Simple Garbage-Collector-Safety

Hans-J. Boehm
Xerox PARC

boehm@parc.xerox.com

Abstract. A conservative garbage collector can typically be used with conventionally compiled programs written in C or C++. But two safety issues must be considered. First, the source code must not hide pointers from the garbage collector. This primarily requires stricter adherence to existing restrictions in the language definition. Second, we must ensure that the compiler will not perform transformations that invalidate this requirement.

We argue that the same technique can be used to address both issues. We present an algorithm for annotating source or intermediate code to either check the validity of pointer arithmetic in the source, or to guarantee that under minimal, clearly defined assumptions about the compiler, the optimizer cannot ‘‘disguise’’ pointers. We discuss an implementation based on a preprocessor for the GNU C compiler (gcc), and give some measurements of program slowdown.

 This originally appeared in *Proceedings of the ACM SIGPLAN ’96 Conference on Programming Language Design and Implementation SIGPLAN Notices ??*, ?, pp. ??-??.

Garbage-Collector-Safety

Automatic garbage collection can significantly simplify program development. It can also help to isolate program errors to one module by helping to ensure that no module can invalidate a data structure maintained by another. Though it is hard to quantify this benefit, most expert guesses seem to place it in the range of 30-40% of program development time for programs that manipulate complex linked data structures (cf. [Rovner85]). Furthermore, in our experience, garbage collected programs tend to be based on higher level abstractions where appropriate, and thus tend to have fewer arbitrary restrictions on functionality (*e.g.* input size limitations).

Tracing garbage collectors identify all accessible memory by starting at program pointer variables, and traversing all pointers through the heap. *Conservative* garbage collectors (cf. [Bartlett88], [BoehmWeiser88], [Rovner85], [Boehm 95]) can do so even in the presence of incomplete information about pointer identity by treating any bit pattern that might represent the address of a heap object as a pointer. This may result in some extra memory retention, but this is rarely significant [Boehm93]. This approach enables garbage collectors to operate easily with conventional programming languages such as C and C++, and with minimal or no modification to existing compilers. It has been used by many language implementations that use C as an intermediate code (cf. [Bartlett89], [AtkinsonEtAl89], [Omohundro91], [RoseMuller92], [SchelterBallantyne88]) , and it facilitates interoperation between C and higher level programming languages.

Although conservative garbage collectors require minimal cooperation from the compiler, they do require some to guarantee correct operation. For example, a conventional C compiler may replace a final reference `p[i-1000]` to the heap character pointer p by the sequence:

```
p = p - 1000;
... p[i]...
```

If a garbage collection is triggered between the replacement of `p`, and the reference to `p[i]`, there may be no recognizable pointer to the object referenced by `p`. Thus such code is not *GC-safe*. (Here we assume the existence of multiple threads of control. Similar examples can be constructed if the collector can only be invoked at function call sites.)

Note that the fundamental problem is not the way in which `p[i-1000]` is computed, but the fact that the original value of p is overwritten before the computation is complete. Thus the problem is to convince the compiler to preserve some values longer than they appear to be needed, rather than to suppress specific optimizations.

Similar problems may occur as a result of induction variable optimizations, or in the construction of a large constant address displacement on a machine that provides only a small signed displacement field in machine instructions. Other examples can be found in [BoehmChase92].

Such problems are in fact extremely rare with existing compilers. Conservative garbage collection is commonly used with conventional unmodified optimizing compilers, and to our knowledge, the above problems have *only* been observed in examples contrived for the purpose. This is in strong contrast to conventional optimization bugs in production compilers.

Nonetheless we would like to be able to generate GC-safe code to guarantee safety of the approach. We would at least like to be able to point to a practical alternative in case such a problem is discovered. And we would like to defend against future clever optimizations that may increase the frequency of such problems.

For most compilers, it is possible to guarantee GC-safety by generating fully debuggable code. If the values of all logically visible variables are explicitly stored for debugging purposes at all program points, then they will also be available for the garbage collector. Unfortunately, performance considerations make such an approach (and some other solutions, *e.g.* involving frequent uses of `volatile` variables) unattractive.

Unfortunately, GC-safety is likely to introduce some runtime overhead, distinct from time spent in allocation and garbage collection routines. (See, for example, [OTooleNettles94], [DetlefsDosserZorn93], and [BoehmDemersShenker91] for some measurements of conventional collector overhead.) Since the problem is essentially never observed in practice, there is some argument that the introduced overhead should be very small with ‘‘sufficiently good’’ program analysis. In particular all existing programs can theoretically be compiled on existing compilers as they are now. Here we look at the performance of a rather simple, but decidedly imperfect analysis.

Source Checking

A related problem is that of checking that the original source code is safe for use with a garbage collector.

Recall that the C language allows some arithmetic on pointers. However a value may be added to a pointer only if the result and the original pointer are addresses within the same object. Either may also point one past the end of the object, which we handle by allocating all heap objects with at least one extra byte at the end. We make some additional assumptions about the input program:

1) No integers are converted to heap pointers. In fact, conversion of a pointer to an integer and back, without intervening arithmetic, is benign, as is the common practice of converting very small integers to pointers that are never dereferenced. Disguised pointer arithmetic is not. Hashing on pointer values is no problem, since we effectively assume a nonmoving garbage collector. Our preprocessor issues warnings when nonpointer values are directly converted to pointers. It could and should also issue warnings when the same thing is accomplished by a cast between different structure pointer types or the like.

