Draft: Precompiling C++ for Garbage Collection

Daniel R. Edelson*

INRIA Project SOR
Rocquencourt, BP 105
78153 Le Chesnay Cedex
France
edelson@sor.inria.fr

26 March 1992

Abstract

Our research is concerned with compiler-independent, efficient and convenient, garbage collection for C++. Most collectors proposed for C++ have either been implemented in a library, or in a compiler. As an intermediate step between those two, this paper proposes using precompilation techniques to augment a C++ source program with code to allow type-accurate garbage collection. In this way, the garbage collector can be more portable and distributable than a collector within a compiler, while simultaneously more convenient (i.e., more practical) than a type-accurate collector that is implemented entirely within a library.

The collector that is under development is based on precompiler-generated *smart pointers* as a replacement for raw pointers in the C++ program. The precompiler emits the smart pointer definitions, and the user is required to utilize them in place of raw pointers. These smart pointers supply functionality that allows the collector to locate all of the roots in the program. The precompiler also generates code that allows the collector to locate internal pointers within objects. This paper describes the architecture of the system, whose first implementation as a simple mark-and-sweep collector is underway. The paper also describes how the collector may eventually be extended with generations.

Keywords

C++, garbage collection, compilers, precompilers, smart pointers, mark-and-sweep, memory management

*Author's other affiliation: Computer and Information Science, University of California, Santa Cruz, CA 95064, USA, *daniel@cse.ucsc.edu*

1 Introduction

C++ is nearly alone among modern object-oriented programming languages in not providing garbage collection. The lack of GC decreases productivity and increases memory management errors. This situation persists principally because the common ways of implementing GC are deemed inappropriate for C++. In particular, tagged pointers are unacceptable because of the impact they have on the efficiency of integer arithmetic, and because the cost is not localized.

In spite of the difficulty, an enormous amount of work has been and continues to be done in attempting to provide garbage collection in C++. The proposals span the entire spectrum of techniques including (not exhaustively):

- concurrent atomic garbage collection implemented in the cfront C++ compiler [Det90],
- library-based object-management including reference counting and mark-and-sweep [Ken91],
- library-based mostly copying generational garbage collection from ambiguous roots [Bar89],
- library-based reference counting through *smart pointers*[1] [Mae92],
- library-based mark-and-sweep garbage collection using smart-pointers [Ede92a]
- compiler-based garbage collection using smart pointers [Gin91],
- library-based mark-and-sweep or copying collection using macros [Fer91], and
- library-based conservative generational garbage collection [BW88, DWH+80].

[1] Smart pointers [Str91, Str87, Ede] are discussed later in this paper.

The vast number of proposals, without the widespread acceptance of any one, reflects how hard the problem is.

The goal of our research is to make type-accurate garbage collection available to the C++ community. This ideal imposes strong restrictions on the collector. Given the speed with which C++ compiler technology and the C++ language definition are advancing, any particular version of any compiler quickly becomes obsolete. In order to be carried along with the evolution of the state-of-the-art, and to be usable by anybody regardless of what compiler they choose, the collector must not be implemented in the compiler.

In the past, we have proposed implementing GC strictly in application-code. It would be something like "GC implemented in a library." The problem with this approach was that it required too much effort from the user. They had to first customize/instantiate the library (a substantial piece of work), and then follow its rules. Overall, this was a tedious and error prone process.

To solve our goal of compiler-independence, while keeping the associated complexity to the user to a minimum, we are now proposing *precompiling* C++ programs to augment them for garbage collection. In essence, the precompiler performs the necessary customization on the C++ program every time it gets compiled. The user still needs to cooperate with the collector, but the number of things to remember (and thus the likelihood of errors) is greatly reduced.

In this paper we discuss our collector architecture and related techniques for supplying garbage collection at the C++ source code level. Then, we describe the transformations that augment a C++ program with the necessary code for it to utilize garbage collection. The remainder of the paper is organized as follows: Section 2 discusses related work in garbage collection and memory management for C++. Section 3 provides an overview of the major techniques that we utilize to implement garbage collection. Section 4 describes the transformations that the proposed precompiler carrys-out to augmented a program for GC. Finally, section 5 concludes the paper.

2 Related Work

There is a significant body of related work, in the general field of GC, in C++ software tools, and specifically in collectors for C++.

