CS270 Project - Spring 1998

David Gay
EECS Department
University of California, Berkeley
dgay@cs.berkeley.edu

April 8, 1998

1 Background

In a region-based memory allocation scheme each allocated *object* is placed in a program-specified *region*. Memory is reclaimed by destroying a region, freeing all the objects allocated therein. A simple example is shown in Figure 1. Each iteration of the loop allocates a small array. The call to `deleteregion` frees all arrays.

```
void f() {
  Region r = newregion();

  for (i = 0; i < 10; i++) {
    int *x = ralloc(r, (i + 1) * sizeof(int));
    work(i, x);
  }
  deleteregion(r);
}
```

Figure 1: An example of region-based allocation.

In our system [GA98], a region r can be deleted only if there are no *external* references to objects in r (a reference external to r is any pointer not stored within r). This rule is enforced by keeping a *reference count* for each region; deleting a region is a no-op when the reference count is nonzero. However, maintaining reference counts is generally considered to be expensive, particularly because updates of local variables are very frequent.

Thus, an efficient version of region-based memory management must reduce or eliminate the cost of performing reference count updates for local variables. In [GA98] we update a region's reference count to take account of local variables only when deleting a region. This update is performed by scanning the stack to find all these local variables. However, this scan requires that the compiler be modified. Also, this stack scan is a fairly expensive operation.

2 Problem

An alternative approach to reducing the cost of reference counting local variables is based on the following observation: for each local variable v, the statements of a function f can be divided into three sets:

- S: The statements were the reference count for v must be taken into account. The variable v is said to be *scanned*. These statements are all calls to functions that may delete a region.
- U: The statements where the reference count for v must not be taken into account. The variable is said to be *unscanned*. These statements are the entry and exit points of the function and assignments to v.
- O: All other statements.

The *scan* operation changes variable v from unscanned to scanned by incrementing the reference count of the region containing the object pointed to by v. An *unscan* operation changes variable v from scanned to unscanned by decrementing the reference count of the region containing the object pointed to by v. Both of these operations take only a few machine instructions.

By assigning every statement in O to either S or U, the necessary scan and unscan operations are completely determined. The aim is to choose such an assignment that minimises the total time spent in scan and unscan operations, and hence the cost of reference counting local variables. This problem is named MinRC in the rest of this paper.

3 Formalising MinRC

Formally, each function f has an edge-weighted control-flow graph $G = (V, E, W)$ and local variables $v_1, v_2, \ldots, v_n$. The edge weights correspond to execution frequencies. The sum of the weights on all incoming edges to a node n must be equal to the sum of the outgoing weights for all nodes n except the entrance and exit nodes for f.

Each local variable v_i is considered independently. Each node n of G is assigned an initial colour:

- *white*: the reference count for v_i must be taken into account at n.
- *black*: the reference count for v_i must not be taken into account at n.
- *grey*: all other nodes.

Solving MinRC then becomes equivalent to assigning the colour white or black to each grey node so as to minimise the sum of weights between white and black nodes in the resulting coloured graph.

Some properties are immediately apparent:

- If all predecessors of a grey node n are white (respectively black) in an optimum solution, then there is an optimum solution where that node is white (respectively black). Proof: k is the sum of the weight of incoming edges to n. If all predecessors are white and n is black, then the cost at this node is at least k. If n is white, then the cost is at most k (in the case where all successor of n are black).
- If all successors of a grey node n are white (respectively black) in an optimum solution, then there is an optimum solution where that node is white (respectively black).

4 Solution

The solution to MinRC turns out to be easy: it is a MinCut problem with a specified source and sink. This MinCut can be found by solving MaxFlow [FF56] in the undirected graph $G' = (V', E', W')$ built from $G = (V, E, W)$ as follows:

- $V' = \{v | v \in V \wedge v \text{ is a grey node}\} \cup \{sb, sw\}$ where sb is a "super-black" produced by merging all black nodes in G, and sw is a "super-white" node produced by merging all white nodes in G. Pick sw as the source and sb as the sink for MaxFlow .
- E', W' are the obvious edges and weights obtained when merging the black and white nodes of G: multiple edges between the same pairs of nodes are merged into a single edge with weights summed.

