Compiler Support for Garbage Collection in a Statically Typed Language*

Amer Diwan *Eliot Moss* *Richard Hudson* †

Object Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

(To appear in SIGPLAN '92 PLDI)

Abstract

We consider the problem of supporting compacting garbage collection in the presence of modern compiler optimizations. Since our collector may move any heap object, it must accurately locate, follow, and update all pointers and values derived from pointers. To assist the collector, we extend the compiler to emit tables describing live pointers, and values derived from pointers, at each program location where collection may occur. Significant results include identification of a number of problems posed by optimizations, solutions to those problems, a working compiler, and experimental data concerning table sizes, table compression, and time overhead of decoding tables during collection. While gc support can affect the code produced, our sample programs show no significant changes, the table sizes are a modest fraction of the size of the optimized code, and stack tracing is a small fraction of total gc time. Since the compiler enhancements are also modest, we conclude that the approach is practical.

1 Introduction

As part of ongoing efforts to implement orthogonal persistence [1] and garbage collection for Modula-3 [2], we have designed and implemented compiler techniques to assist the garbage collector and the persistent memory manager. Our work has been done in the context of Modula-3, but is applicable to other statically typed languages.[1] In the remainder of this section we describe the requirements we had to meet to support garbage collection in our context.

With regards to persistence, our scheme must allow objects to be moved, and possibly removed from main memory altogether, for buffer management [3, 4] purposes. Moreover, since orthogonality allows *any* object to become persistent, *all* objects need to be movable. This requirement is also essential for fully compacting garbage collection (cf. [5, 6]), which yields good locality and fast object allocation time.

Portability to a wide variety of hardware and software platforms is one of the key goals for Persistent Modula-3. Therefore our scheme must not rely on any special hardware support (such as hardware pointer tags).

Our scheme must have minimal impact on run-time performance. Our work is being done in the context of a highly-optimizing compiler.[2] Thus we must not defeat or disallow any compiler optimizations. This is a challenge since the compiler and optimizer are not bound by the rules of the source language and may introduce complex pointer manipulations. We also want to avoid tagging objects except when explicitly required by the language.

In a statically typed language, the compiler knows which global variables contain pointers. It also knows which stack locations and registers contain pointers at any point in a program. In the following sections we describe a technique that exploits this compile-time knowledge to assist the garbage collector in locating and updating pointers in the stack and in the registers, and at the same time meets our requirements: the ability to move objects, portability, and minimal impact on performance. After describing the scheme, we present some experimental results.

2 Basic Problems

Unambiguous full copying collection (cf. [8, 5, 6]) must be able to determine if an object is reachable from other live objects or from the roots. Moreover, the garbage collector must be able to find all pointers to a given object so that they may be updated when the object is moved. These requirements translate to a number of low level requirements on the collector: (i) it must be able to determine the size of heap allocated objects, so that they can be copied; (ii) it must be

*This project is supported by National Science Foundation Grant CCR-8658074, Digital Equipment Corporation, Apple Computer, and GTE Laboratories.

†The authors can be reached electronically via Internet addresses {diwan,moss,hudson}@cs.umass.edu.

[1] The Modula-3 type system allows some dynamism, but type safety of all constructs (except those permitted only in `UNSAFE` modules) can be checked at compile time.

[2] Our compiler is based on `gcc 2.0` and uses its optimizer [7].

able to locate pointers contained in heap objects, so they they can be both traced and updated; (iii) it must be able to locate pointers in global variables; (iv) it must be able to find all references in the stack and in the registers at any point in the program at which collection may occur; (v) it must be able to find objects that are referred to by values created as a result of pointer arithmetic; (vi) and it must be able to update these values when the objects involved are moved.

Modula-3 requires type descriptors in heap objects which makes it straightforward to determine the size of heap allocated objects and to find pointers within them. Thus it is easy to trace the heap. Since Modula-3 is a statically typed language, compile-time location of pointers in global variables is also simple. Locating pointers in the stack and in registers is more difficult, because the stack layout and register assignments may vary even within a procedure. We must also be able to handle compiler temporaries containing pointers, in the stack and in registers.

Updating and following pointers is complicated if pointers do not always point directly to objects. We say a pointer is *tidy* if it points at the object header (or some standard fixed offset from the header). Untidy pointers may be introduced by language features or by compiler optimizations. In Modula-3, pointers to the interior of objects are created by the `VAR` parameter passing mechanism, by the `WITH` statement, and by `SUBARRAY` expressions. Here are examples of optimizations that create untidy pointers:[3]

Strength Reduction: The body

```
A[i] := 13;
INC (i);
```

of an array initialization loop can be turned into `*p++ = 13`, with `p` appropriately initialized.