2.1 Conservative GC

Conservative garbage collection is a technique in which the collector does not have access to type information so it assumes that anything that might be a pointer actually is a pointer [BDS91, BW88]. For example, upon examining a quantity that the program interprets as an integer (in a register, perhaps), but whose value is such that it also could be a pointer, the collector would assume the value to be a pointer. This is a useful technique for accomplishing garbage collection in programming languages that don't use tagged pointers, and in the absence of compiler support.

Boehm, Demers, et al. describe conservative, generational, parallel mark-and-sweep garbage collection [BDS91, BW88, DWH$^+$80] for languages such as C. Russo has adapted these techniques for use in an object-oriented operating system written in C++ [Rus91a, Rus91b]. Since they are fully conservative, during a collection these collectors must examine every word of the stack, of global data, and of every marked object. In addition, Boehm discusses compiler changes to preclude optimizations that would cause a conservative garbage collector to reclaim data that is actually accessible [Boe91].

Conservative collectors sometimes retain more garbage than type-accurate collectors because conservative collectors interpret non-pointer data as pointers. Often, the amount of retained garbage is small, and conservative collection succeeds quite well. Other times, conservative techniques are not satisfactory. For example, Wentworth has found that conservative garbage collection performs poorly in densely populated address spaces [Wen90, Wen88]. Russo, in using a conservative collector to reclaim dynamic storage used by an object-oriented operating system, has also found that inconveniently large amounts of garbage escape collection [Rus91a]. Lastly, we have tested conservative garbage collection with a CAD software tool called ITEM [Kar89, Ede92b, Ede92a]. This application creates large data structures that are strongly connected when they become garbage. A single false pointer into the data structure keeps the entire mass of data from being reclaimed. Thus, our brief efforts with conservative collection in this application proved unsuccessful.

As these examples illustrate, conservative collection is a very useful technique, but it is not a panacea. Since it has its bad cases, it is worthwhile to investigate type-accurate garbage collection.

2.2 Partially Conservative

Bartlett has written the *Mostly Copying Collector*, a generational garbage collector for Scheme and C++ that uses both conservative and copying techniques [Bar89, Bar88]. This collector divides the heap into logical pages, each of which has a *space-identifier*. During a collection an object can be promoted from from-space to to-space in one of two ways: it can be physically copied to a to-space page, or the space-identifier of its present page can be advanced.

Bartlett's collector conservatively scans the stack and global data seeking pointers. Any word the collector interprets as a pointer (a root) may in fact be either a pointer or some other quantity. Objects referenced by such roots must not be moved because, as the roots are not definitely known to be pointers, the roots can not be modified. Such objects are promoted by having the space identifiers of their pages advanced. Then, the root-referenced objects are (type-accurately) scanned with the help of information provided by the application programmer; the objects they reference are compactly copied to the new space. This collector works with non-polymorphic C++ data structures, and requires that the programmer make a few declarations to enable the collector to locate the internal pointers within collected objects.

Detlefs generalizes Bartlett's collector in two ways [Det90]. Bartlett's collector contains two restrictions:

1. Internal pointers must be located at the beginning of objects, and
2. heap-allocated objects may not contain "unsure" pointers.[2]

Detlefs' relaxes these by maintaining type-specific map information in a header in front of every object. During a collection the collector interprets the map information to locate internal pointers. The header can represent information about both sure pointers and unsure pointers. The collector treats sure pointers accurately and unsure pointers conservatively. Detlefs' collector is concurrent and is implemented in the *cfront* C++ compiler.

2.3 Type-Accurate Techniques

Kennedy describes a C++ type hierarchy called OATH that uses garbage collection [Ken91]. Its collector algorithm uses a combination of reference counting and mark-and-sweep. In OATH, objects are accessed exclusively through references called *accessors*. An accessor implements reference counting on its referent. Thus, the first reclamation algorithm available for OATH objects is reference counting. In addition, the reference counts are used to implement a three-phase mark-and-sweep algorithm that can collect cyclic data structures. The three-phase algorithm proceeds as follows. First, OATH scans the objects to eliminate from the reference counts all references between objects. After that, all objects with non-zero reference counts are root-referenced. The root-referenced objects serve as the roots for a standard mark-and-sweep collection, during which the reference counts are restored.

[2] An unsure pointer is a quantity that is statically typed to be either a pointer or a non-pointer. For example, in "union { int i; node * p; }x;" x is an unsure pointer.

