Abstract

This paper describes ways that storage allocation optimization, though "correct", can convert a running program into one that fails. A general "safety condition" is proposed and applied to some existing and proposed storage allocation optimizations. These are shown to be unsafe or not general. Application of the safety condition yields several classes of storage allocations that may safely be optimized to stack allocations. For one useful class of allocations optimization is shown to be NP-complete.

Safety considerations for storage allocation optimizations

David R. Chase
Olivetti Research Center
Menlo Park, California

1 Introduction

Optimizations that convert heap allocations into stack storage allocations have been described in several papers and used in several compilers [BGS82, KKR+86, MJ76, Sch75]. These optimizations are expected to save time because allocation and deallocation from a stack is usually faster than either allocation and explicit deallocation or allocation and garbage-collected reclamation[1]. Furthermore, stack allocation improves locality of reference and generates addresses that will not change and need not be known to the garbage collector. This can permit additional optimizations to take place.

Improved lifetime analysis [Rug87, Cha87, RM88] and the growing use of garbage collection in programming languages (for example, Cedar Mesa, Lisp, ML, Modula-2+, Russell, Smalltalk) make it likely that these optimizations will appear in compilers. Unfortunately, unrestricted application of storage allocation optimizations is unsafe; these optimizations can convert programs that run well into programs that fail. Existing strategies for applying storage allocation optimizations are either over-restrictive or unsafe. This paper describes a safety condition for storage allocation optimizations and presents a strategy that is both general and safe.

[1] This is clearly true for a non-compacting collector.

2 Run-time model

The storage allocation optimizations discussed here assume single-threaded execution with an area of memory managed by a garbage collector and an area of memory managed as a stack. The stack is not garbage collected; this means that allocation and deallocation from the stack are guaranteed to be fast and that addresses of objects allocated on the stack will not change, but this also means that using all of the memory available for storing the stack is a fatal error. It is also assumed that stack space and heap space may be traded for each other, or that stack space is very large. At certain points in the program "new" (unaliased) storage is obtained from the heap; I will call these points *allocation sites*.

In the absence of interprocedural optimizations [Mur84] activation records are removed from the stack upon procedure exit. Within a procedure activation the allocation of additional storage from the stack is permitted; this storage is implicitly freed when the procedure is no longer active. This model does not necessarily prevent the use of coroutines, concurrency, continuations, or closures, but it is necessary that "stack" allocation and deallocation be fast with respect to heap allocation and garbage collection in order to make these optimizations profitable. This is not always true, but it is true often enough that these optimizations are interesting.

In describing optimizations and strategies I will assume the existence of some interprocedural information, but I will try to limit this to general-purpose information (i.e., call graphs and information about assignment of parameters to global variables and function results).

3 Allocation optimizations

The use of storage allocation optimizations has been proposed by Schwartz [Sch75], Muchnick and Jones [MJ76, MJ81], Barth [Bar77], Ruggieri [Rug87], and

Chase [Cha87], and implemented in compilers for Lisp [BGS82, Ste77], Scheme [Ste78, KKR+86], and Russell [BD85].

In Lisp compilers the optimization is applied to function closures and to floating point numbers. To allow the first-class treatment of functions activation records must be (in the most general case) allocated on the heap. In most cases, however, this general treatment is not necessary and the cost of heap allocation can be avoided by allocating activation records on a stack. For example, if a function F does not lexically contain any function-generating (lambda) expressions then F's activation record may be stack-allocated.

An alternate implementation strategy heap-allocates cells for variables and places pointers to these cells within activation records. A closure for a function inheriting lexical names contains pointers to the objects bound to the names instead of a pointer to the activation record which binds the names. Using this technique activation records are always stack-allocated but variables bound in the record may need to be heap-allocated and creation of a closure can be more expensive. If a function F contains no function-generating expressions then the variables bound in F need not be heap-allocated. Figure 1 shows two possible implementations of the activation record for a function F returning a closure.

Lisp implementations often represent floating-point numbers as pointers to floating point numbers in machine representation[2]. Doing this for all floating point data results in slow floating point performance. The S-1 Common Lisp compiler avoids this problem by heap-allocating floating point numbers only when (1) they are returned as a result from a (user) function or (2) they are accessible from a `cons` cell. If a non-heap number appears only in arithmetic operations then no memory needs to be allocated at all; if it is passed as a parameter to a function[3] or used in a context where its tag bits might be examined then memory is stack-allocated for it.

For the SETL compiler Schwartz proposed a stack allocation strategy based on an interval partition of a program's control flow graph. An allocation site A within an interval I may be converted to a stack allocation if the lifetime of the storage does not escape I. Because intervals are single-entry and disjoint it is easy to record the stack height at interval entry and reset it on edges leaving the interval. Here the allocation optimization is applied to all types of storage allocation, not just numbers.