Virtual Array Origin: If `A` is an array of type `ARRAY [7..13] OF INTEGER`, the obvious method of accessing `A[i]` is:
`*(&A[7] + (i - 7) * sizeof (int))`
The subtraction can be avoided by creating an (untidy) pointer to `A[0]` and using it to index into the array.

Common Subexpression Elimination: The code

```
A[i,j] := 10;
A[i,k] := 20;
```

may be compiled into

```
t = &A[i];
*(t + j * sizeof (int)) = 10;
*(t + k * sizeof (int)) = 20;
```

if the optimizer can determine that `t` is being computed twice and that `i` is not updated.

[3]In our examples we present source code in Modula-3 and compiler/optimizer output in C. Note, though, that our Modula-3 compiler generates assembly code.

Double Indexing: The code

```
A[i] := 1;
B[i] := 2;
```

may be optimized to

```
t1 = &A[0] + (i * sizeof(int));
t2 = &B[0] - &A[0];
*t1 = 1;
*(t1 + t2) = 2;
```

which is useful on machines that have addressing modes with two or more index registers, such as the SPARC.

We use the term *derived value* for any value created by pointer arithmetic, and the term *base value* for any value participating in the derivation. Note that a derived value may be an untidy pointer to the interior of an object (strength reduction example), an untidy pointer that points outside the object to which it refers (virtual array origin example), or even a non-pointer value (double indexing example), and the examples given above may not exhaust the possibilities. In Section 3, we describe a scheme that handles a broad class of pointer arithmetic, which includes all optimizations performed by `gcc`.

3 Solutions

We construct tables at compile time to assist the collector in locating and updating all pointers in the stack and in the registers. We construct one set of tables per *gc-point*. A gc-point is a program point where a collection might occur.[4] An alternative is to use tags or type descriptors in the stack. We decided against tagging stack allocated objects because the stack layout is relatively static, and thus amenable to tabular description, and stack frames are created and destroyed at a high rate, so the overhead of maintaining any kind of descriptors in the stack is likely to be unacceptable.

We construct three kinds of tables for each gc-point in a procedure: *stack pointers*, *register pointers*, and *derivations*. The stack pointers table encodes the locations in the procedure's stack frame that contain live tidy pointers at the gc-point. Likewise, the register pointers table encodes the registers that contain live tidy pointers at that gc-point. The derivations table describes the derivation of all derived values live at the gc-point. In this section we concentrate on the conceptual contents and usage of the tables, and defer consideration of implementation issues to Section 5.

At garbage collection time, the first task is to locate the tables for each frame on the stack. This is done by extracting return addresses from frames and using them to search a table that maps gc-points to gc tables. We can use the stack pointers table directly. Using the register pointers table requires additional information about which registers were

[4]We give details on choosing these points in Section 5.

a

Base Location	Relation
b1	+
b2	−
b3	+

Figure 1: Derivations table for `a` at program point `p`

saved at each call point, so that the register contents can be reconstructed as of the time of the call.

The derivations tables are needed for updating derived values when their base values change. At each gc-point, each live derived location is associated with a table that describes its derivation at that point. For example, Figure 1 shows the derivation table for a variable `a` whose value is derived as:

```
a := b1 + b3 - b2 + E
```

where `E` is some integer expression that does not use pointers or derived values.

There are two steps to updating the derived values. The first step occurs immediately after all the tables have been located. In this step, the value of `E` is calculated and stored in `a`. To calculate `E` we adjust `a` by applying the inverse operation for each base value of `a`:

```
a := a - b1 - b3 + b2
```

Note that the order in which derived values are updated is crucial: a derived value must be updated *before* any of its base values. Thus, if a base value is a derived value itself, then its value must be adjusted *after* that of any values derived from it and *before* any values from which it is derived. We take two measures to ensure this ordering. First, we visit the derivations table of a callee before that of its caller. Second, the derivations tables for a given gc-point are ordered such that the derivations table of a derived value comes before the derivations tables of its base values. Note that circular dependencies cannot occur because derivations are always made from previously calculated base values.

The second step of the update occurs after garbage collection has completed. Its purpose is to reconstruct the derived values from the updated bases. This step uses the new base values to re-derive `a`. In the above example the new values of `b1` and `b3` are added to `a` while that of `b2` is subtracted. Once again the order in which updates occur is important; a value needs to be updated before any values derived from it. This order is exactly the reverse of that required in the previous step.

