A Mark-and-Sweep Collector for C++

Daniel R. Edelson

University of California
Santa Cruz, CA 95064
USA
daniel@cse.ucsc.edu

INRIA Project SOR
F-78153 Rocquencourt Cedex
France
edelson@sor.inria.fr

Abstract

Our research is concerned with compiler-independent, tag-free garbage collection for the C++ programming language. We have previously presented a copying collector based on root registration. This paper presents a mark-and-sweep garbage collector that ameliorates shortcomings of the previous collector. We describe the two collectors and discuss why the new one is an improvement over the old one. We have tested this collector and a conservative collector in a VLSI CAD application, and this paper discusses the differences. Currently this prototype of the collector imposes too much overhead on our application. We intend to solve that problem, and then use the techniques described in this paper to implement a generational Mark-and-Sweep collector for C++.

1 Introduction

C++ is a modern, object-oriented imperative programming language that has been steadily gaining in use since the mid-1980s [Str91]. C++ supports multiple inheritance. It is primarily statically typed, but a restricted form of dynamic typing allows data structures and functions to be polymorphic within an inheritance hierarchy. C++ has parameterized types (called `templates`) and exception handling.

Programming in C++ is no simple task. The complexity of its semantics has been unfavorably compared to that of Ada. The programming task is further complicated by the lack of automatic storage reclamation, or garbage collection (GC). The programmer must pay attention to reclaiming objects, while at the same time avoiding dangling references.

The absence of GC from C++ results from various design goals of C++. A basic principle of C++ is localized cost (pay for what you use, when you use it). The inclusion of the feature in the language must not impact the efficiency of programs that do not use the feature. Many garbage collectors lack this property, particularly when viewed at the level of the module, rather than the program.

There is already a moderately large body of existing C++ code. A collector that is object-code compatible with existing code, and that can be distributed in library form, will be most useful. In this respect we seek *loose coupling* between the collector and compiler [Det90].

Garbage collection is an integral component of languages such as CommonLisp [Ste84], ML [Wik87], Smalltalk-80 [GR83], and Self [CUL89]. It is also present in other object-oriented imperative languages such as Eiffel [Mey88] and Modula-3 [CDG+88]. Various garbage collectors have been proposed or can be used with C++ [EP91,Bar89,BDS91,Ken91,Det90]. A primary problem that all these collectors solve is locating pointers (roots) on the stack and in global data. We have previously presented a copying collector for C++ [EP91]. In this paper we present a new mark-and-sweep collector and explain why it remedies several shortcomings of our previous work. The new collector is intended to become a general, useful option for those C++ programs that can benefit from GC.

The rest of this paper is organized as follows: Section 2 gives an overview of the new mark-and-sweep collector. Section 3 briefly presents the previous collector, explains its problems, and shows how the new collector solves them. Section 4 compares the features and efficiency of the mark-and-sweep collector to a conservative collector. Section 5 examines related work, and section 6 concludes the paper.

---

2 A Mark-and-Sweep Collector

We have implemented a mark-and-sweep garbage collector for C++. It is currently being tested in a VLSI CAD application that performs logic minimization and optimization. The application makes extensive use of dynamic memory, requiring graph data structures that are mutated over time.

The collector itself consists of a memory allocator, parameterized type definitions, and parameterized functions. It is implemented in a library outside of the compiler.

2.1 Terminology

A pointer that the application manipulates is called a *root*. This includes pointers in global data, on the run-time stack, and in registers. It does not include pointers contained within objects; those are referred to as *internal pointers*.

Any object that is reachable from some root by following a sequence of references is *live*. An allocated object that is not live is *garbage*. The job of the garbage collector is to locate and deallocate every garbage object.

A collector for a statically typed language is called *type-accurate* if every value that the collector interprets as a pointer is statically typed to be a pointer. The opposite of type-accurate is *conservative* [BW88]. Conservative collectors assume that any value that *might* be a pointer actually is a pointer. Partially conservative collectors such as [Bar89] and [Det90] are conservative in certain regions of memory and type-accurate in others. This is described in section 5. The collector described in this paper is type-accurate.

Later in this paper we use the term *virtual function*. In C++ a virtual function is a function that is dynamically bound. Our garbage collector accesses objects through virtual functions in order to support polymorphic data structures.