In OATH, a method is invoked on an object by invoking an identically-named method on an accessor to the object. The accessor's method forwards the call through a private pointer to the object. This requires that an accessor implement all the same methods as the object that it references. Kennedy implements this using preprocessor macros so that the methods only need to be defined once. The macros cause both the OATH objects, and their accessors, to be defined with the given list of methods. While not overly verbose, the programming style that this utilizes is quite different from the standard C++ style. Additionally, current compiler technology renders long macros, such as those required for OATH, quite difficult to debug. A precompiler would have substantial benefits over a preprocessor for a system like OATH.

Goldberg describes tag-free garbage collection for polymorphic statically-typed languages using compile-time information [Gol91], building on work by Appel [App89]. Goldberg's compiler emits functions that know how to locate the pointers in all possible (necessary) activation records of the program. For example, if some function $\mathcal{F}$ contains two pointers as local variables, then another function would be emitted to mark from those pointers during a collection. The emitted function would be called once for every active invocation of $\mathcal{F}$, on the stack, upon a collection, to trace or copy the sub-datastructure reachable from each pointer. Upon a collection, the collector follows the chain of return addresses up the run-time stack. As each stack frame is visited, the correct garbage collection function is invoked. A function may have more than one garbage collection routine because different variables are live at different points in the function. Clearly, this collector is very tightly coupled to the compiler.

Yasugi and Yonezawa discuss user-level garbage collection for the concurrent object-oriented programming language ABCL/1 [YY91]. Their programming language is based on active objects, thus, the garbage collection requirements for this language are basically the same as for garbage collection of Actors [Dic, KWN90]. Their position paper describes a process very similar to the one proposed in this paper, namely, translating a source program into another source program that is augmented for GC. The programming paradigms for C++ and ABCL/1 [ANS91, Yon90] are quite different; each introduces its own problems that the collector needs to solve.

Ferreira discusses a C++ library that provides garbage collection for C++ programs [Fer91]. The library supplies both incremental mark-and-sweep and generational copy collection, and supports pointers to the interiors of objects. The programmer renders the program suitable for garbage collection be placing macro definitions at various places in the program.

For example, every constructor must invoke a macro to register the object, and every destructor must invoke the complementary a macro to un-register the object. Another macro must be invoked in the class definition to add GC members to the class, based on the number of base classes of the class. To implement the remembered set for generations (cf 4.3), the collector requires a macro invocation on every assignment to an internal pointer. Similarly to the collector we describe in [Ede92a], this collector requires that the programmer supply a function to locate internal pointers. Ferreira's collector can also scan objects conservatively in order to obviate the need for programmer-coding of this function.

Maeder describes a C++ library for symbolic computation systems based on smart pointers and reference counting [Mae92]. The library contains class hierarchies for *expressions*, *strings*, *symbols*, and other objects that are called *normal*. To improve the efficiency of assignment of reference counted pointers, Maeder uses the address of a discrete object as a replacement for the NULL pointer. The smart pointers support debugging by allowing the programmer to detect dangling references: rather than being deleted, an object is marked *deleted*, and subsequent accesses to the object cause an error to be reported. Other functionality allows the programmer to detect memory leaks, by reporting objects that are still alive when the program terminates.

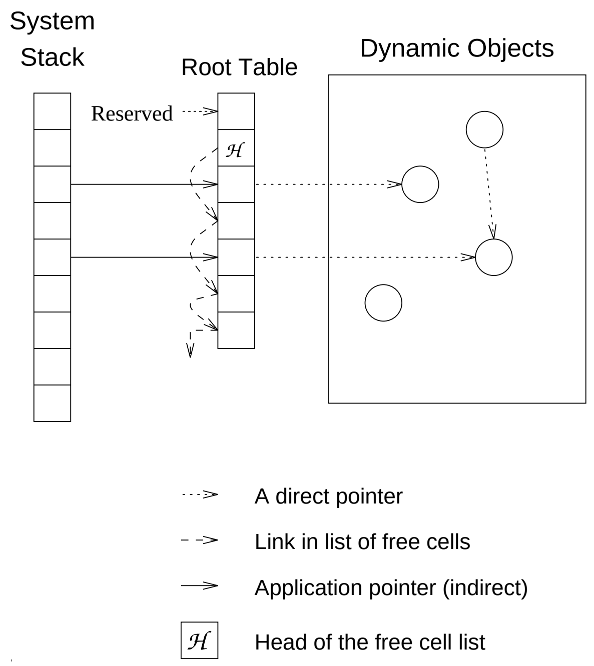

Figure 1: Root indirection

The mutator's roots are indirect through the root table. The root table contains *active* cells and free cells. The free cells are linked into a free list.