We have made two assumptions in the design of the derivations table. First, we assume that the base values are live whenever values derived from them are live. This is necessary for us to be able to update the derived values. In Section 4 we show how we ensure this property. As a side effect of this requirement, we never need to follow derived values to find reachable objects. This is because we require the lifetime of a base value to include that of its derived values. Hence, any object reachable via a derived value is also reachable via a non-derived value. Second, we assume that the operations used in the derivation ($+$ and $-$ in this example) have inverses. Invertibility allows us to use the technique outlined above to adjust a derived value if one (or more) of its base values change as a result of collection. In the above example, invertibility allowed us to update `a` given only the base values; no information about `E` was needed. Our current implementation handles two kinds of operations in a derivation ($+$ and $-$),[5] but it can easily be extended to handle other invertible operations as well. Thus, we currently handle all deriving expressions of the form:

$$\sum_i p_i - \sum_j q_j + E$$

where p_i and q_i are pointers or derived values, and E does not involve either pointers or derived values. To handle non-commutative operations, we would need to be careful about the order of the base values in the table for each derivation. To handle non-invertible operations the tables would have to be redesigned to allow the entire deriving expression to be recomputed at run time.

4 Some Complications

The job of the compiler would be simple if it could correctly, statically, and unambiguously identify the base values for each derived value at any given point in the program. Unfortunately, this is not the case for at least three scenarios: (i) when a base value dies before a value derived from it, (ii) when multiple derivations of a value reach a gc-point, and (iii) when indirect references are used as base values in a derivation. In this section we describe each of these problems and present our solutions to them.

The following example illustrates the *Dead Base* problem:

SOURCE

```
A: REF ARRAY [1..10] OF INTEGER;
FOR i := 1 TO LAST (A^) DO
  s := s + A^[i];
END;
```

OPTIMIZED

```
for (i = 1; i <= 10; i++) {
  s = s + *A++;
  <gc-point>
}
```

[5] These are the only operations exploited by the `gcc` optimizer.

If data flow analysis can determine that `A` is dead after the loop, then the compiler may use `A` to efficiently step through the array. In this code, `A`'s base value (the original value of `A`) is not available to the collector inside the loop. Hence, if collection is triggered at `gc-point` then the collector will be unable to update `A`.

We solve this problem by making our compiler consider a use of a derived value as a use of each of its base values.[6] This forces the compiler to retain the base values for the lifetime of the values derived from them. While this can affect performance by increasing the lifetime of variables, which in turn can increases register pressure, we try to minimize its impact by careful selection of base values. When multiple copies of a base value are available, we give preference to stack allocated base values over register allocated ones (to reduce register pressure), and to values in user declared variables over values in compiler temporaries (to shorten temporary lifetimes).

The problem of *Ambiguous Derivations* occurs when multiple derivations of a derived value reach a program point. This is illustrated in the following example:

SOURCE
```
i := 1;
WHILE (cond)
  IF (inv) THEN
    PRINT (P[i]);
  ELSE
    PRINT (Q[i]);
  END;
  INC (i);
END;
```

OPTIMIZED
```
i = 1;
if (inv)
  t = &P[0] + 1;
else
  t = &Q[0] + 1;
while (cond)
  PRINT (*(t + i++));
```

If `inv` is invariant in the loop, the optimizer may hoist the conditional out of the loop causing `t`'s derivation to be ambiguous inside the loop; `t` is derived from either `&P[0]` or `&Q[0]`.

We solve this problem by introducing *path variables* for each ambiguously derived value. The path variable encodes which one of the possible derivations actually happened. The following code segment illustrates it for the example above:

```
i = 1;
if (inv) {
    t′ = <path 1 taken>
    t = &P[0] + 1;
}
else {
    t′ = <path 2 taken>
    t = &Q[0] + 1;
}
while (cond)
  PRINT (*(t + i++));
```

When our compiler detects an ambiguous derivation, it emits tables for each possible derivation; the appropriate derivations table is chosen at run time based on the value of the path variable.

An alternative solution to the ambiguous derivations problem is to use *Path Splitting* similar to Chambers and Ungar [9]. Figure 2 demonstrates this technique. In Figure 2, the body of the loop is duplicated such that the derivation of `t` in each copy of the loop body is unambiguous.