2) Pointers are not hidden from the garbage collector by writing them to files and reading them back in, or by writing them to collector invisible (or misaligned) memory locations. To our knowledge, this is possible in a strictly conforming ANSI C program only via pointer input with either a `scanf` variant and `%p` format or with `fread` into a pointer-containing type, or with a call to `memcpy` or `memmove` with arguments whose types don’t match. Thus this should be easily checkable, though we currently don’t do so.

Thus the practical issue in checking the safety of a program is to ensure that the ANSI C requirements on pointer arithmetic are satisfied. Our main goals are to ensure that no objects are prematurely collected, and that garbage collector data structures are unlikely to be overwritten. We restrict our attention to heap pointers, since that both ensures the first goal, and makes the overwriting of collector data structures much less likely.

Our garbage collector provides a facility for checking whether two pointers reference the same heap object. Hence it suffices to check after each pointer arithmetic operation that the result still points to the original object. We argue below that the program can be annotated with such checking calls in exactly the same manner as annotating it for GC-safety. In fact, the checking calls ensure GC-safety, though not in a performance-optimal fashion.

Admittedly the annotated program will incur a significant performance loss, at least without substantially more analysis than we perform. We expect such checking to be performed only during debugging, by analogy to the current use of systems like Purify [HastingsJoyce92].

Our checking is not completely accurate, since the garbage collector rounds up object sizes. But it is sufficient to ensure that on a machine with typical RISC alignment restrictions at most unused memory can be accidentally referenced through an incorrectly computed heap pointer to a primitive type. It is currently still possible to reference or overwrite other memory if C structures are accessed as a whole, e.g. if they are passed as parameters or assigned to other structures. This could be remedied at minimal cost with the insertion of an additional check.

Related Work

We extend, refine and implement the work presented in [BoehmChase92] and [EllisDetlefs93].

Unlike [BoehmChase92] we start with the assumption that the garbage collector recognizes all pointers to the interior of an object, not just to the first byte of the object. Recent experience suggests that this is the right framework, particularly for typical C++ implementations which implicitly generate pointers to the interior of an object. The techniques of [Boehm93] can greatly reduce the danger of space leakage that we previously associated with this approach. This new assumption greatly simplifies our task. Unfortunately, it also usually invalidates assumption (A) of [BoehmChase92], so our correctness argument has to be different.

An approach even more similar to ours here is presented in [EllisDetlefs93], but not in a great amount of detail.

There has been much work on the generation and representation of object and stack layout information by the compiler for the garbage collector. Recent examples include [DiwanMossHudson92] [Goldberg91], and [Fradet94]. The idea is to generate static information (either tables or traversal functions) to communicate pointer locations to the garbage collector. Thus these techniques also introduce minimal overhead into client code, while producing much more precise information for the garbage collector. (The overhead is often very similar to that described here, though the details will vary depending on the expressiveness of the layout representation scheme.) The disadvantages of such an approach are that it is hard to accommodate code written in languages such as C or often code compiled by more than one compiler, table size and/or interpretation overhead may be a problem, and typically garbage collections must be restricted to only be triggered at specific interruption points. The last point may introduce other forms of overhead in some multithreaded environments.

To our knowledge, none of this work addresses the issue of run-time overhead required to keep pointers accessible, though [DiwanMossHudson92] does discuss the necessity for such overhead.

There have also been many proposals for completely source-level implementation of garbage-collection using either the C++ constructor/destructor mechanism or Ada 9X ‘‘finalization’’. These typically greatly increase the number of memory references necessary for pointer assignments or pointer variable creation. Hence they are not performance competitive with conservative collectors, though the expense may be unavoidable in hard real-time environments [Edelson91].

There has been a substantial amount of work on safer C implementations (cf. [HastingsJoyce92], [AustinBreachSohi94], [JonesKelly95].) Most of these have concentrated on detecting erroneous memory accesses, not erroneous pointer or subscript arithmetic. Though the two are related, the distinction is important. A common bug (sometimes referred to incorrectly as a ‘‘technique’’) in C code is to represent an array as a pointer to one element before the beginning of the array’s memory. This fails in a garbage collected system. It may also result in incorrect pointer comparisons on segmented memory machines.

Like Purify, but unlike [AustinBreachSohi94], our checked code uses the same data structure layout as unchecked code. Hence it is possible, indeed trivial, to use checked code with third party object-code-only libraries, or with faster, unchecked modules.

Our checked code is very similar in spirit to recent independent work by Jones and Kelly [JonesKelly95]. The major differences are that we do not check references to statically allocated and stack memory, we use the garbage collector’s data structures to determine whether two pointers reference the same object, and we take a different approach to inserting the checking code. The garbage-collector-based check is probably somewhat more efficient, since it relies primarily on mapping any address to the beginning of the corresponding object, an operation crucial to the collector’s performance. (Their fundamental data structure is a splay tree of objects, we use a tree of fixed height 2 describing pages of uniformly sized objects.) Hence both the allocator and collector are tuned to make such lookups very fast. Since we use existing collector data structures, the checking can be implemented purely in a preprocessor. Our approach to inserting checking code is significantly different, in that we essentially treat pointer offset calculations as pointer arithmetic. This appears to result in better checking of structure accesses.