3 Overview

The two hard problems that a garbage collector must solve are: 1) finding all the pointers that reference dynamically allocated objects, and 2), locating pointers within objects. Pointers of the first kind are called *roots*; pointers of the second kind are called *internal pointers*.

Our collector locates the roots by requiring that they be indirected through the *root table*. The internal pointers are located using structure tags that are associated with all dynamically allocated objects, and that index information about the type of the object to determine the offsets of their internal pointers. Some pointers are not handled in either of these two ways; they will be discussed later.

3.1 Indirect Root Tables

This collector is based on root indirection. Every root that the application manipulates is indirect through the *root table*. The root table consists of one or more arrays of cells. A cell that is in use, and therefore that contains a direct pointer, is called *active*; a cell that is free is called *inactive*. The inactive cells are linked into a free list. Whenever a cell is required, one is taken from this list. When no free cell is available, a new cell array is allocated. The root table initially consists of a single cell array; but more may be added over a program's lifetime. Figure 1 illustrates a table consisting of a single cell array. In figure 2 the root table has been extended with another array.

When allocating a new cell, it is necessary to be sure that one is available. Normally, this would require a test and a conditional branch. Upon obtaining the pointer to the next free cell, the root table management code would check to be sure that the pointer is not NULL. However, the first thing that is ever done with the new cell is to read it's value to obtain the next link in the free list of cells. We can use that read operation to eliminate the test and conditional branch.

There is always one cell array that was the last to be allocated. This array is special in that its inactive cells are *sequentially* linked into a free list. This must be true, because when the array is allocated, all of its cells are free, and they are all linked into a free list. We *read protect* the last page of this array [AL91]. During program execution, when we attempt to load the link stored in the first cell on the read protected page, the program receives and handles a signal. That signal

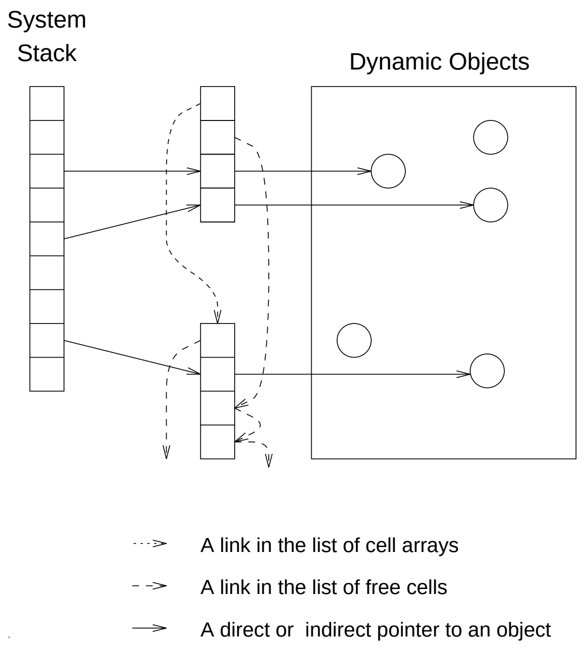

Figure 2: The list of root tables

The root table consists of cell arrays that are linked into a list. The first word of each array contains its link.

---

tells the system to allocate and link in a new cell array. A new diagram of a cell array is presented in figure 3. The shaded area illustrates the read-protected region.

3.2 Smart Pointers

We have indicated that pointers manipulated by the programmer must be indirect through the root table, but we have not indicated how this is accomplished. We require that the programmer use *smart pointers* [Str91, Str87, Ede] in place of pointers. Smart pointers are user-defined C++ class objects that behave like pointers. The smart pointers use operator overloading so that the standard indirection operators, $*$ and ->, can be used to transparently access objects and members through the smart pointers.

There is a discussion of the best way to organize smart pointers for a polymorphic C++ class hierarchy in [Ede]. The basic goal is to support both polymorphism, and smart pointers to `const` objects. To accomplish this, we use two smart pointer classes for every user class. One of the smart pointer classes substitutes for pointers to mutable objects; the other replaces pointers to `const` objects. These two classes are related in that one is a derived class of the other. Specifically, the smart pointer class for mutable objects derives from the smart pointer class for `const` objects. This supplies an implicit type conversion, in the desired direction, between the two classes.