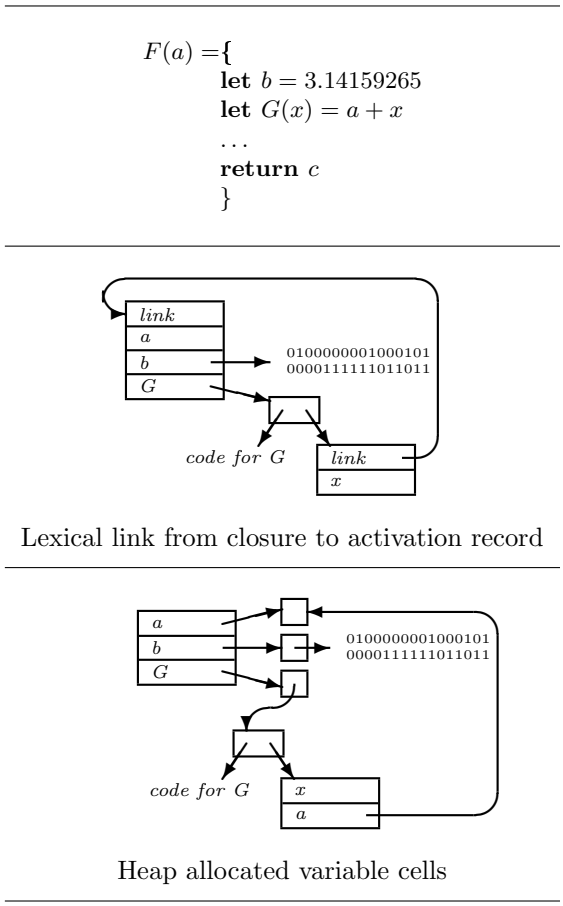

Figure 1: Implementations of a closure-returning function

Barth's technique does not shift allocation to the stack, but does remove reference counting operations and introduce explicit deallocation of storage. This is somewhat less efficient than stack allocation, though there is no reason that his work could not be adapted to replace heap allocations and deallocations with stack operations.

Ruggieri proposes a "local heap" for each activation record for storage whose lifetime is contained within the function activation (that is, the storage does not implement any part of any object assigned into a global variable or reference parameter or returned as a result). At procedure exit the associated heap is guaranteed to contain no live storage; this technique appears efficient if garbage collection of local heaps can be avoided.

[2]This is necessary when portions of each machine word on an untagged architecture are set aside for tag bits. Floating point formats typically use all the bits in a machine word, including those intended for use as tag bits by a Lisp implementation.

[3]A function storing one of its parameters into a `cons` cell or returning the parameter must ensure that it has been allocated from the heap.

4 Safety

The safety condition is just this: "optimization should not convert a program that runs robustly into one that does not." By "running robustly" I mean that the amount of memory available to the program is a good fraction larger than what the program actually needs; thus, slightly perturbing the storage needs of the program will not cause it to run out of memory. Note that a program running in the smallest possible amount of memory will need to collect garbage very frequently and thus run very slowly; thus, such programs are "not robust".

The stack allocation of numbers in Lisp[4] and the SETL interval allocation strategy are both unsafe. They both fail the test (in certain situations) by allocating on the stack many objects that become inactive[5] while live objects are still on top of the stack. If the optimizations are not applied then the shorter-lived objects are allocated from the heap where their storage can be reused if this is necessary.

A loop containing a short-lived allocation is sufficient to demonstrate this behavior. The loop below will iterate N times, and each iteration will allocate memory for a. If memory for a is stack-allocated then each iteration will increase the size of the stack. For large enough N the optimized program will fail, but a program allocating memory for a from a garbage-collected heap will only pause to collect garbage. Because (for this example) the storage allocated for a is not live from one iteration of the loop to the next it is clear that there will in fact be garbage to collect and that the program may continue after collection.

for $i = 1$ **to** N **do**
    $a \leftarrow$ **new**$(\ldots)$
    ...
    end

The Lisp allocation of numbers can also interfere with tail-call elimination. Performing the optimization can prevent the elimination of a tail-call if the caller is responsible for popping the number off of its stack, or it can increase the size of the stack frame reused by the called routine (or it can increase the size of the number stack, if a separate stack is used). Performing the stack-allocation optimization in either case removes one of the principal benefits of tail-call elimination: the ability to write a recursive function that runs in constant space.

[4] As described in the literature, though not in fact. In a loop stack allocations for floating point temporaries are invariant and can be hoisted out of the loop [Ste88].

[5] I will use the term "inactive" to mean "not reachable via a chain of pointers and containment from the root registers". Note that "inactive" is not a antonym for the flow analysis term "live".

Both Barth and Ruggieri's techniques are safe; Barth's approach does not use the stack and reclaims storage as soon as this is possible, and Ruggieri's approach both limits the stack space used and collects garbage within the stack, and falls backs to heap allocation if all else fails.