Compiler Safety Problem Statement

We are interested in compiling ANSI C [ANSI89], minimally restricted as defined above, such that the object code resulting from a strictly conforming program is guaranteed to operate correctly with a conservative garbage collector, if we replace every call to the *malloc*, *calloc* and *realloc* functions by corresponding calls to a collecting allocator, and remove all calls to *free*.

We refer to the machine stack, registers, and statically allocated memory as *GC-roots*. We assume that the garbage collector preserves every object that is reachable by following pointers from a GC-root, and possibly through heap resident pointers. We assume that the garbage collector recognizes any address corresponding to some place inside a heap allocated object as a valid pointer. ([Boehm95] satisfies these assumptions in its default configuration.)

It follows from our restrictions on C programs that every heap object that may be accessed is accessible by following a chain of pointers from program variables. Some of these pointers may point to the interior of the object they reference. We would like to guarantee that the generated object code satisfies the same property at every program point. That is, there must be a path from the GC-roots, possibly through other heap objects, to every potentially accessible heap object. In particular, either program variables or equivalent compiler temporaries, should be explicitly stored, possibly in a machine register, as long as heap objects they refer to may be accessed. This suffices to ensure the correctness of a suitable conservative garbage collector.

Returning to our original example of compiling the expression `p[i-1000]`, our goal is to ensure that either the pointer `p`, or a pointer to someplace inside the array, is kept visible to the collector during the evaluation of the expression, even if `p` is dead after the expression is evaluated. Note that this does not necessarily inhibit any traditional optimizations. It may require another register to preserve the original value of p, and thus conceivably add register spill code. (On machines with only two operand instructions, it may also directly add a small amount of additional code.)

Our goal is to ensure that this property is satisfied with minimal effect on the quality of compiler generated code, and in such a way that it can be retrofitted to existing programming language implementations. We will describe our algorithm for ensuring garbage-collector-safety primarily as a source-to-source transformation on C code. We use a source-to-source transformation both because it provides a convenient vehicle for explanation, and because in the short term it appears to be an interesting implementation strategy. It makes it possible to provide for the GC-safety of some compilers without altering the compiler at all. Since GC-safety is usually not an issue for unoptimized code, we expect that our prepass will generally be used only in conjunction with the optimizer, and hence the time required to run the prepass is less of an issue in this context.

We make the following assumptions about the target compiler, in addition to the expected correctness assumptions:

0) Every allocation call in the source results in a corresponding call to an allocation function in the object code. Every store or load to/from the heap in the object code corresponds to an access of or assignment to the corresponding object in the source. (There may be fewer loads and stores in the object code.)

1) It must be possible to define a macro KEEP_LIVE(*e*,*y*), with the following semantics: KEEP_LIVE(*e*,*y*) evaluates to the value of the pointer expression *e*, but with the added constraint that the value of the pointer variable *y* must be visible to the garbage collector (i.e. treated as live) until the expression is completely evaluated, that is until the value of *e* is visible to the garbage collector. We will augment the source program with KEEP_LIVE expressions to ensure that relevant pointers are kept visible by the compiler.

The use of the word ‘‘macro’’ here should not be take too literally. The expansion may depend on the types of the expressions and involve additional temporary variables. Since we can generate the expansion explicitly, it does not need to be expressible as a C macro.

2) The value of a KEEP_LIVE expression must be treated as opaque, in the following sense: The value must be explicitly and continuously stored in either a GC-root or the appropriate heap object(s), provided the value is used directly (without intervening pointer arithmetic) in a dereference operation, or used as the second argument of another KEEP_LIVE expression. It must be visible to the collector at all points between the evaluation of the original KEEP_LIVE and the final use. Thus, if we have `p =` KEEP_LIVE(...)`; q = p;` ...`*q`..., then the result of the KEEP_LIVE expression must be explicitly stored until `*q` is retrieved. However, the same is not required for `p =` KEEP_LIVE(...)`; q = p + 4;` ...`*q`..., since the value of `p` is not dereferenced directly, and the pointer addition might conceivably be subject to optimization. We can force either `p` or `q` to be explicitly visible at all times by writing: `p =` KEEP_LIVE(...)`; q =` KEEP_LIVE(`p + 4, p`); ...`*q`...,

Informally, KEEP_LIVE has two effects; it causes *y* to be kept live during the evaluation of *e*, and it causes the compiler to lose all information about how the resulting value was computed, thus preventing it from discarding the value and subsequently recomputing it.

One way to implement KEEP_LIVE(*e*,*y*) is as a call to an external function whose implementation is unavailable to the compiler for analysis, but which actually just returns its first argument. In all environments of which we are aware, this will force the value to be stored explicitly (perhaps in a register). The value will continue to be explicitly available through a dereference or another KEEP_LIVE in the presence of all standard compiler optimizations.