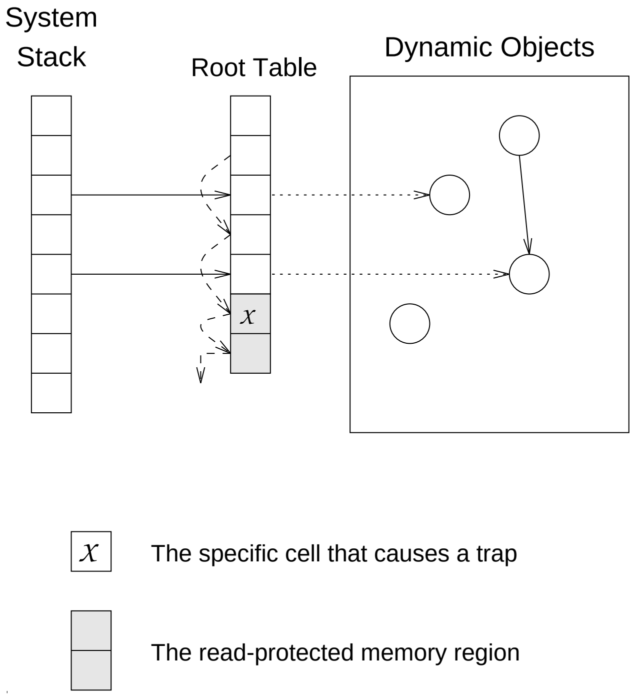

Figure 3: A cell array's protected page

The most recently allocated cell array has its last page read protected. When the protection violation occurs, a new array is allocated and linked to the others.

A similar technique could be used to support implicit pointer type conversions for pointers to base class objects and pointers to derived class objects [Ken91]. However, this fails in the presense of multiple inheritance and permits an erroneous type conversion [Ede]. Therefore, we support smart pointers in a class hierarchy through user-defined type conversions. Figure 4 shows a hierarchy of user-classes, along with the corresponding smart-pointer organization.

The next question is how to insert the smart pointer class definitions into the program. C++ contains a mechanism for implementing *parameterized types*, called `templates`, that seems like the perfect technique. However, generation from templates does not give the smart pointer classes the implicit type conversions that they must have to behave like normal pointers. Thus, templates alone are not sufficient. One way to define them would be through hand coding, abbreviated with preprocessor macros. Grossman [Gro92], for example, uses smart pointers similar to ours for transparent access to objects on disk or across a network. In his system, the smart pointers are defined through hand-coding, but inheritance is used to eliminate some redundant code. We, too, suggested hand coding with macros in [EP91]. However, this is inconvenient and error-prone. This paper proposes that a C++ precompiler generate the smart pointer classes.

3.2.1 Using Smart Pointers

As in [Ede92a], this collector requires that the programmer utilize smart pointer objects in place of raw pointers. A precompiler could detect the declaration and use of raw pointers and substitute smart pointers, but we are initially trying to achieve somewhat more modest goals. Thus, we continue to require that the programmer manipulate objects of type "R_T" rather than raw pointers of type $T*$.

3.3 Internal Pointers

The collector described in [Ede92a] requires users to supply a hand-coded mark() function for each type of collected object. This function locates the internal pointers for the collector. The mark functions are selected through static function overloading.[3] This is possible because the collector uses multiple root tables, in particular, the collector used one root table per static type of smart pointer.

The advantage of this scheme is that absolutely no dynamic type information is required by the collector for non-polymorphic data structures. In particular, there are no structure tags. The main disadvantages of this approach are twofold. Firstly, hand coding is error prone. If a user changes a class definition, they may also need to change the mark() function; failure to do so can lead to obscure gc errors. Secondly, the use of many root tables leads to fragmentation. This wastes memory and decreases efficiency.

In order not to fragment memory among many root tables, this implementation uses only a one root table. Some dynamic mechanism is thus required for identifying internal pointers: dynamic objects are given structure tags. Standard C++ features are used to assign an unique integer to every garbage collected class. When an object is allocated, its tag is passed in to the memory allocator through the overloaded new operator. The tag is stored in the allocator header associated with the object.

Separately, the precompiler locates the offsets of the internal pointers in every collected class, and creates a vector of those offsets. Any unsure pointer [Det90] is treated conservatively, as in Detlefs' collector. Then, the vector of offsets is registered in the memory allocator with the type's tag. This combination of structure tags and internal pointer offsets allows the collector to traverse the data structure.