To safely convert a heap allocation into a stack allocation a compiler must ensure that (among other things) doing so will not lead to "excessive" storage waste. What is "excessive" depends upon the runtime environment. If stack and heap storage may be traded for each other, then it suffices to guarantee that the fraction of wasted stack storage never exceeds some bound; this is so because most garbage collection algorithms waste a certain fraction of the storage available. It is difficult for a compiler to place a bound on the fraction of storage wasted, however, because this requires both detailed knowledge of allocation sizes and knowledge about what storage *must* be active. In the example below allocations for data stored into the array a will remain live throughout the loop's execution but allocations for data stored into b will only last until the next assignment to b. If both allocations are performed on the stack then the percentage of the stack wasted in this loop depends on the relative sizes of the two allocations.

for $i = 1$ **to** N **do**
    $a[i] \leftarrow$ **new**$(\ldots)$
    $b \leftarrow$ **new**$(\ldots)$
    ...
    end

There are certain special cases where it is not difficult to bound the fraction of stack storage wasted; these will described later in the paper. A more conservative and tractable approach is to bound the amount (number of bytes) of wasted storage. This can be done using *may* information, and does not require detailed knowledge of allocation sizes. With appropriate assumptions it suffices to guarantee that a bound exists, if it is not known.

Note that in either case secondary waste may appear. That is, pointer-containing data allocated on the stack can prevent the reclamation of some heap storage. If the stack-allocated pointers are inactive, and their (heap-allocated) referents are otherwise inactive, then (many) garbage collectors will be unable to reclaim the referents even though they are heap-allocated. Secondary waste presents a problem because it can be very difficult to determine at compile time how much storage is actually wasted.

There are cases, however, where the compiler can determine that secondary waste is not possible. If it is possible to (safely) stack allocate all of the stack pointer referents, then there is nothing allocated from

the heap to waste. If the lifetime of a referent can be shown to be longer than the stack residence of a pointer, then there will be no secondary waste. This can happen when the referent is returned as (part of) a function result or assigned to (part of) a global variable or a reference parameter.

A better way to avoid this problem is to design the collector so that it only examines the fixed portions of each stack frame (that is, stack-allocated data is only examined if it is accessible from a variable or temporary result in some stack frame). It appears that this technique will be effective, so I will ignore the problem of secondary waste in the rest of the paper.

5 One safe strategy

This strategy requires certain simplifying assumptions:

1. each allocation is much smaller than the total amount of memory available;
2. non-recursive call chains are not especially deep;
3. the number of allocation sites in a procedure is not extremely large.

Taken together, these assumptions imply (vaguely) that the sum of the sizes of the allocations (counting only once per site) is not larger than the size of the stack. Another interpretation of this is "it's safe to compile Fortran". Given a call graph, a compiler can verify that assumptions two and three are correct. The first assumption, unfortunately, cannot always be verified at compile-time, but I will claim that it is reasonable. If short-lived, large allocations occur often, they cause frequent garbage collection, and in a non-compacting collector they can cause fragmentation problems. It is also possible, though not as desirable, to check the size of objects at allocation sites and place the large ones in the heap[6]. Given these assumptions, safety may be expressed in terms of number of objects allocated instead of number of bytes allocated.

For each allocation site in a program that may use the stack, the strategy ensures either that at most one instance of the allocation is ever stack-allocated, or that every instance stack-allocated (possibly excepting the last one) is active. This limits the number of wasted objects to at most one per allocation site, which is assumed to be an acceptable amount of waste.

[6]This is less desirable in a compacting collector because the possibility that the object's address may change can prevent some other optimizations.

5.1 Single-instance allocations

If the stack is popped on procedure exit, then many sites will never have more than one object allocated on the stack at a time. In non-recursive procedures (eligible) allocation sites that are not contained in any strongly connected region of the control flow graph may use the stack because the allocation can execute at most once before it is implicitly freed by exit from the procedure. In this example, storage for a may be stack-allocated because its allocation is not in a loop and f is not recursive (it is assumed that the data in a is not assigned to a global variable or to c; otherwise stack allocation would be incorrect).

$f(x) =$ {
let $a =$ **new...**
... no recursive calls
return c
}

The strategy also allows stack-allocation in portions of recursive procedures that cannot flow into a recursive call; allocation in this situation is clearly single-instance.

5.2 Multiple-instance allocations

Multiple-instance allocations include those in recursive procedures on paths to recursive calls and those occurring within loops. When almost all allocations performed at a site are guaranteed to be live within a procedure activation it is safe to transform the allocation into a stack allocation; such sites will be called *long* allocation sites. If a multiple-instance allocation can be transformed into a single-instance allocation, then (after that transformation) stack allocation is safe; such sites will be called *short* allocation sites. In either case, an allocation site must satisfy *all* safety constraints that apply in order to be stack-allocated.

5.2.1 ... in recursive procedures

Allocations that are (definitely) live *across* a recursive call may be safely stack-allocated. This is so because all instances of the allocation are active, except possibly for the one in the most recent procedure activation. In this example storage for a is live across the recursive call so it may safely be stack-allocated.

$f(x) =$ {
let $a =$ **new...**
$\ldots f(y) \ldots$
$\ldots a \ldots$
}

Allocations that are not live into a recursive call may also be transformed into single-instance alloca-

tions. The model for this transformation is the introduction of a non-recursive procedure replacing code leading up to the recursive call; allocations in the introduced procedure satisfying safety and correctness constraints may be stack-allocated. In practice this is accomplished by identifying allocations which are not live into the call and which are performed after all allocations that are live into the call. The stack pointer is saved, the allocations (and associated computations) occur, and the stack pointer is restored before the call.