2.2 Indirection Tables

In order to collect garbage, the collector must be able to locate all the roots. There are several ways of accomplishing this:

Conservative scanning: Conservative garbage collectors examine every word on the stack, in global data, and in the registers. Any word that might be a pointer is interpreted as a pointer. This technique is used to provide GC in languages such as C and C++ in which minimal run-time type information is available. Conservative collection generally precludes copying collection because updating an integer that was interpreted as a pointer would be incorrect.

Tags: Collectors based on tags examine every word on the stack, in global data, and in the registers. Every word has a tag that indicates whether or not it is a pointer. Arithmetic efficiency is reduced for tagged integers; this violates the principle of localized cost. This solution is generally seen as undesirable for languages such as C and C++.

Stack-frame decoding: Garbage collectors based on stack-frame decoding require that the compiler place map information in each stack frame. This **map** indicates what pointers are present as local variables or temporaries in that function invocation. The collector "unwinds" the stack, and interprets the map information that it finds in every activation record. Using this information, it marks the objects reachable from the roots present in that activation record. This solution permits source-level compatibility with existing code: it generally requires recompilation of the libraries.

Root registration: Collectors based on root registration record the addresses of the roots in auxiliary data structures. Collectors based on this technique have the potential for object-level compatibility for existing code. However, there are disadvantages with root registration for copying collection in C++ that are discussed later in this paper.

Root indirection: Collectors based on root indirection permit the application to manipulate only indirect pointers. Each indirect pointer references a direct pointer that is located in a *root table*. During garbage collection, the collector just needs to scan all of the root tables in order to locate the roots. This method, too, has the potential for object-level compatibility between code that uses garbage collection and code that does not. Its disadvantages include the level of indirection and the cost of maintaining the tables.

The collector presented in this paper is based on root indirection. Each root table is an array of cells. A cell currently containing a direct pointer is called *active*; a cell that is free is called *inactive*. The inactive cells are linked into a linked-list. When a cell is required one is taken from this free list. When no free cell is available, a new root table is allocated. The root tables are linked into a linked list. Figure 1 illustrates a single table. Figure 2 illustrates the list of tables.

2.3 Marking and Internal Pointers

During the mark phase of a garbage collection, the collector must traverse all internal pointers and mark

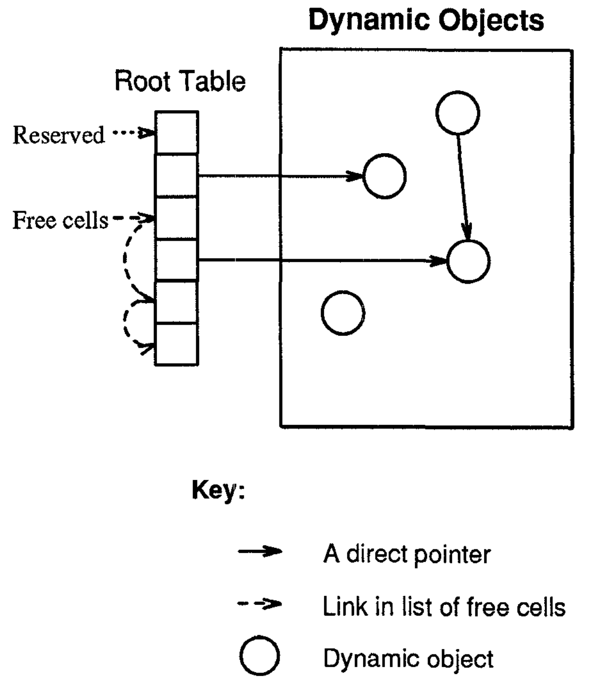

Figure 1: An root table contains direct pointers and free cells. The free cells are linked into a free list. The first cell of each table is reserved for the list of tables.

their referents. There are a number of ways that internal pointers can be identified. Bartlett's collector requires that the internal pointers be located at the beginning of the object, and that a count of the pointers be made available to the collector when the object is allocated [Bar89,Det90]. Detlefs' collector stores map information in an allocator header located immediately before the beginning of the object; the collector interprets this information to locate the internal pointers. Boehm and Weiser's collector scans the entire object conservatively: the size of the object is available in the allocator header preceding the object [BW88].