Currently we use the path variable scheme to disambiguate derivations. Both solutions have overheads. The path variable technique adds assignments to the program; the path splitting technique increases the code size and is also more complicated than the path variable scheme. We selected the path variable scheme because it is simpler and we believe ambiguous derivations are rare, and thus the run-time overhead is not significant.

The problem of *Indirect References* occurs when the location of a base value is not known at compile time. This can happen if the base value is obtained by an indirect reference.

SOURCE
```
a:  REF ARRAY [1..5] OF
     REF ARRAY [5..9] OF INTEGER;
foo (a^[2]^[6]);
```

COMPILED
```
foo ( *(a + sizeof (int))
      + sizeof (int))
```

In the example, if the parameter to `foo` is passed by reference, then the expression pushed on the stack is derived from the value in memory location `a + sizeof (int)`. Hence, we cannot determine the location of the base value at compile time. We solve this problem by preserving the intermediate reference in a stack slot or register, thus causing the derivation to refer to a value in a compile-time known location. Indirect references pose a problem only for machines with complicated addressing modes. We expect that this problem will not arise for load/store architectures.

[6] We need to do this only if the derived value is live at some later gc-point.

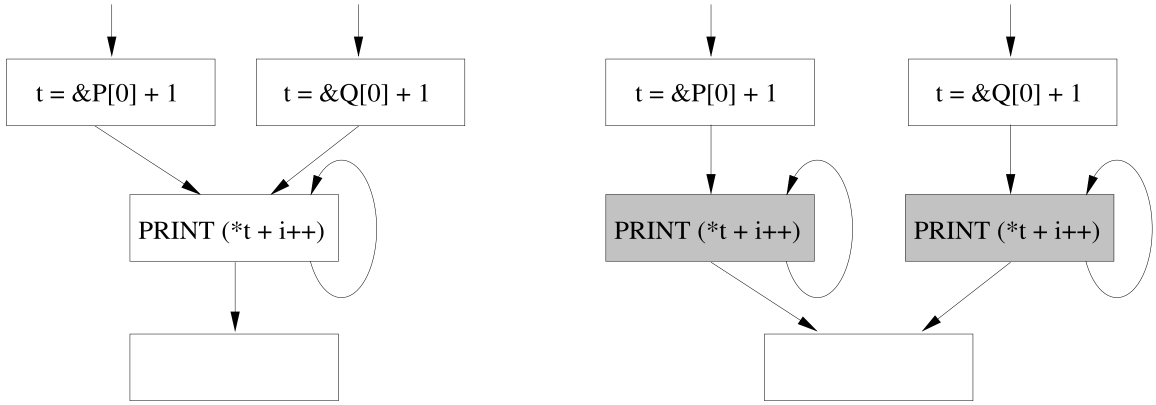

Figure 2: Disambiguating derivations by path splitting

5 Implementation Issues

The organization of the tables and the selection of gc-points might have a significant impact on the performance of our scheme. The tables should be as small as possible but at the same time the collector must be able to extract the information it needs efficiently; compactly encoded tables are likely to have higher decoding overhead. Since tables are emitted at each gc-point, the number of gc-points affects the space overhead of our scheme. Selection of gc-points is especially relevant in a pre-emptive multi-threaded environment. Since a thread switch can occur at any time, we must be prepared to handle a collection when a thread is not at a gc-point. In Sections 5.1, 5.2, and 5.3, we survey some possible solutions to these concerns and justify the choices we have made.

5.1 Table Organization

Storing a list of all live tidy pointers in the stack at each gc-point in a procedure is likely to be expensive. We expect that the variation in stack layout at different gc-points is usually small and thus we consider using *delta* tables at gc-points. A delta table encodes how the information at a given gc-point differs from the information in some other table (called its *ground* table). Our implementation uses a scheme called δ-*main*.

In the δ-*main* scheme, each procedure has a *main* table which describes all slots in the frame of that procedure that contain pointers at some gc-point. Given this, a delta table merely describes which entries of the main table are valid at the gc-point. Since the delta table needs to contain only liveness information, only one bit per entry in the main table is needed per gc-point.

Our current implementation uses the δ-*main* main scheme for stack allocated non-derived pointers only. The registers table has 1 bit per hard register; any attempt to compact this information further is likely to yield little or no improvement. We do not use a delta scheme for the derivations table because in our experience derived values are rare; moreover, they tend to have short lifetimes and thus the information varies

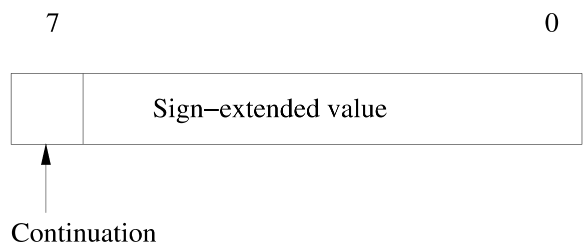

Figure 3: Packing words into bytes.

widely between gc-points. For instance, an important source of derived values in Modula-3 is call-by-reference, which creates derived values that are live at only one gc-point (the call). We therefore store full information for derived values at each gc-point.