It may not be safe to stack-allocate objects that are live into but not across a recursive call. With detailed interprocedural analysis it may be possible to discover that all such objects are live up to a point within a called procedure and that the stack is only popped from that point on, but such analysis is beyond the scope of this paper. Lacking such analysis, a compiler cannot stack-allocate storage that is live into but not across a recursive call because it can derive no bound on the number of wasted objects.

In this example d may safely be stack-allocated if it is freed (i.e., popped off the stack) before the recursive call to f; b may be safely stack-allocated because b is live across the call; c may not be safely stack-allocated because c is live into the call but not across it; and a may not be safely stack-allocated even though a's lifetime does not extend into the call because b must be freed before a can be freed and b is used after the call.

$$
\begin{aligned}
f(x,y) = \{ & \\
& \textbf{let } a = \textbf{new}\ldots \\
& \textbf{let } b = \textbf{new}\ldots \\
& \textbf{let } c = \textbf{new}\ldots \\
& \textbf{let } d = \textbf{new}\ldots \\
& \ldots f(b,c)\ldots \\
& \ldots b\ldots \\
& \}
\end{aligned}
$$

Recursion can interfere with other otherwise safe stack-allocation techniques; these problems will be pointed out where they occur below. Some compilers do stack allocate objects into recursive calls [BGS82], but only when the objects are known to be small (machine floating point numbers) when compared to an activation record.

5.2.2 ... within loops

I assume the following model for allocation optimization within loops. A *loop* is a single-entry strongly connected region with no embedded cycles that do not contain the root node (an interval, in a reducible flowgraph). When loops are nested the inner loop(s) will be treated as a single node in the outer loop. In optimization loop invariant computations and initializations for strength-reduced calculations are placed in a "landing pad" prepended to the loop entry [CLZ86, ACK81] and all edges into the loop from outside the loop are redirected to this landing pad. Storage allocation optimization will make use of this landing pad for some allocations and to record the stack height upon entry to the loop.

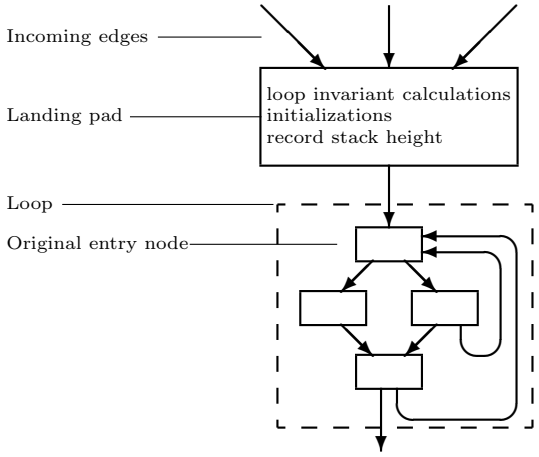

Figure 2: Typical loop with landing pad

When all storage allocated at a site within a loop is guaranteed to be live until loop exit then that site's allocation may be performed on the stack. Guaranteeing safety of stack allocation in this situation is more difficult because it more easily interferes with other constraints and because the analysis required is difficult (showing that storage must be live requires proof of liveness over all paths). Suppose a loop L contains a long allocation site A, and suppose there is a recursive call C after L. At C, it is necessary either that all storage allocated at A is live (and will be live past C) or that all storage allocated at A is dead. If liveness or deadness of all objects allocated at A cannot be guaranteed by the compiler, then stack allocation at A is potentially unsafe. The analysis is especially difficult because the compiler must prove not only that *an* instance must be live, but that *all* instances must be live. This is difficult because the multiple instances will necessarily be referenced from an array or linked data structure.

The most general condition for a loop allocation site to be short is that its lifetime within the loop cannot contain cycles passing through the allocation site. When this is the case the stack height can be reset to its entry height when traversing edges to the allocation site. Notice that the stack is not necessarily reset on edges back to the header node, and it is not reset at loop exit. This may leave one instance of the allocation on the stack after executing the loop, but

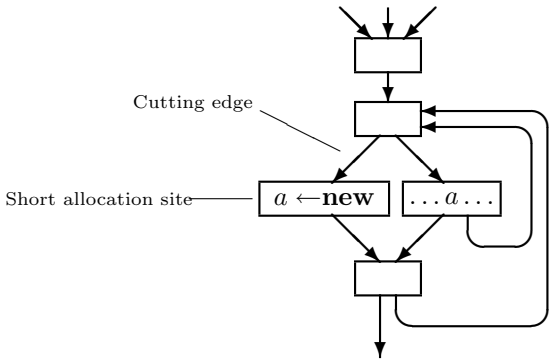

Figure 3: Short allocation example

this is safe by assumption and less restrictive. If the allocation size is loop-invariant then the allocation may be performed in the landing pad. This avoids the costs of adjusting the stack.