Our collector associates a virtual *mark* function with every collected type. The mark function is coded or generated specifically for the type and can locate the internal pointers. The collector uses static type information to invoke the mark function on each root-referenced object. The mark function sets the mark bits of an objects and its descendents. Currently the mark functions must be hand-coded. A future implementation of the collector will take the form of a pre-compiler that will generate them automatically.

2.4 The Allocator

Our memory allocator is derived from Lea's *libg++* allocator [Lea91]. There are lists of blocks whose sizes are powers of two as well as intermediate sizes. To satisfy an allocation request the smallest block size that is large enough is used. If no block of the desired size is available, a larger block is broken up, just as in the buddy system [Knu73]. We have modified Lea's allocator to support mark bits and sweeping.

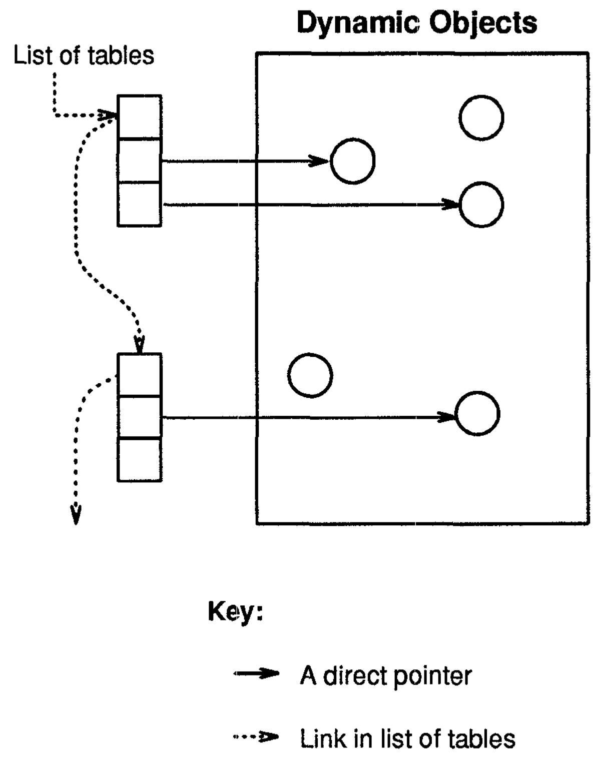

Figure 2: The root tables are themselves linked into a list. The first word of each table contains its link.

The allocator maintains a bitmap with mark bits for all the objects that it can allocate. The mark bits are stored contiguously to improve garbage collection locality.

These functions have been added to the allocator to support collection:

`mark(p)` marks the object referenced by p and returns the previous value of its mark bit.

`marked(p)` returns the value of p's mark bit.

`sweep()` iterates over all the objects, deallocating every allocated, unmarked object—all mark bits are cleared.

2.5 Marking and Sweeping

Marking the live objects is accomplished in the following way: the collector examines every cell in the indirection tables. The status of a cell, *active* or *inactive*, is determined by the cell's value. For every active cell, the sub-data structure that it references is marked. First the cell's referent is marked. Then, recursively, the referent's descendents are marked. At this point the algorithm is recursive for convenience only. Eventually we intend to replace it with a constant-space algorithm such as Detlefs' [Det90].

Sweeping is accomplished with a call to the allocator's sweep() routine. That causes every object that is both allocated and unmarked to be deallocated. In the process, all of the mark bits are cleared. In addition, as described in §2.7, a function call is performed on each object to *finalize* the object immediately before it is deallocated.

2.6 The C++ Interface

The main interface problem is guaranteeing the integrity of the pointer tables. If the application misuses a direct pointer, the next garbage collection might inadvertently collect live data. This is solved using class objects that behave like pointers, and that encapsulate the actual direct pointers. This has been referred to as a *smart pointer* scheme [Ede90,Str87]. A smart pointer can be dereferenced or compared the same way a standard pointer can. When a smart pointer is created it allocates a direct pointer cell from an indirection table. The smart pointer manipulated by the application is a pointer to an indirection table cell. When the smart pointer is destroyed it returns its cell to the free cell list. The creation and destruction semantics are implemented with C++ constructors and destructors. The smart pointers are implemented as parameterized types derived from the types of objects they reference. They overload the C++ indirection operators * and -> in order to simulate direct pointers.

One of the most significant advances of C++ over C is support for polymorphic type hierarchies. A garbage collector for C++ must operate with polymorphic data structures to be most useful. This collector supports polymorphic type hierarchies by using one root type for every type in the hierarchy. The root types support implicit type conversions corresponding to the valid conversions of the raw pointer types.

