

Static Garbage Collection for L3 (15-745)

Neelakantan Krishnaswami (neelk+@cs.cmu.edu)
William Lovas (wlovas+@cs.cmu.edu)

2 April 2006

Abstract

A simple, type-based escape analysis allows for an optimization that stack allocates some ordinarily heap-allocated intermediate data structures. This optimization can have a significant impact: 20-40% speedup on certain programs.

1 Introduction

Safe languages like L3 use garbage collection to automatically manage the allocation and deallocation of program objects. Garbage collection has the advantage that it is provably safe, eliminating a large class of program errors.

However, the reachability heuristic collectors use is always an approximation to the actual lifetimes of program objects, and so garbage-collected programs can use more memory than they really need. Since modern architectures have a very complex memory hierarchy¹ with severe performance penalties for moving down the hierarchy, this means that excessive memory usage can potentially have a drastic impact on the performance of a program.

Furthermore, we conjecture that many programs make use of intermediate data structures allocated in a stack discipline. Because L3 is statically typed and first-order, we can employ a simple, type-based analysis to determine a conservative approximation of escapability. Using the results of this analysis, we instrumented the L3 compiler to replace heap allocation with stack allocation in cases where the types showed that an allocation's escape was impossible. In our benchmark, this optimization led to significant performance improvements.

In this way, we hope to avoid having to implement the very complex analyses[1, 2] that are necessary in languages like C, due to unsafe features like type casts.

2 Design

Our analysis detects the impossibility of escape using the types of functions. In the absence of global variables, the only ways an allocation can escape from a

¹Three levels of cache, main memory, plus disk-based virtual memory is not an atypical memory hierarchy.

function $f : \tau_1, \dots, \tau_n \rightarrow \tau$ are through its arguments and its return value. So first, we classify the sorts of types whose values might escape via another type, denoted $E[\tau]$. $E[\tau]$ is the least fixed point of the following recursion equations:

$$\begin{aligned} E[\text{int}] &= \emptyset \\ E[\text{bool}] &= \emptyset \\ E[\text{void}] &= \emptyset \\ E[\tau^*] &= \{\tau^*\} \cup E[\tau] \\ E[\{l_1 : \tau_1, \dots, l_n : \tau_n\}] &= \bigcup_{i=1}^n E[\tau_i]. \end{aligned}$$

We require the least fixed point since **struct** types in L3 may be recursive. This least fixed point is easy to compute using a depth-first traversal of a type.

Given a function $f : \tau_1, \dots, \tau_n \rightarrow \tau$, we can determine whether an allocation of type τ^* can escape simply by checking if $\tau^* \in E[\tau] \cup (\bigcup_{i=1}^n E[\tau_i])$. If escape is impossible, we may replace the allocation with a stack allocation.

3 Implementation

To implement our optimization, we added a production to the abstract syntax of L3 called **StackAlloc**, with the same parameters as the **Alloc** production. Then a simple walk over the abstract syntax of a program replaces **Allocs** with **StackAllocs** whenever the type-based analysis deems this safe.

We next added an IR tree expression **ALLOCA e**. Source-language **StackAlloc** expressions are translated in the same way as **Alloc** expressions — the input is error-checked and extra space is allocated to make the fat pointers — except the core runtime call, **CALL ("_alloc", [e])**, is replaced by an IR **ALLOCA e** expression.

An IR statement of the form **MOVE (dest, ALLOCA e)** roughly generates assembly code like **sub t1, %esp; mov %esp, t2**, if **dest** generates code that stores the destination in temporary **t2** and **e** generates code that stores the result of the expression into **t1**. Thus an external call to the runtime allocation facility becomes two x86 assembly instructions.

4 Results

We tested our optimization on an L3 program that computes the leaves of a binary tree; the source code is included in Appendix A. The program takes as input a binary tree containing integer values at each node and produces as output an array of integers containing the values stored at the leaves of the tree. To compute the leaves, it allocates two internal linked lists: one containing a “work queue” of subtrees left to process and one containing the leaves found so far. Both lists cannot escape the **leaves** function, and our analysis detects this.

| number of leaves | unoptimized (s) | optimized (s) | improvement |
|------------------|-----------------|---------------|-------------|
| 32K | 0.10 | 0.08 | 20% |
| 64K | 0.20 | 0.14 | 30% |
| 128K | 0.39 | 0.27 | 31% |
| 256K | 0.85 | 0.49 | 42% |
| 512K | 1.51 | 1.20 | 21% |
| 1024K | 3.52 | 2.25 | 36% |
| 2048K | 6.54 | 4.97 | 24% |

Table 1: Running time for optimized “leaves” program versus unoptimized “leaves” program. (Tests were run on a 3 GHz Pentium 4 with 1 GB of RAM using a stack size of 128 MB.)