In our measurements we observed that delta and registers tables for adjacent gc-points are often identical. Also, many registers tables, many delta tables, and most derivations tables are empty. We keep a descriptor at each gc-point which indicates if any of the tables at that gc-point are empty, or if they are identical to the table at the preceding gc-point.

5.2 Compressing the tables

Despite the compact representation provided by the δ-*main* scheme, we found that the tables were unacceptably large: about 45% of the size of optimized code (see Section 6.1). In this section, we describe the packing techniques that we use to reduce the table sizes to about 16% of the optimized code size.

The stack tracing tables are generated in two phases. The first phase produces tables of 32 bit words. Each memory location is encoded into a word, and the delta tables and register pointers tables occupy an integral number of words.[7] The second phase goes through the table of words and packs

[7] The number of words used for a delta or a register pointers table depends on either the number of entries in the ground table or the number of hard registers.

Figure 4: A ground table entry that fits into 1 byte.

them into bytes. The high bit of each byte determines if it is the last byte in the encoding of a word or if the following byte is also part of the word (see Figure 3). The bytes are stored from most- to least-significant, and the first byte is sign-extended, since many offset (and hence many word values) are negative.

Each entry in a ground table encodes a stack location that is live at some point in the procedure. The low two bits of the encoding identify the base register (FP, SP, or AP, for the VAX). The remaining bits are the offset (in words) from the base register. Most entries in the ground table fit into one byte each (see Figure 4).

Each entry in a derivation table encodes either a register or a memory location. This encoding is more involved than that of the ground table because entries in this table are not restricted to {FP, SP, AP} $\pm$ offset. Thus, most entries in the derivations table require 2 bytes.

The register pointers table contains 1 bit per hard register; most of these tables compact to 1 or 2 bytes each. The delta table contains 1 bit per entry in the ground table. Most of our procedures had fewer than 8 stack allocated pointers, allowing most delta tables to be compressed to 1 byte. Besides the above mentioned tables, we have a descriptor at each gc-point that encodes whether any of the tables at the current gc-point are empty or are identical to those at the previous gc-point. This information packs into 1 byte per gc-point.

At each gc-point we find the appropriate tables by using a mapping from program counter values to gc tables. We compress this by using distances between gc-points in conjunction with the start address of the enclosing module, instead of using 32 bits for the program counter value at each gc-point. The distances are not available until link time; our compiler assumes that distances between adjacent gc points can fit in two bytes. If the distances had been available to our compiler, we would have been able to compress most distances to 1 byte, yielding an additional savings of 1 byte per gc-point.

There is one important consideration that our current implementation does not handle: each pointer contained in an array is treated as a separate variable. We have no way of indicating patterns (e.g., starting from address `a`, the next 200 stack location are pointers). We have a design for compact descriptions of arrays, and it will be simple to add it to the implementation. Our benchmarks did not use any such arrays, so adding this space optimization would not affect the result we report here.

5.3 Selecting GC-Points

Selecting gc-points in a single threaded environment is easy: all calls can be considered gc-points since all allocation is done via a call and hence collection will never be triggered at a non-call point.[8] Of course, some calls do not need to be gc-points. If the compiler performs inter-procedural analysis then it can determine that some procedures never allocate any heap storage and thus calls to them need not be gc-points. In our current implementation all calls except for ones to *non-allocating* procedures are considered gc-points. The non-allocating procedures are statically determined (for instance run-time error reporting routines) rather than via inter-procedural analysis. We may explore refinements in future work.