Access to the standard memory allocator is still permitted for objects such as strings and vectors. The C++ free-store operators (new and delete) are overloaded for collected classes to allocate objects from the collector's memory allocator. Figure 2.6 shows a sample class declaration of a collected class.

This system does not attempt to completely prevent the programmer from acquiring raw (direct) pointers. Indeed, as shall be seen, the careful use of direct pointers represents a useful (if dangerous) optimization. This also provides *weak* pointers[1]. However, the programmer must take care never to have an object whose *only* references are direct pointers; the object would be erroneously garbage collected.

```
class anything {
    ...
public:
    virtual void mark();
    virtual void destroy();
    void * operator new(size_t);
    void operator delete(void * p);
    static void gc();
};
```

Figure 3: A sample collected class.

2.7 Finalization

C++ supports initialization and destruction of objects though *constructors* and *destructors*. A constructor function initializes an object. A destructor is invoked when an object becomes inaccessible to free resources associated with the object. Destructors are essentially synchronous: they execute when the object becomes inaccessible. In the case of local variables, for example, this is when the variable leaves scope.

The use of constructors is perfectly consistent with garbage collection. Destructors, however, present a problem. Precisely when (garbage collected) objects become inaccessible is generally unknown. This renders synchronous destruction impossible [Str91]. Instead, we implement *finalization* [Lam83].

Finalization is essentially the equivalent of an asynchronous destructor. When an object is garbage collected, immediately before its storage is deallocated, the object is finalized. Thus, finalization is used to "clean-up" after an object. For instance, if an object has hardware resources associated with it, finalization is used to free those resources when the object is collected. In our system the destroy member function is called on every object when it is reclaimed by the collector. In order to avoid this overhead, destroy may be defined as a non-virtual (meaning *statically bound*), null inline function.

3 The Previous Copying Collector

The collector that we describe in [EP91] is a copying collector for C++ that is based on root registration. It implements the basic copy collector algorithm as described in [FY69].

We have tested several memory allocators with the copy collector. One memory allocator uses an explicit bounds-check to determine when it is out of

---

[1] A *weak* pointer is a reference that does not cause an object to be retained during garbage collection. The object will be retained only if it is also referenced by a non-weak pointer.

storage. Another allocator uses virtual memory write-protection to avoid the explicit test [App87]. The research that we were duplicating used a programming language with comparatively simple initialization semantics in which every word of a new object is initialized. In C++, on the other hand, the semantics of initialization are almost entirely controlled by the application programmer. In C++ an allocator that uses virtual memory protection to avoid an explicit bounds-check is not worth the added complexity [Ede90].

Since this is a copy collector it moves objects and must be able to update roots. This collector tracks the roots with two data structures. Only one of the two is necessary; the other is present as an optimization.

The two data structures are a doubly-linked list of root pointers and a stack of root pointers. Every time a root is created it inserts its address into one of the two lists. Roots that are allocated and destroyed in a LIFO pattern, such as variables that are local to a function, can be tracked with a stack. These roots push their addresses onto the root stack. Other roots, such as pointers contained within other dynamically allocated objects, can't be tracked with a stack. They insert their addresses into the doubly-linked list. During a collection, every root's address is in one of the two lists. Therefore, the collector can: (1) find the roots, (2) copy all the reachable objects, and (3) update the roots.

Stackable roots (roots whose addresses are tracked with the stack) are about four times as expensive to create and destroy as plain pointers. Roots that are tracked with a doubly-linked list (termed *doubly-linked roots*) are about four times as expensive as stackable roots. Doubly-linked roots can serve in place of stackable roots, but their inefficiency makes that unattractive.

3.1 Problems with the Copy Collector

It turns out that stackable roots are very rarely usable. For example, global pointers, function parameters, and pointer-typed expression temporaries all must be doubly-linked roots. This is because the language does not require LIFO construction and destruction for those objects. For example, if a global variable and a local variable were both stackable roots, the local variable could conceivably be constructed before the global. This can easily occur if the variables are in separately-compiled files. This can lead to the root-stack becoming corrupted. The uselessness of stackable roots brings the efficiency of the copy collector into question.