This implementation of KEEP_LIVE is, of course, terribly inefficient. More efficient implementations are suggested in [BoehmChase92], and one is described below.

Our problem then is to annotate the original expression by replacing a number of expressions *e* with KEEP_LIVE(*e*,*y*) in such a way that the above rules guarantee that every heap object is accessible via an interior pointer chain from the time of its allocation until its last access.

We will assume that allocation functions return a result that is (treated as) the value of a KEEP_LIVE expression.

In order to simplify the presentation, we will assume that the following kinds of expressions either return nonpointers or occur as the right side of an assignment to a local variable that is not assigned elsewhere in the same expression. In effect, we assume that temporaries have already been introduced, so that we can name the results of these subexpressions:

1) Pointer dereferences.

2) Function calls.

3) Conditional expressions.

We will refer to these as *generating* expressions.

Note that the introduction of the appropriate temporaries at source level is slightly more complex than one might like, but it is possible.

We will also assume that the only pointer dereferences are in expressions of the form **e*, and dereferences occur as late as possible with an explicit * operator. The `[]` and `->` operators occur only inside an & operator. Thus the expressions e_1`[`e_2`]`.x, `(*`e`)`.$x.z$, and e `->` x have been replaced by `*&(`e_1`[`e_2`]`.x`)`, `*&(`e `->` $x.z$`)` and `*&(`e `->` x`)` respectively. We assume expressions of the form &*e have been simplified to e.

Again, for the purposes of the presentation, we will ignore some compexities that must be handled by a source level implementation. For example, we ignore the fact that the C expression e `->` x will not actually involve a dereference if the field x has array type.

An Algorithm

We inductively define BASE(e), for pointer valued expressions e, to be the pointer variable from which the value of e is computed, or NIL if there is no such pointer variable; that is we define BASE(e) such that e and BASE(e) are guaranteed to point to the same object whenever e points to a heap object.

This is somewhat complicated by the presence of the & (address of) operator. Thus we simultaneously define BASEADDR(e) to be the possible base pointer for &e.

BASE(0) = NIL
BASE(x) = x
    if x is a variable and possible *heap* pointer
BASE(x `=` e) = x
    if x is a pointer variable
BASE(x `=` e) = BASE(e)
    if x is not a pointer variable
BASE(e_1 `+=` e_2) = BASE(e_1)
BASE(e_1 `-=` e_2) = BASE(e_1)
BASE(e_1 `++`) = BASE(`++` e_1) = BASE(e_1)
BASE(e_1 `--`) = BASE(`--` e_1) = BASE(e_1)
BASE(e_1 `+` e_2) = BASE(e_1)
    where e_1 is the expression with pointer type
BASE(e_1 `-` e_2) = BASE(e_1)
BASE(e_1`,` e_2) = BASE(e_2)
BASE(&e_1) = BASEADDR(e_1)

BASEADDR(x) = NIL
    if x is a variable
BASEADDR(e_1`[`e_2`]`) = BASE(e_1)
    if BASE(e_1) is not NIL
BASEADDR(e_1`[`e_2`]`) = BASE(e_2)
    if BASE(e_1) is NIL
BASEADDR(e_1 `->` x) = BASE(e_1)

Note that BASE is not defined for generating expressions. Generating expressions as subexpressions need not otherwise be considered, since they are assumed to fall into the first assignment case. BASE is also not defined for expressions with `[]` or `->` as the outermost operator, since they always occur inside &, and hence only BASEADDR is relevant.

BASEADDR is again not defined for generating expressions, since they are not l-values, and thus their address may not be taken. (Pointer dereferences are l-values, but have been transformed so they do not occur inside an '&' operator.) Similarly BASEADDR is not defined for other expressions that are not l-values, such as expressions with & as the outermost operator.

Our algorithm is now simple to state: replace every pointer-valued expression e that occurs as the right side of an assignment, or as the argument of a dereferencing operation, or as a function argument or result, by the expression KEEP_LIVE(e, BASE(e)). C increment and decrement operators are treated as assignments.

Correctness

This argument is of necessity informal, since a formal argument would need to be based on formal semantics of both C and at least some aspects of the target machine language. But we claim that it's sufficiently precise that given both of these it would not be hard to formalize.

Define an object to be *source-reachable* if a pointer to the object can be produced by a sequence of pointer dereferences, and (legal ANSI C) pointer addition operations starting from a program variable. We claim that there is no way to access a previously allocated object in a strictly conforming ANSI C program meeting our restrictions if it is not source-reachable.

We define an object to be *GC-reachable* if it can be reached by following a chain of addresses starting with one stored in a GC-root, and such that every subsequent address is stored in the heap object referenced by the preceding one.

Observe that any pointer value which according to the source semantics should be stored in a program variable or in the heap is explicitly stored either in the GC-roots or in the corresponding heap object until its final access. This follows from the fact that all pointer values are generated by KEEP_LIVE expressions.

Assume that at some point before the final access to object P, it becomes GC-unreachable. Consider the path to P along which the last access takes place. Consider the first object Q in this path to become GC-unreachable. Since the last access to Q has not yet taken place, according to the preceding observation, a pointer to Q must be source-reachable. Hence it must have been generated by a KEEP_LIVE expression, and the result of this KEEP_LIVE expression will be subsequently referenced. Hence it should be explicitly stored in a GC-root or in the preceding heap object along the chain. Thus we obtain a contradiction.