Selecting gc-points in a multi-threaded environment with pre-emptive scheduling is more challenging since collection may be triggered while threads are suspended at non-gc-points. In our approach, if a thread triggers collection then the suspended threads that are not at gc-points are resumed and allowed to run until they all reach gc-points. We accomplish this by ensuring that resumed threads reach (and block on) a gc-point in a bounded amount of time, and that they do not do any allocations. Ensuring that a resumed thread does not allocate any memory before it reaches a gc-point requires that most calls be gc-points. To avoid an unbounded wait for threads to become ready for collection, we insert a gc-point in all loops that do not have a *guaranteed gc-point* in them. A loop has a guaranteed gc-point if an allocating-procedure or a nested loop is executed at each iteration of the loop, regardless of the path taken through loop. In addition to gc-points at calls and in loops, we need gc-points at places where a thread can block.[9]

6 Results

This paper is not about a fast garbage collection technique. It is about how garbage collection can be assisted by compile-time acquisition of information, and have minimum impact on compiler optimizations. As such, our results are not timings for the garbage collector; they are measurements of the sizes of the compile-time tables generated, the effect our schemes have on compiler optimizations, and the time required to decode the generated tables. In Section 6.1 we give the table sizes for each of our benchmarks, in Section 6.2 we describe the effects of our scheme on the quality of generated code , and in Section 6.3 we report the time required to decode the tables at garbage collection time.

[8] This will not work if allocation is done inline, in which case we must include inline allocations as gc-points.

[9] In most systems these points are call points so they do not need special treatment.

6.1 Table Sizes

We measured table sizes for 4 Modula-3 programs: `typereg`, `FieldList` [10], `takl` [11] and `destroy` [12]. `typereg` implements type registration and type comparisons using structural equivalence for our Modula-3 runtime system. `FieldList` implements command parsing for a UNIX shell. We considered `typereg` and `FieldList` to be good programs to use for our measurements for two reasons. First, they are "real" programs rather than synthetic benchmarks. Second, they consist of a number of short routines with frequent calls. Since we consider most calls as gc-points, we felt that this would represent a worst case scenario. We chose `takl` because it is a well known benchmark. We chose `destroy` because it is heavily recursive and triggers garbage collection frequently, and thus stresses the code that decodes the tables at garbage collection time.

In Table 6.1 we list relevant data about each of the benchmark programs. (The `-opt` suffix indicates that compiler optimizations were turned on.) In Table 6.1 we give the corresponding table sizes, under both the full information and the *δ-main* schemes, with and without each of byte and identical-to-previous compression. Here is the key for interpreting the columns of these tables:

Size Program size in bytes.

NGC Number of gc-points that had non-empty tables.

NPTRS Total number of pointers.

NDEL Number of delta tables emitted.

NREG Number of register pointers tables emitted.

NDER Number of derivations tables emitted.

Plain Table sizes as a percentage of *code size* with no compression.

Previous Table sizes as a percentage of *code size* when a descriptor is used to indicate that a table is identical to that at the previous gc-point.

Packing Table sizes as a percentage of *code size* when byte level packing is used.

PP Table sizes as a percentage of *code size* when both *Previous* and *Packing* are used.

None of our benchmarks had any ambiguous derivations and therefore the compiler introduced no path variables.

From Table 6.1 it can be seen that storing full information at each gc-point (with packing) generally produces larger tables than those produced by *δ-main* (with packing). However the difference is not great. *δ-main* is based on the assumption that procedures have many non-empty gc-points and many live stack allocated pointers at each gc-point. If this is not the case, then storing full information at each gc-point can yield table sizes comparable to *δ-main* without the extra run-time decoding overhead of *δ-main*. However, our measurements indicate that the run-time overhead of decoding these tables is small, so there is little practical benefit to storing full information at each gc-point (see Section 6.3).

For the *δ-main* scheme, both *Packing* and *Previous* tend to reduce table sizes. Applying both *Packing* and *Previous* reduces the table size from about 45% of the size of the optimized code to about 16%.

6.2 Effects on the optimized code

Our schemes have no effect on the *optimized code* produced for any of our benchmarks. There are, however, some instructions introduced in the *unoptimized code*. Most of the differences result from needing to preserve indirect references at gc-points. There are 12 cases where this occurs in `typereg` for the VAX and 32 cases in `FieldList` for the VAX; here is a typical case:

Without gc restrictions
```
addl2 (r7),r0
```

With gc restrictions
```
movl (r7),r1
addl2 r1,r0
```

Our solution to the dead base pointer problem adds two moves to the unoptimized `FieldList`; both are inserted to preserve a clobbered base value.

Note that gc-safety, as proposed by Boehm[10] [13], encounters the same requirement, so this is a basic safety concern rather than a result of our approach. Also, this particular code effect is not likely to occur on load/store architectures.

Compiler support for garbage collection may have other effects on the generated code besides the ones described above. In particular, most generational schemes perform *store checks* [14] when pointers might be written into heap locations. This is a property of the garbage collection scheme[11] and therefore we do not "charge" this overhead to our scheme.