Unfortunately, if there is more than one short allocation site or a recursive procedure call within the loop then safe optimization is not so simple. If there is a recursive call in the loop, but not in the live range of a short allocation with loop-invariant size then it may be unsafe (pedantically speaking) to move the allocation into the landing pad. The difficulty arises because the storage is allocated before it is used; N recursive calls lead to N instances of unused storage reserved for future use[7]; make N large enough and available storage is exhausted. This problem is not helped by using heap instead of stack allocation in the landing pad; the storage is still reserved and can consume all of the heap if N is large enough. The example below demonstrates this problem.

```
f(x) = {
       for i ← 1 to N {
           if . . . then f(y)
           a ← really_big_allocation
           }
       }
```

When there is more than one short allocation site in a loop it is necessary to find a set of edges C (for *cut*) in the loop that is not in the live range of any short allocation site and that cuts all paths from the union of the live ranges to the allocation sites. Such a set may not exist; when this is the case finding the smallest number of sites to heap allocate that will allow (safe) stack allocation of the rest is an NP-complete problem (proof in appendix). The problem remains NP-complete if a minimum cost acyclic union of rooted acyclic lifetimes is sought (proof in appendix). It is not known (to the author, at this time) if there is a tractable algorithm for finding a minimum cost union of rooted acyclic lifetimes that contains no cycles through allocation sites. Notice also that recursive calls within the loop can interfere with this strategy for stack storage allocation because only some of the stack-allocated storage might be live at the call site.

[7] Again, I am ignoring the size of the allocations and assuming the worst.

6 Variations

6.1 Ineligible allocations

Short loop allocations are useful even when the lifetimes escape the containing procedure. Ordinarily such sites are not eligible for stack allocation, but if all references to the storage allocated there can be discovered at compile time then it is possible to copy the object into the heap after exit from the loop. This avoids the time and space cost of heap allocation within the loop. If the allocation size is loop-invariant (and there are no recursive procedure calls within the loop) then in-loop allocation and copying may be avoided altogether by performing a heap allocation in the landing pad.

6.2 Nested loops

When loops are nested the inner loops are treated as single nodes in the analysis of the outer loop and all allocations escaping the inner loop are treated as a single allocation in the outer loop (that is, the apparent lifetime in the outer loop of storage from sites in the inner loop is the union of the sites' actual lifetimes). It is possible for an allocation in the inner loop to be long or short in both loops, short in the inner but long in the outer, or long in the inner and short in the outer. All three of these cases are handled well by the approach described in the previous section. Examples of the four combinations possible are shown in figure 4.

When an allocation site is short in both loops that means that its lifetime in the inner loop contains no cycles back to the site and the lifetime in the outer loop also contains no cycles back to the inner loop (which is treated as a single node in that analysis). Each loop has a landing pad in which the stack height at loop entry is recorded and the stack is reset to this height when cutting edges (in either loop) are traversed.

For an allocation site to be short in an inner loop and long in an outer loop its lifetime *in the inner loop* must contain no cycles back to the site, escape the inner loop, and definitely escape the outer loop. To maintain safety other storage which escapes the inner loop must also definitely escape the outer loop.

Here the allocation for b is long in both loops.	$a \leftarrow$ **new_array**
	for $i \leftarrow 1$ **to** n **{**
	  for $j \leftarrow 1$ **to** m **{**
	  $b \leftarrow$ **new**
	  $a[i,j] \leftarrow b$
	  }
	  }

Here the allocation for b is short in the inner loop but long in the outer loop.	$a \leftarrow$ **new_array**
	for $i \leftarrow 1$ **to** n **{**
	  loop {
	  $b \leftarrow$ **new**
	  }
	  $a[i] \leftarrow b$
	  }

Here the allocation for b is long in the inner loop but short in the outer loop.	**for** $i \leftarrow 1$ **to** n **{**
	  $a \leftarrow$ **new_array**
	  loop {
	  $b \leftarrow$ **new**
	  $a[i] \leftarrow b$
	  }
	  }

Here the allocation for b is short in both loops.	**for** $i \leftarrow 1$ **to** n **{**
	  loop {
	  $b \leftarrow$ **new**
	  }
	  $\ldots b \ldots$
	  }

Figure 4: Nested allocations

In the outer loop the stack height is not reset because the memory allocated in the inner loop is treated as a long allocation, but in the inner loop the stack is reset to its entry height each time a cutting edge is traversed (note that the stack entry height is higher each time the inner loop is entered from the outer loop).

When an allocation is long in the inner loop it appears as a single large allocation to the outer loop. If it is also long in the outer loop then the stack grows in both inner and outer loops. If it is short in the outer loop then the stack is reset to its (outer loop) entry height each time before entering the inner loop.

6.3 Combined short and long allocations

A difficulty arises when both short and long allocations are in the same loop or recursive subroutine. Safe optimization of a short allocation site must reset the stack before performing another allocation at that site, but this is not possible if a long allocation (in the same loop nesting level or subroutine) occurs before the short allocation can be freed. Without information about the relative sizes of the allocations it is unsafe to optimize the short allocation in this situation.