[3] Combined with virtual functions when necessary.

3.4 Garbage Collection

The collector is currently a stop-the-world mark-and-sweep collector. Collection can be triggered by any one of a number of events. The user can trigger a collection if it is known that a lot of garbage has been produced. Alternatively, the memory allocator can initiate garbage collection because an allocation limit has been reached.

In order to collect garbage, the collector examines every cell in the root table. A cell that is free contains a free-list pointer to another cell. Thus, any cell that points into some cell array of the root table does not contain a direct pointer. By contrast, cells that do not point into the root table contain direct pointers. The mark algorithm is invoked on those cells to set the mark bits associated with the reachable data structure.

The traversal algorithm for marking uses explicit stacking and iteration. Every time an object is visited, its mark bit is tested and set. If the object was previously unmarked, the structure tag in the object's allocation header is used to obtain information about internal pointers. Then, the internal pointers are pushed onto the stack and the algorithm continues with the next pointer in the stack.

4 Precompilation

The precompiler preprocesses a file, parses it, and then performs the following transformations:

- For every class, the precompiler emits smart pointer definitions to replace raw pointers for referring to objects of that class. The smart pointer definitions are prepended to the file.
- Code is inserted to generate a structure tag for every class. The new operator is overloaded for the class so that, every time an object is allocated, the structure tag for the object's type is passed into the allocator where the tag is stored in the allocation header. The class must use this operator new; the user cannot supply another one.
- For every class, the internal pointer offset information is generated. Specifically, the offsets of internal pointers are compiled into a vector and associated with the type's structure tag. Thus, given an object and the corresponding structure tag, the collector can identify the pointer members of the object. These pointer members include not just normal members, but also members of nested types.

In a future step, the programmer could actually program with raw pointers. The precompiler would recode the application to use smart pointers instead of

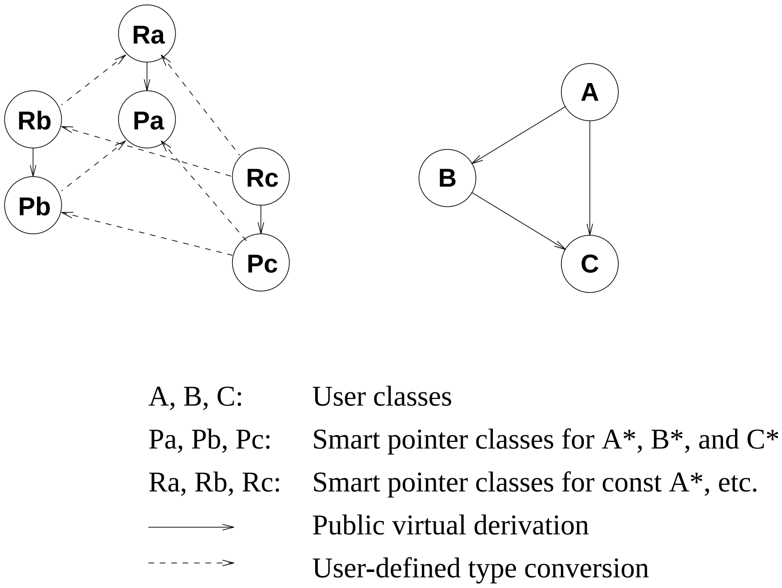

Figure 4: The smart pointer organization for the indicated user classes.

raw pointers. But this is not yet under consideration as the ramifications of this proposal are quite substantial.

4.1 Unsure Pointers

An *unsure* pointer is a datum that might be a pointer or might not [Det90]. For example, in conservative collection, every properly aligned word of program data is an unsure pointer because the collector does not have access to type information, and thus cannot know what actually is a pointer. This collector uses much more type information than a conservative collector, but can not[4] totally eliminate unsure pointers. To deal with them, and also to be efficient, this collector uses conservative techniques in two ways.

One place where this collector is conservative is on pointers that are unsure at the C++ source code level. For example, if an object contains a member that is a `union` of a pointer and a non-pointer, the precompiler would treat that member as a pointer. The second place where the collector is conservative is on `this` pointers.

4.2 `this` pointers

In C++, whenever a method (member function) is invoked on an object, a pointer to the object is passed to the method on the stack. This pointer is called the `this` pointer. Through the `this` pointer the method can access the object's instance data.