Another problem involves member functions. 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 in fact roots; as such they must be updated during a collection. This can be accomplished if the addresses of the this pointers are stored in the root doubly-linked list. However, the C++ language definition [X3J91,Str91] forbids taking the address of this pointers. Thus, the collector can only be implemented in a customized compiler; it cannot be implemented and distributed in a library, as is our goal.

The final problem with the copy collector involves the use of two kinds of roots: stackable and doubly-linked. A stackable root requires two words, one for the pointer and one for the link. A doubly-linked root requires three words, one for the pointer and two for links. Suppose an object contains a root as instance data. What kind of root is required? The answer depends on how the object is allocated. For safety's sake doubly-linked roots must be used.

It turns out that, with an implementation in a library, stackable roots can only be used for variables that are local to a function.

3.2 Comparing the Mark-and-Sweep and Copy Collectors

The main similarity between the copy collector and the mark-and-sweep collector is their shared C++ interface. In both cases, the dynamic memory operators new and delete are overloaded so that garbage collected objects are allocated from the collected heap. Smart pointers are used to enable the collector to locate the roots.

In the case of the copy collector the smart pointers are direct pointers that register their addresses in auxiliary lists. Both singly-linked and doubly-linked lists are required.

In the mark-and-sweep collector the smart pointers are indirect through pointer tables. Allocating a smart pointer, unless a new table must be allocated, requires only a singly-linked list operation. There are no doubly-linked lists.

The copy collector requires that this pointers on the stack be updated during each collection. This is illegal if the collector is distributed in a library. Using mark-and-sweep collection the this pointers need only to be examined, not updated. This removes the need for taking the addresses of this pointers.

4 Comparison with a Conservative Collector

We have compared our collector to the conservative collector described in [BW88] in a VLSI CAD application. The application generates and manipulates graphs of dynamically allocated If-Then-Else gates, called *ites*. Given a logical description of a graph, the application creates and optimizes it to minimize the number of gates that it contains.

The ites exist simultaneously in two data structures: a network and a hash table. The network comprises the collected dynamic data structure. The hash table contains a pointer to every ite. However, the hash table entries must be *weak pointers*. That is, if the only reference to an ite is in the hash table, then the ite is garbage and should be collected. Furthermore, it is desirable to remove the hash table entry when an ite is collected.

4.1 Features

The Mark-and-Sweep collector provides all the features needed by the application. The roots manipulated in the application are smart pointers. The hash table contains raw pointers. This prevents hash table entries from retaining ites during collection. Finalization is used to remove the hash table entries when objects are collected.

The conservative collector does not directly support weak pointers. However, it does not scan memory managed by other dynamic memory allocators, including the standard C `malloc` allocator. Therefore, pointers that are dynamically allocated by other memory allocators allocator are weak pointers. We allocate the hash table pointers using `malloc` to make them weak pointers. It is desirable to remove the hash table entries when ites are collected. The conservative collector does not support finalization. The lack of finalization means that garbage collection results in an invalid hash table. Therefore, for these tests, we ran the application for some time with no garbage collection, then garbage collected once and exited the application. This methodology was used for both collectors.

4.2 Efficiency

Our experiments were conducted on a Sun Sparcstation IPC (4/40). We ran the application on fixed input data under each of the two collectors. We measured the CPU time spent running the application and the time spent garbage collecting.

	M-and-S	Conservative
	Small Input	
Application Time	297s	178s
Mark Time	<0.01s	2.12s
Sweep Time	8.13s	0.15s
Data Reclaimed	4MB	146KB
	Large Input	
Application Time	1283s	751s
Mark Time	<0.01s	5.67s
Sweep Time	30s	1.12s
Data Reclaimed	8MB	298KB

Unfortunately, random values in global data kept the conservative collector from reclaiming the data structure. Therefore, we cannot compare the efficiency of the garbage collection process itself. This should be viewed as an unusual case; other results using conservative collection have shown much better reclamation percentages [BDS91].

It is clear from the data that the indirection and overhead of maintaining the root tables in this version of the mark-and-sweep collector imposes a lot of overhead on the application. Indirect pointers are only required when a new unique reference may be created. Determining this fact is harder than indiscriminate use, but still easier than manual reclamation. These results reflect considerable effort to optimize the application by often using direct pointers when possible. We believe that by further limiting the use of indirect pointers we can greatly reduce even this overhead.