6.3 Timings

While good compression of the gc tables is important for our scheme to be practical, the time to decode those tables must also be reasonable. We do not yet have a complete implementation of the garbage collection run-time, but we have an initial version of stack tracing which we timed on the `destroy` benchmark. `destroy` builds a complete tree of specified branching factor and depth. It then repeatedly builds a new subtree at some fixed intermediate depth, and

[10] Actually, Boehm does not appear to have recognized the indirect reference problem in the work we cite above. He focused on situations that extend the lifetime of a derived pointer but did not address cases where the lifetime of a base pointer might be shortened, e.g., by its being overwritten in the heap.

[11] For instance, page traps could be used instead of store checks to implement generational schemes.

Program	Size	NGC	NPTRS	NDEL	NREG	NDER
typereg	3154	59	87	58	26	3
typereg-opt	2289	52	122	39	41	0
FieldList	4594	51	103	45	18	11
FieldList-opt	3330	82	319	61	70	11
takl	457	8	11	8	6	0
takl-opt	437	9	18	6	9	0
destroy	1240	12	14	11	2	0
destroy-opt	552	14	18	4	13	0

Table 1: Statistics of each of the benchmark programs

	Full Info		δ-*main*			
Program	Plain	Packing	Plain	Previous	Packing	PP
typereg	45.5	14.3	35.0	28.2	12.3	10.6
typereg-opt	51.4	17.2	41.6	35.5	16.0	14.0
FieldList	30.3	11.1	16.4	14.8	6.1	5.6
FieldList-opt	64.7	22.9	53.0	47.6	20.8	18.7
takl	51.6	17.9	41.1	34.1	16.0	14.2
takl-opt	55.8	19.7	43.9	37.5	17.6	15.6
destroy	17.1	5.9	17.1	15.2	6.6	6.1
destroy-opt	46.4	17.4	42.8	38.4	18.1	16.5

Table 2: Table sizes as a percentage of code size

replaces a randomly chosen subtree of the same height with the new subtree. We ran `destroy` in our Smalltalk system, which uses the accurate scavenging scheme [15] we plan to install in the Modula-3 run-time. We found that collections averaged 280 ms of elapsed time. We coded the benchmark in Modula-3 as similarly as possible, and caused "collections" at approximately the same points. To determine stack tracing costs, we ran two versions of the Modula-3 program, one with "collection" being a stack trace, the other with "collection" being a null call, and calculated stack tracing to take 470 μs per collection. However, the difference between the runs was small, and the variance significant even with many repetitions in a system running in single-user mode, so the 90% confidence limit is that stack tracing takes less than 1710 μs per collection for this program. The corresponding numbers per stack frame traced are 27 μs and 98 μs, respectively. We ran these tests on a VAXStation 3500, which is generally rated at 3 to 5 VAX MIPS, suggesting that our current code executes on the order of 100 to 400 VAX instructions per frame traced. We believe we can tighten this up measurably.

Whether one uses the 470 μs per collection figure or the 1710 μs one, there are two additional factors to take into account in comparing stack tracing overhead with overall gc time. First, the `destroy` benchmark is unusually gc intensive. Programs that create a lot of objects, but where most do not survive to the next collection, exhibit something like five times lower gc cost. Also, a Modula-3 collector may be faster than a Smalltalk collector since for Modula-3 we can generate type-specific routines for tracing heap objects, and avoid Smalltalk's object and pointer decoding overhead. We will be generous and allow a factor of two speed up for Modula-3, though we doubt the advantage is really that great. Thus, in less gc-intensive Modula-3 programs, we estimate the ratio of stack tracing time to total gc time to be less than 1710/28000 = 6% (470/28000 = 1.7%). We conclude that stack tracing overhead is only a small part of gc time, even in a high performance scavenging collector.

7 Related Work

Algol-68 implementations were the first to produce compiler generated routines to assist in garbage collection. In the Branquart and Lewi scheme [16], tables are produced that map stack locations to the appropriate garbage collection routine. Unlike our scheme, these tables have to be updated every time a reference to the heap is created on the stack.

Goldberg's compiler [17] produces stack tracing routines. The return address in a call is used to locate a routine that knows how to trace the frame of the caller. His work is not done in the context of an optimizing compiler and thus he does not address many of the issues we handle.