These pointers are potentially roots. However, `this` pointers are very common and we don't want to negatively impact their efficiency. Furthermore, they are managed by the compiler, from which this collector is independent. Thus, we do not impose any special behavior, semantics, or indirection on `this` pointers. The solution we propose is to coarsely decode the stack, treating every word that might be a `this` pointer conservatively.

In every activation record on the call stack, if the call is to a member function, then the first function parameter is a `this` pointer. That parameter is always in a known place in a register or in memory. We can decode the stack just sufficiently to examine the first parameter to every function. This parameter will be treated conservatively on the assumption that it may be a `this` pointer.

4.3 Generations

In generational garbage collection, the objects are segregated into *transient* objects and non-transient objects, where the transient objects are expected to become garbage quickly [Ung86, UJ88, Moo84, LH83, DWH+80, Wil90]. The collector concentrates its effort on the transient objects, and does not spend time collecting stable objects that have probably not become garbage. The following observation is exploited to determine which objects are transient and which are

[4] and should not, e.g., `unions`

not: *young objects tend to die young, and old objects tend to persist* [Ung86]. Thus, generational collectors concentrate their effort on the young objects and collect older objects much less frequently.

The difficult problem in implementing the *remembered set*. The remembered set is those references from old objects to young objects, called *back-pointers*. This set is needed in order to collect the young objects, since, when collecting the young objects, the roots are all of the normal pointers (on the stack, in global data, and in registers), and also the pointers contained in old objects that reference young objects. Maintaining the remembered set is that main source of overhead for generational collectors.

Demers et al. [DWH+80] implement the remembered set in a conservative collector as follows. The remembered set is those pages of old space that are thought to contain pointers to the young space. The remembered set is initially empty, and all the pages of the old-space are write-protected using the operating system interface to the virtual memory hardware. Then, every time the mutator writes to a protected old-space page, the program receives an exception that is handled by the collector. The collector unprotects, and adds to the remembered set, the page on which the fault occurs. After this, the mutator can write to the page freely. This allows the collector to identify a superset of the pages that contain back-pointers. During a collection, the collector conservatively scans all of the pages that are in the remembered set. Any page that does not contain at least one pointer to a young object is deleted from the remembered set and re-write-protected.

This technique can be used in our collector. It is our intention to implement it as soon as work on the precompiler and basic memory allocator is sufficiently advanced.

4.4 Status

The design and development of this system are both underway. The smart pointers work, as does most of the code to perform a garbage collection. The precompiler has been prototyped using an existing C++ compiler as the starting point. The modified C++ compiler parses the user's C++ code and emits smart pointers and other declarations. The precompiler does not yet reintegrate the emitted code back into the original source program. The simple non-generational mark-and-sweep collection algorithm, using explicit stacking, has been implemented. The memory allocator and the conservative scan for `this` pointers are being implemented.

5 Conclusions

C++ is a very well designed language considering its goals, however, the complexity of its semantics is daunting. Adding to that complexity by requiring manual storage reclamation makes programming in C++ difficult and error-prone. A widely available garbage collector would be of great benefit to the community.

Many garbage collectors have been proposed for C++ but none has yet gained widespread acceptance. Some are tightly coupled to the compiler; such collectors must be supported by a major C++ compiler vendor, otherwise the compiler that implements the collector will quickly become obsolete. Other collectors are loosly coupled to the compiler. These collectors sometimes require considerable effort and/or impose restrictions for programmers. As an intermediate step between these two extremes, this paper proposes precompiling C++ programs for garbage collection. For type-accurate garbage collection, this is more convenient for the programmer than a pure library-based approach. Simultaneously, this is very portable and does not require that customers utilize any specific compiler.

The collector that we describe is under implementation as an (initially) non-generational mark-and-sweep collector. After it is running, we will immediately add generations, and eventually, concurrency. This has the possibility of providing efficient convenient garbage collection to a large part of the C++ programming community.

Acknowledgements

I would like to thank Peter Dickman for his helpful comments after reading a draft of this paper. I would also like to thank Marc Shapiro for supporting this research.

References

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA (USA), April 1991.

[ANS91] ANSI X3J16/ISO WG21 working document, May 1991. Draft ANSI/ISO standard for the C++ programming language.

[App89] Andrew W. Appel. Runtime tags aren't necessary. In *Lisp and Symbolic Computation*, volume 2, pages 153–162, 1989.