The benefits of this type-accurate collector, finalization, weak-pointers, and potentially generations, indicate that it is worthwhile to try and improve the efficiency of the collector.

5 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++. Several of these collectors have been made publically available, as ours will be in the near future.

Boehm et al. have conducted research in conservative garbage collectors [BDS91,BW88]. Their garbage collectors work without any compiler support in languages like C and C++. These collectors are sequential and parallel non-generational mark-and-sweep collectors. Russo has adapted these techniques for use in Choices, a C++ object-oriented operating system toolkit [Rus91,RMC90]. Since they are fully conservative, during a collection they must examine every word of the stack, of global data, and of every marked object.

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 quantify. The root-referenced objects must not be moved because the roots can not be modified. Those objects are promoted by having the space identifiers of their pages advanced. Then the root-referenced objects are (type-accurately) scanned; the objects they reference are compactly copied to the new space. This collector works only with homomorphic data structures, not polymorphic ones.

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.

Kennedy describes a C++ type hierarchy called OATH that uses garbage collection [Ken91]. It's 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 semantics and reference counting on its referent. OATH uses a three-phase mark-and-sweep algorithm. 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 root set for a standard mark-and-sweep garbage collection.

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 collect garbage at various points in the program. Upon a collection, the collector follows the chain of return addresses up the run-time stack. As each stack frame is visited an associated 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.

The collectors by Boehm, Bartlett and Kennedy are implemented in libraries. Goldberg's and Detlefs' collectors must be implemented in a compiler.

6 Conclusions

Garbage collection for C++ is a difficult and important problem. The language philosophy does not permit traditional techniques such as tags. There is a wide variety of alternatives that have been proposed or are possible. These include Bartlett's Mostly Copying collector, Boehm's conservative collectors, Kennedy's reference counting collector, our copying and mark-and-sweep collectors, and many others. There is not yet a generational collector that supports polymorphism, nor has any particular collector gained widespread use.

In this paper we have presented the techniques that support our mark-and-sweep collector for C++. This has been implemented in a library and we are currently testing it in a VLSI CAD application. Our short term goal is to improve its efficiency. Our long-term goals include parallelizing it and supporting generations. This promises to yield a C++ garbage collector that is consistent with the language and useful in a wide variety of applications.

7 Acknowledgements

I would like to express my gratitude to Anne Urban for her detailed comments after reading a draft of this paper. I would like to thank Dirk Coldewey for very timely international logistical support. I would like to thank Soeren Soe for help with the ITEM application.

References

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[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 the SIGPLAN '91 Conference on Programming Language Design and Implementation.* ACM Press, June 1991.

---

[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.

[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 the SIGPLAN '91 Conference on Programming Language Design and Implementation* [Ass91], pages 157–164.

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

[CDG+88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical report, Digital Systems Research Center and Olivetti Research Center, Palo Alto, CA, 1988.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89 Conference Proceedings*, pages 49–70. Association for Computing Machinery, ACM Press, October 1989.

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

[Ede90] Daniel Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, June 1990. M.S. Thesis.

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

[FY69] R. Fenichel and J. Yochelson. A LISP garbage-collector for virtual-memory systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* [Ass91], pages 165–176.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, Reading, MA, 1983.

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

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison, Wesley, Reading, Mass., 1973. Second ed.

[Lam83] Butler W. Lampson. A description of the Cedar language: A Cedar language reference manual. Technical Report CLS-83-15, Xerox PARC, 1983.

[Lea91] Doug Lea. A memory allocator for *libg++*, 1991. private communication.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[RMC90] Vincent Russo, Peter W. Madany, and Roy H. Campbell. C++ and operating systems performance: A case study. In *Usenix C++ Conference Proceedings*, pages 103–114, San Francisco, CA, April 1990. Usenix Association.

[Rus91] Vincent Russo, 1991. Using the parallel Boehm/Weiser/Demers collector in an operating system: private communication.

[Ste84] Guy L. Jr. Steele. *Common Lisp: The Language.* Digital Press, Burlington, MA, 1984.

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

[Str91] Bjarne Stroustrup. *The C++ Reference Manual.* Addison-Wesley, 2nd edition, 1991.

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

[Wik87] Ake Wikstrom. *Functional programming using standard ML.* Prentice Hall, 1987.

[X3J91] ANSI Committee X3J16. Draft standard for programming language C++, May 1991.