Boehm [13, 18] is currently incorporating garbage collection support in a C compiler. He is using an ambiguous roots collector and his main concern is ensuring that all live objects have at least one pointer to their headers (i.e., there are no live objects that are reachable only from derived values). This problem is similar to our dead base pointer and indirect references scenarios described in Section 4. Since he never moves objects he does not need to deal with the issues in updating derived values.

Exception handling implementations in CLU, Trellis, and Modula-3 also use compiler generated tables. In our Modula-3 implementation [19] tables are generated for each point where an exception may be raised. The tables contain the addresses of handlers for the exceptions that can be raised at that point.

Zurawski and Johnson [20] emit compile-time tables to allow them to construct the unoptimized state of the program from the optimized state. Like us, they have to deal with the effects of pointer arithmetic introduced by the optimizer. Their focus, however, is on debugging; some optimizations are disallowed to make debugging possible. There is a general similarity between the simpler kinds of information we need for garbage collection and what is needed for symbolic debugging in the presence of optimization. Debuggers do not need to update values or handle the derived value cases that we do, however.

8 Conclusions

We have described and evaluated compiler techniques for supporting fully compacting garbage collection in a statically typed language. We started with the following requirements: the ability to move any object, portability, and low run-time overhead. We met these requirements by making extensive use of the information available to the compiler. While we are not the first to recognize the availability of the compile-time information, we believe that we are the first to exploit it so thoroughly in a *highly optimizing compiler*.

9 Acknowledgements

Tony Hosking provided us with garbage collection measurements from the UMass Smalltalk system. We would also like to thank Chuck Lins for his extensive comments on a draft of the paper.

References

[1] M. Atkinson, K. Chisolm, and P. Cockshott, "PS-Algol: an Algol with a persistent heap," *ACM SIGPLAN Not.*, vol. 17, pp. 24–31, July 1982.

[2] G. Nelson, ed., *Systems Programming in Modula-3*. New Jersey: Prentice Hall, 1991.

[3] J. E. B. Moss, "Implementing persistence for an object oriented language," COINS Technical Report 87-69, University of Massachusetts, Amherst, MA 01003, Sept. 1987.

[4] A. L. Hosking, "Main memory management for persistence," Oct. 1991. Position paper presented at the OOPSLA '91 Workshop on Garbage Collection.

[5] J. F. Bartlett, "Compacting garbage collection with ambiguous roots," Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation, Feb. 1988.

[6] J. F. Bartlett, "Mostly-copying garbage collection picks up generations and C++," Technical Note TN-12, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA 94301, Oct. 1989.

[7] R. M. Stallman, *GCC*. Free Software Foundation, Cambridge, MA.

[8] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software: Practice and Experience*, vol. 18, pp. 807–820, Sept. 1988.

[9] C. Chambers and D. Ungar, "Making pure object oriented languages practical," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, (Phoenix, Arizona, Oct. 1991), pp. 1–15, *ACM SIGPLAN Not. 26*, 11 (Nov. 1991).

[10] S. Harbison. Personal Communication, 1992.

[11] R. P. Gabriel, *Performance and Evaluation of Lisp Systems*. Cambridge, MA: MIT Press, 1985.

[12] A. L. Hosking, J. E. B. Moss, and D. Stefanović, "A comparative performance evaluation of write barrier implementations," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, (Vancouver, Canada, Oct. 1992). To appear.

[13] H.-J. Boehm, "A proposal for GC-safe C compilation," Oct. 1991. Position paper for OOPSLA '91 Workshop on Garbage Collection.

[14] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, Pennsylvania, Apr. 1984), pp. 157–167, *ACM SIGPLAN Not. 19*, 5 (May 1984).

[15] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight, "A language-independent garbage collector toolkit," COINS Technical Report 91-47, University of Massachusetts, Amherst, MA 01003, Sept. 1991. Submitted for publication.

[16] P. Branquart and J. Lewi, "A scheme for storage allocation and garbage collection in Algol-68," in *Algol 68 Implementation* (J. E. L. Peck, ed.), North-Holland Publishing Company, 1971.

[17] B. Goldberg, “Tag-free garbage collection in strongly typed programming languages,” in *Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*, (Toronto, Ontario, Canada, June 1991), pp. 165–176, *ACM SIGPLAN Not. 26*, 6 (June 1991).

[18] H.-J. Boehm, “Personal communication,” July 1991.

[19] A. Diwan, “Exception handling in Modula-3.” Internal OOS Document, 1990.

[20] L. W. Zurawski and R. E. Johnson, “Debugging optimized code with expected behavior,” *ACM Trans. Programming Languages and Systems*, To appear.