When both a short and long allocation are in a loop it is necessary to keep two copies of the stack entry height; one determines the beginning of the long allocation (for later popping, if it is not implicitly popped at procedure exit) and the other determines the current reset height for short allocations. This second pointer is needed because the stack resets at cutting edges must only free short allocations, not long ones. It is unlikely that the compiler will be able to determine the size of a short allocation within a loop because if the size is constant, then it is also loop invariant and the allocation will be moved into the loop's landing pad (unless there is interference with recursive calls). Therefore, in loops containing long and short allocation sites a run-time check is necessary to prevent excess waste. If the size of the long allocation is known then the short allocation can be checked to ensure that it is not too large (allocating on the heap if it is). If the size of the long allocation is unknown at compile time then that size must be checked. If the allocation is not large enough compared to the storage used for short allocations in the latest iteration of the loop[8], then the storage must be obtained from the heap. When storage is obtained from the heap the reset stack height is not increased in order that the short allocations currently on top of the stack will be freed when the next cutting edge is traversed. Figure 5 shows the initializations and run time checks for a loop containing a long and a short allocation.

Combined long and short allocations in recursive subroutines cause problems. Overlap among short allocations is not a problem because all short allocations may be freed before a recursive call. Long allocations, however, may span arbitrary portions of the subroutine. Thus, they may interfere with short allocations and other long allocations. An ad hoc method for dealing with this is to only attempt to stack allocate long allocations that are live until exit from the subroutine; this imposes a stack discipline on the long allocations to prevent them from interfering with each other. Overlapping short allocations are then handled with the run-time checks used in loops.

[8] A good check for 50% waste is to compare the requested size of the long allocation with the difference between the current stack height and the current stack reset height.

In the landing pad	*reset_ptr* ← *stack_ptr* *base_ptr* ← *stack_ptr*
At the short allocation (*a* ← **new**(*size*))	*stack_ptr* ← *reset_ptr* *a* ← *stack_ptr* *stack_ptr* ← *stack_ptr* + *size*
At the long allocation (*b* ← **new**(*size*))	**if** *size* > *stack_ptr* − *reset_ptr* **then {** *b* ← *stack_ptr* *stack_ptr* ← *stack_ptr* + *size* *reset_ptr* ← *stack_ptr* **}** **else {** *b* ← *heap_allocation*(*size*) **}**

Figure 5: Run-time safety checks in a loop

6.4 Ignoring recursion

In many cases the problems caused by allocations that are not live across a recursive call or are trapped before a recursive call are more theoretical than practical. Most language implementations do not remove or null out or otherwise identify variables that become dead before exit from a subroutine. This means that a garbage collector will treat them as active and not collect the storage which they use, even though it could safely do so.

As long as the safety goal for allocation optimizations is to do no worse than an unoptimized implementation it is perfectly safe to stack allocate in this situation. Such a program, optimized, may waste stack space, but unoptimized it will waste heap space. An exception to this occurs when linked data structures are analyzed and portions are determined to be eligible for stack allocation. If such a structure is modified, then it is possible for all references to the portion to be lost and it becomes collectible. If the analysis for allocation optimization is this ambitious, then a call graph is necessary for safe, effective optimization. Interprocedural information will also improve the effectiveness of such an optimizer because subroutines are otherwise assumed to modify their parameters and global variables.

The other choice is to produce an implementation that is more robust than an unoptimized implementation. Here, storage references that are certainly dead across a recursive call can be removed before the call. This can improve the effectiveness of the garbage collector, though a small overhead per recursive call is imposed.

7 Putting it all together

This collection of details and interdependent rules needs to be converted into an algorithm if it is to be of any use in a compiler (or if it is to be compared to other techniques). It is also useful to note which rules require analysis so global or hard that it likely to be unavailable, since there is little point in the compiler asking unanswerable questions. So, here is one list of steps:

1. Identify loops and nested loops in the program's control flow graph (interval analysis is suitable for this if the graph is reducible).

2. Identify recursive call sites if the call graph is available (if not, then assume that all calls are recursive).

3. Determine storage lifetimes (using algorithms of Schwartz, Ruggieri, or Chase) and eligible allocations.

4. Find loops containing no recursive calls. In these move all loop-invariant single-instance allocations into the appropriate landing pads.

5. Determine short allocations. In this context a "short" allocation is one whose lifetime does not extend into a recursive call and contains no edges back to the allocation site.

6. Stack allocate short allocations from the inside out. Those most deeply nested in loops are regarded as able to generate the most garbage and consume the most time in allocation, and thus are the most important to perform on the stack. Conflicts (unions of allocation lifetimes containing cycles through allocation sites) must be resolved in an ad hoc manner since optimal solution is intractable. Insert stack resets on cutting edges and before recursive calls. In the case of overlapping short lifetimes it is necessary to observe LIFO order when freeing storage, but because all allocations are short it is guaranteed that this will be possible because no short allocations are live at recursive call sites. (Care must be taken when an allocation is live into a loop containing a recursive call site; a set of cutting edges consistent both with the allocations in the loop and the allocation outside the loop must be found (it will exist) and the reset stack height will be chosen so that the outer allocation is freed the first time a cutting edge is traversed.)