We compiled the “leaves” program using both our optimizing compiler and a reference compiler without our stack allocation optimization, varying the number of leaves in the input tree; the results are shown in Table 1. Our optimized code runs between 20% and 40% faster than the unoptimized code, demonstrating the effectiveness of stack allocation in this test case.

We imagine similar test cases based on, for example, breadth-first search of a graph, a task that requires an intermediate non-escaping queue data structure.

5 Conclusions and future work

Automatic static stack allocation is feasible and useful for a language like L3. Our design and implementation favored simplicity above all else and achieved significant results; it would be worthwhile to see how our analysis could be refined or extended to an interprocedural analysis to account for global variables and procedure calls.

Our analysis was simple chiefly because L3 is both type-safe and first-order. In a more expressive language with first-class functions like Scheme or ML, we would need some sort of lambda-aware control-flow analysis similar to that proposed by Shivers [3]. It is not immediately clear whether our design could be extended cleanly to such a language. If so, it would be an excellent optimization for mostly functional programs, since they tend to allocate lots of temporary storage.

References

- [1] Vikram Adve, Dinakar Dhurjati, Sumant Kowshik and Chris Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of Languages Compilers and Tools for Embedded Systems 2003*, San Diego, CA, June 2003.
- [2] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In

Proceedings of the ACM Conference on Programming Language Design and Implementation, pages 282–293, June 2002.

- [3] Olin Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, Atlanta, Georgia, 1988.

A “leaves” benchmark

```
void print_int(n : int) foreign
void print_space(n : int) foreign
void print_newline() foreign

struct inttree {
    val : int;
    left : inttree *;
    right : inttree *;
};

struct intlist {
    head : int;
    tail : intlist *;
};

struct treelist {
    head : inttree *;
    tail : treelist *;
};

#define Nil NULL
/* effectful */
#define Cons(type, h, t, tmp) (tmp) = alloc(1, type); \
                             (tmp)->head = (h); \
                             (tmp)->tail = (t); \
                             (t) = (tmp);

int *leaves(tree : inttree *) {
    var t : inttree *;
    var worklist, wl : treelist *;
    var leaveslist, ll : intlist *;
    var leavesarr : int *;
    var nleaves, i : int;

    leaveslist = Nil;
```

```

worklist = Nil;
Cons(treelist, tree, worklist, wl);

while (worklist != Nil) {
    /* take the next thing off the worklist and setup to work on the tail */
    t = worklist->head;
    worklist = worklist->tail;

    if (t->left == NULL && t->right == NULL) {
        /* found a leaf!  cons it onto the list. */
        Cons(intlist, t->val, leaveslist, ll);
    } else {
        /* not a leaf -- cons the subtrees onto the worklist */
        if (t->right != NULL) {
            Cons(treelist, t->right, worklist, wl);
        }
        if (t->left != NULL) {
            Cons(treelist, t->left, worklist, wl);
        }
    }
}

/* get the length of the list */
nleaves = 0;
ll = leaveslist;
while (ll != Nil) {
    nleaves += 1;
    ll = ll->tail;
}

/* and turn them into an array */
leavesarr = alloc(nleaves, int);
ll = leaveslist;
i = 0;
while (ll != Nil) {
    leavesarr[i] = ll->head;
    ll = ll->tail;
    i += 1;
}

/* finally, return the result */
return leavesarr;
}

/* assumes n is a power of two */
inttree *build_tree(n : int) {

```

```

var tree : inttree *;
tree = alloc(1, inttree);

if (n == 1) {
    tree->val = 7;
    tree->left = NULL;
    tree->right = NULL;
} else {
    tree->val = 5;
    tree->left = build_tree(n / 2);
    tree->right = build_tree(n / 2);
}

return tree;
}

void main() {
    var tree : inttree *;
    var leaves : int *;
    var i : int;

    tree = build_tree(1024); /* vary this parameter */

    leaves = leaves(tree);

    for (i = 0; i < size(leaves); i += 1) {
        print_int(leaves[i]);
        print_space(1);
    }

    print_newline();
}

```