Hence objects remain GC-accessible until the final access. Since all accesses in the object code correspond to accesses in the source, it follows that the object code cannot access collected objects.

Optimizations

The above algorithm is somewhat deficient in several respects:

1. It inserts many unnecessary KEEP_LIVE calls. There is clearly no reason to replace the assignment `p = q` by `p = KEEP_LIVE(q, q)`.
   This is primarily a problem of compilation speed and compactness of the intermediate representation. It can be easily avoided by keeping track of whether or not an expression result is statically known to be simply a copy of a value logically stored elsewhere. If it is, then condition (2) from the definition of KEEP_LIVE guarantees that there is no need to add the KEEP_LIVE.

2. Certain C expressions are difficult to transform at source level. In general, a pointer expression *e*++ should be transformed to `(tmp1 = &(`*e*`), tmp2 = *tmp1, *tmp1 = tmp2 + 1, tmp2)` before inserting KEEP_LIVE calls. But this should be optimized to `(tmp = (`*e*`), (`*e*`) = tmp + 1, tmp)` if *e* is a simple variable that might be register allocated, to avoid forcing *e* to memory. This problem is very likely to disappear if the transformation is made on intermediate code.
3. The choice of base pointer variables may significantly impact the optimizations that can still be performed by the compiler. Consider the canonical string copying loop in C:

```
p = s; q = t;
while (*p++ = *q++);
```

   After the above optimizations, we would transform the loop to:

```
while(  *(tmpa = p,
          p = KEEP_LIVE(tmpa+1,
                        tmpa),
          tmpa)
        = *(tmpb = q,
            q = KEEP_LIVE(tmpb +1,
                          tmpb),
            tmpb));
```

   This is correct, and really specifies the same operations as the original, though less concisely. But it forces the values of `p` and `q` to explicitly appear in a register. This prevents the C optimizer from translating the pointer arithmetic back to indexed loads based on `s` and `t`, which is profitable on some machines that allow a free addition in the load instruction (*e.g.* SPARC).
   A good heuristic appears to be to replace base pointers in KEEP_LIVE expressions by equivalent, but less rapidly varying base pointers, especially if those are likely to be live in any case. With a small amount of analysis we can generate the following less constraining code instead:

```
while(  *(tmpa = p,
          p = KEEP_LIVE(tmpa+1,
                        s),
          tmpa)
        = *(tmpb = q,
            q = KEEP_LIVE(tmpb +1,
                          t),
            tmpb));
```

4. So far, all transformations are safe in a multi-threaded environment, with an asynchronously triggered collector. If we know that garbage collections can be triggered only at procedure calls, the number of KEEP_LIVE invocations could often be reduced dramatically.

Debugging Applications

The above annotation scheme inserts a KEEP_LIVE call around every pointer arithmetic expression. In order to check that a pointer never leaves the object to which it points, it suffices to ensure that the expression (the first argument to KEEP_LIVE) always points to the same object as the base pointer (the second argument to KEEP_LIVE). (If we use a single KEEP_LIVE call around more than one arithmetic operation, the intermediate results may not be valid. But the equivalent program with KEEP_LIVE calls will still be safe in the presence of a collector.)

Thus we can convert our GC-safety preprocessor to a pointer arithmetic checker by simply replacing the KEEP_LIVE call with a function call that does the appropriate checking. For example, assuming `p` is a character pointer, the expression `p + 1` will be transformed by the debugging mode preprocessor to

```
(char (*)) GC_same_obj((void *)((p+1)),
                       (void *)(p)))
```

Here `GC_same_obj` is a real function which takes the place of KEEP_LIVE. It checks that both arguments point to the same object, and the returns the first argument. Since the definition of this function is not available to the compiler, The call to `GC_same_obj` will simultaneously have the intended effect of the KEEP_LIVE call.

Our pointer arithmetic checking is not dependent on the exact type of a pointer. If we cast a ‘‘`struct A *`’’ to ‘‘`struct B *`’’, accesses to fields of the resulting pointer will be checked to verify that they are within the allocated object. (As mentioned above, the only possible exception at this stage is an access to an entire substructure of `B` that is only partially within the allocated object. That would need additional checking code.)

An Implementation

We have built a GC-safe compiler for ANSI C (plus some GNU extensions) by writing a C-to-C preprocessor that annotates the input program as described in the previous sections. The output for GC-safety is initially specific to gcc, i.e. the resulting code is safe only when compiled by gcc. Gcc dependencies are highly localized, so it should be possible to accommodate other compilers in the future. It should be possible to make the output in source-code-checking mode usable with any ANSI C compiler.

The preprocessor could conceivably be used directly with a C++ implementation that first translates to C. Or a similar strategy could be applied at the intermediate code level inside a C++ compiler.

We implement the ''`KEEP_LIVE`'' primitive by taking advantage of gcc's flexible syntax for inline assembly code. Assembly code can reference the value of a C expression, which may be requested to be available in a register or memory location. Thus `KEEP_LIVE` generates an empty assembly code sequence, depending on both arguments. It requests that the first argument be assigned the same location as the result. See below for details.

