882:2011.