7. Find remaining allocations of small constant size with lifetimes that do not include edges back to the allocation site and that do not extend into a recursive tail-call. A subset of these will be stack-allocated before all others at entry to the subroutine. This subset is chosen by (1) selecting the smallest allocations and (2) keeping the sum of the sizes below the maximum waste allowed given the size of the subroutine's activation record. Small allocations are chosen so that as many as possible can be stack-allocated.

At this point the eligible allocations left untreated are long allocations within loops and those of large or unknown size which extend into or across recursive calls. I have chosen to ignore long loop allocations and allocations across recursive calls because long loop allocations require the analysis of linked structures and arrays and because overlap with long allocations can prevent both long and short allocations. Since short allocations are more easily handled, I chose to optimize them first.

It is clear that a call graph is very beneficial to this analysis, since worst-case assumptions will otherwise prevent many stack-allocations. Note that within non-recursive subroutines all eligible allocations outside of loops are short, and that those within loops may be handled by the run-time checks described in section 6.3.

8 Comparison with other strategies

This strategy is designed to be safe, and is in fact at least as safe as any other. Stack allocation of numbers in Lisp compilers appears to be more general, but is in fact more general only when it is unsafe (that is, when they are stack-allocated into a tail-call). Extension of the Lisp strategy for numbers to other data types (strings, for example) is unsafe. The interval-based strategy proposed for the SETL compiler is both less general and unsafe; this approach can allocate storage with lifetime escaping an interval, while the interval approach cannot, but the interval strategy only reclaims stack storage on exit from an interval (allowing unbounded waste within loops).

The strategies of Barth and Ruggieri are both safe because they fall back on garbage collection, but are also less efficient. The strategy proposed here can be combined with their approaches to allow the more efficient use of uncollectible stack storage where it is safe.

Allocation optimization may be more suited to trace-and-sweep and reference counting collectors than it is to the newer (generational) compacting collectors [LH83, Moo84, Ung84]. Appel and MacQueen [AM87] have noted that with sufficient memory the asymptotic cost of allocation and collection of an object reduces to the cost of object allocation, which (with a compacting collector) is comparable to the cost of stack allocation. As Appel also noted, increasing the size of the stack can slow down incremental collection algorithms. However, compacting collectors have other costs [Cha87] and are not always compatible with the (admittedly grubby) semantics of common programming languages.

9 Conclusions

Safety conditions for storage allocation are complex, but can be satisfied for several general classes of allocation. It appears that a call graph is necessary for useful safety analysis.

Good treatments of long allocations are still lacking. Long loop allocations need analysis that doesn't exist yet, and long allocations in recursive subroutines cause problems by overlapping and interfering with short allocations and other long allocations. Optimal selection of short loop allocations is intractable, so (if allocation optimization is to be used) heuristics need to be developed. I have not addressed these problems in this paper.

Finally, practical experience is needed. This paper was written to help determine what is needed for safe storage allocation and what is practical and possible. It is not known how many allocations will be eligible, nor is it known how many of these will be judged "safe", nor if the safety conditions described here are in practice too conservative.

Acknowledgements

Keith Lantz was kind enough to proofread the abstract.

References

[ACK81] F. E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 79–101. Prentice-Hall, 1981.

[AM87] A. W. Appel and D. B. MacQueen. A standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324, 1987.

[Bar77] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[BD85] Hans-Juergen Boehm and Alan Demers. Implementing Russell. Technical Report COMP TR85-25, Rice University, 1985.

[BGS82] Rodney A. Brooks, Richard P. Gabriel, and Guy L. Steele Jr. An optimizing compiler for lexically scoped LISP. In *SIGPLAN Symposium on Compiler Construction*, pages 261–275, 1982.

[Cha87] David Chase. *Garbage Collection and Other Optimizations*. PhD thesis, Rice University, 1987.

[CLZ86] Ron Cytron, Andy Lowry, and Ken Zadeck. Code motion of control structures in high-level languages. In *Conf. Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, 1986.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[KKR+86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *SIGPLAN Symposium on Compiler Construction*, pages 219–233, 1986.

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

[MJ76] Steven S. Muchnick and Neil D. Jones. Binding time optimizations in programming languages: Some thoughts toward the design of an ideal language. In *Conf. Record of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 77–91, 1976.

[MJ81] Steven S. Muchnick and Neil D. Jones, editors. *Flow Analysis and Optimization of LISP-like Structures*. Prentice-Hall, 1981.

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

[Mur84] Thomas P. Murtagh. A less dynamic memory allocation scheme for ALGOL-like languages. In *Conf. Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 283–289, 1984.

[RM88] Christina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conf. Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, 1988.

[Rug87] Christina Ruggieri. *Dynamic Memory Allocation Techniques Based on the Lifetimes of Objects*. PhD thesis, Purdue University, August 1987.

[Sch75] J. T. Schwartz. Optimization of very high level languages—I. Value transmission and its corollaries. *Journal of Computer Languages*, 1:161–194, 1975.

[Ste77] Guy L. Steele Jr. Fast arithmetic in MacLISP. In *Proceedings of the 1977 MACSYMA Users' Conference*, pages 215–224, 1977.