Only optimizations (1) and (2) from above are implemented. However we do expand certain C constructs, particularly increment and decrement operations, in more specialized ways than suggested above. For example, if `p` is a character pointer, then in debugging mode the expression `++p` is expanded to

```
((char (*))
    GC_pre_incr(&(p),
                sizeof(char)*(+(1))))
```

Here `GC_pre_incr` is a function performing the equivalent of a pre-increment operation which also checks that the result points to the original object.

It is highly desirable to run this preprocessor between the normal C preprocessor (macro-expander) and the C compiler. In this way arbitrary macros are handled correctly and the preprocessor is not normally visible during debugging. Hence `KEEP_LIVE` is not generated as a macro call; instead its expansion is generated directly.

Our preprocessor maintains a copy of the input file (including the source line information generated by the C preprocessor). It parses and partially type-checks the source. In the process it generates a list of insertions and deletions, sorted by character position in the original source string. After parsing is complete, the insertions and deletions are applied to the original source. The yacc/bison grammar and scanner were derived from their gcc equivalents.

We do not actually transform dereference operators as described above. Instead we defer generating `KEEP_LIVE`s until enough of the context has been seen to determine the correct transformation. This again introduces complexity which is solely the result of the source level implementation, and wouldn't be necessary if the transformation were done at a lower level.

We have not attempted to tune the performance of the preprocessor to reduce compile time. But for our purposes that hasn't been a significant issue. (In fact we have yet to compile the preprocessor with optimization enabled or assertion checking disabled.) It should be much faster than the rest of the compilation process, and certainly is no slower.

Performance

We measured a small collection of small-to-medium-sized C programs, mostly drawn from the Zorn benchmark suite[DetlefsDosserZorn93]. All of these programs are very pointer and allocation intensive.

Standard C libraries were not preprocessed. This is probably not unrealistic since the critical pieces are likely to be either hand assembly coded, or manually checked for GC-safety or, failing that, thoroughly tested for GC-safety of the normally optimized version.

The programs measured were:

cordtest: 5 Iterations of the test normally distributed with our ''cord'' string package. This was run with our garbage collector. The string package and the test program were processed. No part of the garbage collector itself was. We uncovered and fixed one benign pointer arithmetic bug in the measurement process. (2100 lines, excluding the collector)

cfrac: A factoring program. The smallest member (6000 lines) of Ben Zorn's benchmark collection. It was run with the second largest input supplied by Zorn, and linked with the default malloc/free implementation. Hence pointer arithmetic checking was not operational. (The numbers for unoptimized program execution are not included, since the program makes use of explicit function inlining in a way that does not appear to be immediately compatible with unoptimized compilation by gcc 2.5.8.)

gawk: Version 2.11 of the GNU awk interpreter. This is the second smallest member of the Zorn benchmark suite (8500 lines). It was linked with the default malloc/free implementation and run with the second largest input supplied by Zorn. (We also ran this linked against our garbage collector in an attempt to get a another data point for the cost of pointer arithmetic checking. It ran correctly without checking. With checking enabled, it immediately and correctly detected a pointer arithmetic error which was also an array access error. After fixing that and uncovering two more abuses of pointer arithmetic we gave up. Some of these problems would have been avoided with a more recent version of gawk. It did however serve to test the pointer arithmetic checking code.)

gs: Ghostscript, as distributed with the Zorn benchmark suite (29500 lines). This was linked to use our garbage collector. (In the SPARCstation 2 tests, only the version with pointer arithmetic checking used the garbage collector.) The Ghostscript custom allocator was disabled. It was run with the second largest supplied input file. No pointer arithmetic errors were found. This is probably due to a combination of an unusually clean coding style and the fact that most heap objects have prepended standard headers. Thus a pointer to one before the body of the object would not be discovered. It also could not confuse the garbage collector.

All programs were compiled with gcc 2.5.8 and timed on a Weitek-processor SPARCstation 2 running SunOS 4.1.4, a SPARCStation 10 running Solaris 2.5, and a Pentium 90 running Linux 1.81. (The SPARCStation 2 tests were run with a slightly older version of the preprocessor, but should be comparable.) We give slowdown percentages relative to the unpreprocessed optimized version for the same code preprocessed for GC-safety, the fully debuggable (and hence probably guaranteed safe) code, and debuggable code preprocessed to insert pointer arithmetic checks:

SPARCstation 2:

	-O, safe	*-g*	*-g, checked*
cordtest	9%	54%	514%
cfrac	17%	<needs modifications due to inlining>	
gawk	8%	25%	<fails>
gs	0%	33%	205%

SPARC 10:

	-O2, safe	*-g*	*-g, checked*
cordtest	9%	56%	529%
cfrac	8%	-	-
gawk	8%	48%	-
gs	5%	37%	366%

Pentium 90:

	-O2, safe	*-g*	*-g, checked*
cordtest	12%	28%	510%
cfrac	11%	-	-
gawk	9%	41%	-
gs	6%	17%	279%

Early attempts at measurement suggest that all such timing results are somewhat suspect due to cache effects. At one point, while measuring a dynamically linked executable on a SPARCStation 10, we saw a consistent factor of 2 difference between two copies of the same executable. We subsequently attempted to minimize such effects. For example, the SPARCStation 10 and Pentium 90 numbers refer to an average of several runs of several copies of the executable, with the highest running time discarded. We avoided dynamic libraries on the SPARCStation 10, which seemed to add significant variation. The resulting average execution times appear to be reproducible to within 1 or 2%, though individual execution times still occasionally varied by more than 10%. The measurements remained more or less constant through the last several rounds of bug fixes to the preprocessor.