The cut is then found by disconnecting all saturated edges. All nodes reachable from sb are black, all other nodes are white.

The best complexity for MaxFlow is $O(|V'||E'| + |V'|^2 \log F)$. In control-flow graphs, $|E'| \leq 2|V'|$, and the maximum edge weights are fairly small (definitely less than 32 bits), so the complexity is effectively $O(|V|^2)$. A separate version of MinCut must be computed for each local variable, so the total complexity is $O(|V|^3)$ (assuming the number of local variables is proportional to the function size).

5 Extensions

The code space necessary for MinRC depends on the number of scan/unscan operations, i.e., on the number of edges in the minimum cut. This number of edges can be minimised without affecting the result or complexity of MinCut (see exercise 7.27 of [AMO93]).

The cubic complexity of the algorithm presented above may make it too slow for practical use on large functions. In that case, there are several potential paths for improvement:

- MinRC can be solved independently for each maximum connected subset O of the grey nodes by considering the graph restricted to O and the adjacent white and black nodes.
- Goldberg and Tarjan [GT88] point out that MinCut requires half the work of MaxFlow in push/relabel algorithms: the algorithm can terminate once the saturating edges are known without producing a complete and correct flow.
- Arikati, Chaudhuri and Zaroliagis [ACZ95] present an algorithm for preprocessing a sparse graph to allow MinCut to be computed efficiently (in near-constant time) between any pair of vertices. This can be used to compute our $|V|$ minimum cuts in $O(|V|^2 + \gamma^4 \log \gamma)$ for a parameter γ that depends on the graph. Simple all-pairs MinCut algorithms like [SW97] are not applicable because MinRC needs to know the cuts of a specific subset of all pairs, rather than the smallest amongst all pairs.

6 Related Work

Related work includes all the work on MaxFlow and MinCut , which I shall not attempt to survey here. References include [ACZ95, FF56, GT88, SW97].

A problem that is in some ways similar to MinRC is the *profiling problem* [BL94, Knu73]: profiling counters are placed in a weighted control-flow graph to determine the execution counts of all edges. The aim is to place these counters so as to minimise the runtime cost of profiling.[1] This edge profiling problem is solved

[1] Of course without previous profiling the execution weights on edges are necessarily guesses.

by placing profiling counters on all edges that are not in a maximum spanning tree of the control-flow graph [Knu73]. A similar problem, optimally placing profiling counters on edges to determine the execution counts of nodes, appears NP-complete [BL94].

7 Conclusion

In conclusion, I have found a polynomial time algorithm for placing reference count operations so as to minimise the runtime cost of reference counting. The next stage is to implement this algorithm to answer two questions:

- Does it provides significant advantages over simpler heuristics for placing scan/unscan operations (e.g., scan/unscan before and after every white node) ?
- Is this cubic-in-function-size algorithm too slow for practical use in a compiler ? If this turns out to be the case, then the ideas for optimisation in Section 5 should be investigated.

References

[ACZ95] S. R. Arikati, S. Chaudhuri, and C. D. Zaroliagis. All-pairs min-cut in sparse networks. *Lecture Notes in Computer Science*, 1026:363–??, 1995.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[BL94] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[FF56] L. Ford and D. Fulkerson. Maximal flow through a network. *Canadian J. Math.*, 8:399–404, 1956.

[GA98] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998. to appear.

[GT88] A. Goldberg and R. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[Knu73] D. Knuth. *The Art of Computer Programming (Vol. 1: Fundamental algorithms)*. Addison-Wesley, Reading, Mass., 1973. 2nd ed.

[SW97] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, July 1997.