[Ste78] Guy L. Steele Jr. RABBIT: A compiler for SCHEME. Technical report, Massachusetts Institute of Technology, May 1978.

[Ste88] Guy Steele Jr. Personal communication, January 1988.

[Ung84] David 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*, pages 157–167, 1984.

A NP-completeness proofs

Here are NP-completeness proofs for two problems related to optimal choice of short stack allocation sites in a loop. In both cases a solution with cost less than K can be verified in polynomial time, so both problems are contained in NP. Both problems solve a polynomial-time transformation of Vertex Cover [GJ79], so they are NP-complete.

A.1 Min-cost acyclic union of acyclic lifetimes

Given a rooted strongly connected graph G with no embedded cycles not passing through root (R), and given a set S of rooted acyclic subgraphs D_i in G such that their union contains a cycle, find the smallest set of DAGs to remove from S so that the union of the remaining DAGs is acyclic. Vertex Cover can be transformed to an instance of this problem in polynomial time.

Given an undirected graph $G' = (V', E')$, create a rooted directed graph $G = (V, E)$ with root vertex R and a set of rooted DAGs using the following steps:

1. For each v_i in V', add vertices a_i, b_i, and i to V and edges $\langle R, a_i\rangle$, $\langle b_i, R\rangle$, $\langle a_i, b_i\rangle$, $\langle R, i\rangle$, $\langle i, b_i\rangle$ to E. The resulting graph is rooted at R and strongly connected, and all cycles pass through the root.

2. For each v_i in V', create a DAG D_i with edges $\langle i, b_i\rangle$, $\langle b_i, R\rangle$, $\langle R, a_i\rangle$. At this point the union of the DAGs is acyclic. See figure 6.

3. For each edge $e = \langle v_i, v_j\rangle$ in E', add vertices $c_i^j, d_i^j, c_j^i, d_j^i$ to G, D_i, and D_j and edges

   $$\langle a_i, c_i^j\rangle, \langle a_i, c_j^i\rangle, \langle c_i^j, d_i^j\rangle, \langle d_j^i, b_i\rangle, \langle i, d_j^i\rangle$$

   to D_i and G and edges

   $$\langle a_j, c_j^i\rangle, \langle a_j, c_i^j\rangle, \langle c_j^j, d_j^i\rangle, \langle d_i^j, b_j\rangle, \langle j, d_i^j\rangle$$

   to D_j and G. See figure 7. For each i, D_i is still acyclic and rooted at i, and G is still strongly connected and still contains no cycles not passing through R. Note that the union of D_i and D_j contains 4 simple cycles through R which correspond to the edge between v_i and v_j.

Consider the effects of not including D_i in the union of the DAGs; each set of four cycles broken corresponds to an edge removed if v_i is not included in G'. The smallest set of DAGs excluded from the union corresponds to the smallest vertex cover of G'.

A.2 Min-cost union with no cycles through roots

Given a rooted strongly connected graph G with no embedded cycles not passing through root (R) and a set S of rooted subgraphs S_i in G such that each S_i does not contain a cycle through S_i's root, and given that the union of the subgraphs does contain a cycle through the root of some S_i, find the smallest set of subgraphs to remove from S so that the the union contains no cycle through the root of any S_i. Vertex Cover can be transformed to an instance of this problem in polynomial time.

Given an undirected graph $G' = (V', E')$, create a rooted directed graph $G = (V, E)$ with root vertex R and subgraphs S_i using the following steps:

1. For each v_i in V', add vertex i to V and edges $\langle R, i\rangle$ and $\langle i, R\rangle$ to E.

2. For each v_i in V' create subgraph S_i with edge $\langle i, R\rangle$ and, for each j such that $\langle v_i, v_j\rangle$ is an edge in G', edges $\langle j, R\rangle$ and $\langle R, j\rangle$. Thus, vertex i in G will lie in the cyclic portion of subgraph j if and only if $\langle v_i, v_j\rangle$ is an edge in G'.

Clearly, selecting subgraphs to remove from S so that the union of subgraphs in S contains no cycles through a root of a subgraph in S is equivalent to finding a vertex cover for G'.

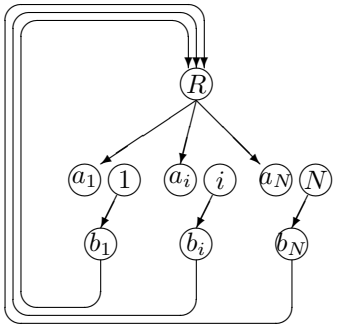

Figure 6: Initial DAG union

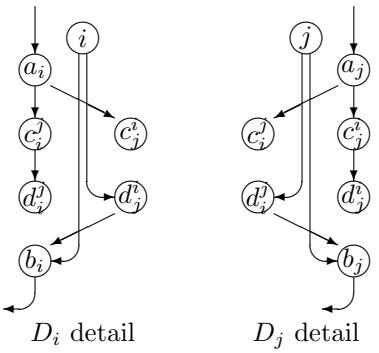

Figure 7: Effects of edge $\langle v_i, v_j \rangle$