[Ass91] Association for Computing Machinery. *Proceedings of PLDI '91*, June 1991. Published as SIGPLAN v26#6.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, California, February 1988.

[Bar89] Joel F. Bartlett. Mostly copying garbage collection picks up generations and C++. Technical Report TN–12, DEC WRL, October 1989.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of PLDI '91* [Ass91], pages 157–164. Published as SIGPLAN v26#6.

[Boe91] Hans-J. Boehm. Simple gc-safe compilation, 1991. Workshop on garbage collection in OOPLs at OOPSLA '91.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[Det90] David Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie Mellon, 1990.

[Dic] Peter Dickman. Trading space for time in the garbage collection of actors. *Submitted to* OOPSLA '92.

[DWH+80] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of POPL '90*, pages 261–269. Association for Computing Machinery, January 1980.

[Ede] Daniel R. Edelson. Smart pointers: They're smart but they're not pointers. Submitted to the 1992 Usenix C++ Conference.

[Ede92a] Daniel R. Edelson. A mark-and-sweep collector for C++. In *Proceedings of POPL '92*, pages 51–58. Association for Computing Machinery, January 1992.

[Ede92b] Daniel R. Edelson. Comparing two garbage collectors for C++, in unpublished form, 1992.

[EP91] Daniel R. Edelson and Ira Pohl. A copying collector for C++. In *Usenix C++ Conference Proceedings* [Use91], pages 85–102.

[Fer91] Paulo Ferreira. Garbage collection in c++, July 1991. Position paper for the OOPSLA '91 Workshop on Garbage Collection.

[Gin91] Andrew Ginter. Cooperative garbage collectors using smart pointers in the c++ programming language. Master's thesis, Dept. of Computer Science, University of Calgary, December 1991. Tech. Rpt. 91/451/45.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of PLDI '91* [Ass91], pages 165–176. Published as SIGPLAN v26#6.

[Gro92] Ed Grossman. Using smart pointers for transparent access to objects on disk or across a network, 1992. private communication.

[Kar89] Kevin Karplus. Using if-then-else DAGs for multi-level logic minimization. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 101–118, Pasadena, CA, 20-22 March 1989.

[Ken91] Brian Kennedy. The features of the object-oriented abstract type hierarchy (OATH). In *Usenix C++ Conference Proceedings* [Use91], pages 41–50.

[KWN90] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of actors. In *OOPSLA/ECOOP '90 Conference Proceedings*, pages 126–134, October 1990. Published as SIGPLAN v25#10.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Mae92] Roman E. Maeder. A provably correct reference count scheme for a symbolic computation system, 1992. in unpublished form.

[Moo84] David Moon. Garbage collection in a large LISP system. In *SIGPLAN Symposium on Lisp and Functional Programming*, pages 235–246. Association for Computing Machinery, 1984.

[Rus91a] Vincent Russo, 1991. Using the parallel Boehm/Weiser/Demers collector in an operating system: private communication.

[Rus91b] Vincent Russo. Garbage collecting an object-oriented operating system kernel, 1991. Position paper at the OOPSLA '92 Workshop on GC in Object-Oriented Systems.

[Str87] Bjarne Stroustrup. The evolution of C++ 1985 to 1987. In *Usenix C++ Workshop Proceedings*, pages 1–22. Usenix Association, November 1987.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2^{nd} edition, 1991.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *OOPSLA '88 Conference Proceedings*, pages 1–17, September 1988. Published as SIGPLAN v23#11.

[Ung86] David Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. The MIT Press, Cambridge, MA, 1986.

[Use91] Usenix Association. *Usenix C++ Conference Proceedings*, April 1991.

[Wen88] E. P. Wentworth. *An environment for investigating functional languages and implementations*. PhD thesis, University of Port Elizabeth, 1988.

[Wen90] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software—Practice and Experience*, pages 719–727, July 1990.

[Wil90] Paul Wilson. Some issues and strategies and heap management and memory hierarchies, August 1990. Workshop on GC in OOPLs at OOPSLA/ECOOP '90.

[Yon90] Akinori Yonezawa. *An Object Oriented Concurrent System*. The MIT Press, 1990.

[YY91] Masahiro Yasugi and Akinori Yonezawa. Towards user (application) language-level garbage collection in object-oriented concurrent languages, 1991. OOPSLA Workshop on Garbage Collection in OO Systems.