To obtain a more robust, though perhaps less relevant measure, we also measured SPARC object code expansions with and without preprocessing. These numbers include only the code that was actually processed, not the standard libraries:

	-O2, safe	*-g*	*-g, checked*
cordtest	9%	69%	130%
cfrac	6%	-	-
gawk	15%	68%	-
gs	19%	73%	160%

Note that the first two columns could be expected to be somewhat indicative of execution times outside of libraries. The last column, on the other hand, grossly understates dynamic instruction counts, since additional procedure calls are introduced.

Analysis

To understand the reasons for the performance cost, it is instructive to look at a very simple C function:

```
char f(char *x)
{
    return(x[1]);
}
```

The body is translated by our preprocessor to the following code, which is not normally intended for human consumption:

```
return((*({ typeof(char ) * __result;;
            asm(" ": "=r" (__result) :
                "0" (&(x[1])),
                "rfmi" (x));
            __result;})));
```

This uses a gcc specific extension to introduce an expression local variable `__result`, and then inserts an empty assembly instruction with the constraint that the address `&(x[1])` must occupy the same location as the output operand `__result`. The assembly instruction has an unused second argument `x`, which may be stored anywhere. Finally `__result` is dereferenced.

The SPARC code generated by gcc -O2 for the return expression is:

```
add %o0,1,%g2
! empty assembly instruction here
ldsb [%g2],%o0
```

In contrast, the normal optimized code is simply

```
ldsb [%o0+1],%o0
```

Note that both versions are perfectly safe in the presence of a garbage collector. The problem is that the empty assembly instruction introduced an explicit program point at which the pointer addition must have been completed, but the load instruction cannot have been completed, since the compiler views it as depending on the result of the assembly instruction. Hence there is no way to take advantage of the index arithmetic in the load instruction. Similar problems occur with pure pointer arithmetic. Together these account for a majority of the overhead.

This observation is consistent with the measurements from the preceding section. If the overhead were primarily due to additional register pressure and hence register spills, one would have expected much more substantial performance degradation on the Intel Pentium machine, which has substantially fewer registers than the SPARC-based machines.

Thus it is safe to assume that most of the slowdown is caused by our somewhat naive implementation, and is not an inherent cost of garbage-collector-safety. The next section explores the possibility of eliminating much of this spurious overhead, and thus getting better bounds on the unavoidable overhead.

A Postprocessor

The above suggests that much of the decrease in object code performance could be eliminated with some peephole optimizations. To test this hypothesis, we built a simple peephole optimizer that operates on the SPARC assembly code level. (The code was derived from a simple SPARC 1/2 instruction scheduler [Boehm94].) It first performs a simple global, intraprocedural analysis that allows us to identify possible uses of register values. It subsequently looks for one of the following three patterns inside each basic block and transforms them appropriately:

1)

```
add  x,y,z  ==>   ...
...               ld [x+y]
ld    [z], ...
```

2)

```
mov  x,z    ==>   ...
...               ...x...
...z...
```

3)

```
add  x,y,z  ==>   ...
...               add x,y,w
mov  z,w
```

Not all of the safety constraints are listed here. An important one is that the register `z` should have no other uses. For this purpose, we arranged for the KEEP_LIVE expansion to introduce a use of the second argument right after the evaluation of the first argument. (It generated a special comment understood by the peephole optimizer.) The arguments that these preserve GC safety are as follows:

1) If the other safety constraints for this transformation are obeyed, then `x` and `y` remain where they were originally live. The transformation could not apply if `z` were originally mentioned as the second argument of a KEEP_LIVE. All other values remain live in the same ranges as before. Hence we cannot invalidate KEEP_LIVE semantics.
2) The same values remain live at all program points, assuming the already necessary safety constraint that `x` is not overridden.
3) The same argument as (1).

We do not reassign registers or reschedule the resulting code.

On a SPARC 10, the execution time and code size degradations from the fully optimized normally compiled code were reduced to:

	running time	*code size*
cordtest	4%	3%
cfrac	2%	3%
gawk	1%	7%
gs	2%	7%

Based on manual inspection of the remaining code, it appears that this is still significantly worse than what could be done with a more precise analysis. It appeared that many of the remaining source of overhead were still basically of the above form, but had been transformed sufficiently by the optimizer that they were not as easily recognizable, often because that would have required more global analysis.

Extensions

It is possible to extend this approach to a collector which considers interior pointers as valid only if they originate from the stack or registers (another possible operating mode of our collector). This requires asserting that the client program stores only pointers to the base of an object in the heap or in statically allocated variables. It would again be possible to insert dynamic checks to verify this. This avoids some complications with allocating large objects as discussed in [Boehm93]. However it interacts suboptimally with C++ compilers that use interior pointers as part of their multiple inheritance implementation.

Acknowledgements

Some of this grew out of prior work with David Chase and extensive discussion with John Ellis.

Rhonda Reese made an earlier version of gcc GC-safe, based on a much earlier approach. That effort convinced me that a source-level approach was interesting, at least in the short-term.

Extensive news group discussions, primarily with Henry Baker, helped to persuade me to pursue this issue more aggressively.

The reviewers provided many useful comments. Much of this would not have been possible without the availability of the GNU C compiler.

References

[ANSI89] *Standard X3.159-1989, American National Standard for Information Systems - Programming Language - C,* American National Standards Institute, Inc.

[AtkinsonEtAl89] Atkinson, Russ, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser, ''Experiences Creating a Portable Cedar'', *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices 24*, 7 (July 1989), pp. 322-329.

[AustinBreachSohi94] Austin, Todd M., Scott E. Breach, and Gurindar S. Sohi, ''Efficient Detection of all Pointer and Array Access Errors'', *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, SIGPLAN Notices 29*, 6 (June 1994), pp. 290-301.

[Bartlett88] Bartlett, Joel F. ''Compacting garbage collection with ambiguous roots'', *Lisp Pointers 1*, 6 (April-June 1988), pp. 3-12.

[Bartlett89] Bartlett, Joel F., *Scheme --> C a Portable Scheme-to-C Compiler,* WRL Research Report 89/1, Digital Equipment Corporation Western Research Laboratory, January 1989.

[Boehm93] Boehm, Hans-J., ''Space Efficient Conservative Garbage Collection'', *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, SIGPLAN Notices 28*, 6 (June 1993), pp. 197-206.

[Boehm94] The SPARC scheduler is available from parcftp.xerox.com:pub/gc/sched.tar.Z. It also operates in a GC-safe mode, along the lines of [BoehmChase92].

[Boehm95] An overview of our conservative garbage collector along with the source code can be accessed from ftp://parcftp.xerox.com/pub/gc/gc.html.

[BoehmChase92] Boehm, Hans-J., and David Chase, A Proposal for GC-Safe C Compilation, *The Journal of C Language Translation 4,* 2 (December , 1992), pp. 126-141. Also available (with the publishers permission) from parcftp.xerox.com:pub/gc/boecha.ps.Z.

[BoehmDemersShenker91] Boehm, H., A. Demers, and S. Shenker,''Mostly Parallel Garbage Collection'', *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 26*, 6 (June 1991), pp. 157-164.

[BoehmWeiser88] Boehm, Hans-J. and Mark Weiser, ''Garbage collection in an uncooperative environment'', *Software Practice & Experience 18*, 9 (Sept. 1988), pp. 807-820.

[DiwanMossHudson92] Diwan, Amer, Eliot Moss, Richard Hudson, ''Compiler Support for Garbage Collection in a Statically Typed Language'', *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, SIGPLAN Notices 27*, 7 (July 1992), pp. 273-282.

[DetlefsDosserZorn93] Detlefs, David, Al Dosser, and Benjamin Zorn, ''Memory Allocation Costs in Large C and C++ Programs'', University of Colorado, Boulder Technical Report CU-CS-665-93. Available for ftp from cs.colorado.edu:pub/techreports/zorn/CU-CS-665-93.ps.Z.

[Edelson91] Edelson, Daniel, ''A Mark-and-Sweep Collector for C++'', *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992, pp. 51-58.

[EllisDetlefs93] Ellis, John R., and David L. Detlefs, "Safe Efficient Garbage Collection for C++'', Xerox PARC Technical Report CSL-93-4, September 1993. Also available from parcftp.xerox.com:pub/ellis/gc/gc.ps.

[Fradet94] Fradet, Pascal, ''Collecting More Garbage'', *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 24-33.

[Goldberg91] Goldberg, Benjamin, ''Tag-Free Garbage Collection for Strongly Typed Programming Languages'', *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 26*, 6 (June 1991), pp. 165-176.

[HastingsJoyce92] Hastings, Reed, and Bob Joyce, ''Fast Detection of Memory Leaks and Access Errors'', *Proceedings of the Winter '92 USENIX conference*, pp. 125-136.

[JonesKelly95] Jones, Richard, and Paul Kelly, ''Bounds Checking for C'', http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html.

[Omohundro91] Omohundro, Stephen M., *The Sather Language*, ICSI, Berkeley, 1991.

[OTooleNettles94] O'Toole, James, and Scott Nettles, ''Concurrent Replicating Garbage Collection'', *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 34-42.

[RoseMuller92] Rose, John R., and Hans Muller, ''Integrating the Scheme and C languages'', *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 247-259.

[Rovner85] Rovner, Paul, ''On Adding Garbage Collection and Runtime Types to a Strongly-Typed Statically Checked, Concurrent Language'', Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, CA, July 1985.

[SchelterBallantyne88] Schelter, W. F., and M. Ballantyne, ''Kyoto Common Lisp'', *AI Expert 3*, 3 (1988), pp. 75-77.