A Better Global Progressive Register Allocator

David Koes*
Seth Copen Goldstein*

* *School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213*

ABSTRACT

We present an improvement to the simultaneous heuristic allocator component of the global progressive register allocator described in our previous work [Koes06]. Our improved allocator decomposes the control flow graph into linear traces which are allocated in the same manner as a single basic block. We investigate two methods for handling the control flow within the traces both of which produce better quality allocations than the simultaneous heuristic allocator.

KEYWORDS: Register Allocation; Progressive Solver

1 Introduction

A *global progressive register allocator*, as described in [Koes06], uses an expressive model of the register allocation problem to quickly find a good allocation and then progressively find better allocations until a provably optimal solution is found or a compilation time limit is reached. Our global progressive allocator uses a global multi-commodity network flow (GMCNF) model to explicitly capture the important components of register allocation such as spill code optimization, register preferences, coalescing, and rematerialization. Our progressive solution technique utilizes the theory of Lagrangian relaxation to compute a lower bound on solution quality and to calculate Lagrangian prices on constrained nodes in the GMCNF model which are used to push heuristic allocators closer to the optimal solution.

In the GMCNF model of register allocation an allocation of a variable within a basic block exactly corresponds to a path in a constrained network. At each iteration of our progressive algorithm, we update the Lagrangian prices on each node in the network. Then a heuristic allocator constructs an allocation where the priced cost of each variable's allocation is as close as possible to the priced cost of allocating that variable irrespective of the other variables. As the prices converge, allocations that meet this criteria are likely to be optimal. We have developed a heuristic simultaneous allocator that attempts to achieve this goal. In this paper we describe an improvement to this simultaneous allocator that performs better in the presence of control flow.

[1]E-mail: {dkoes,seth}@cs.cmu.edu

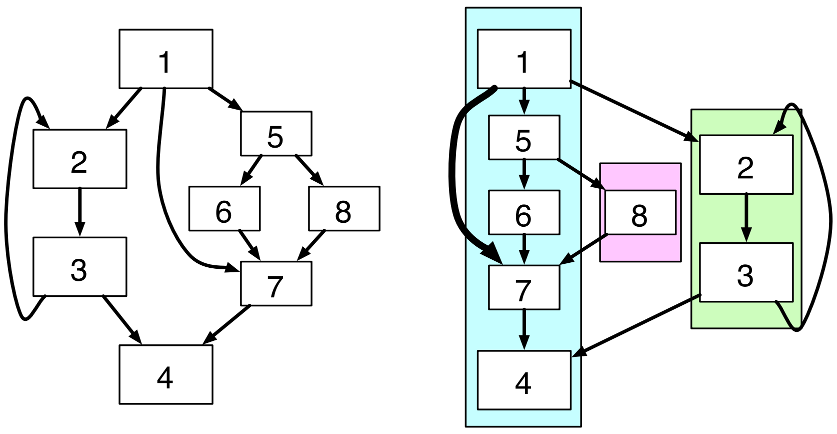

Figure 1: A example control flow graph (left) decomposed into traces (right). The bold edge is an example of control flow internal to a trace that complicates allocation.

The simultaneous allocator functions similarly to a second-chance binpacking allocator [Trau98] but uses the priced GMCNF model to guide eviction decisions. The allocator traverses the control flow graph in depth first order. For each block, it computes both a forwards and backwards shortest-path for every variable. These paths take into account that the entry and exit allocations of a variable may have been fixed by an allocation of a previous block. After computing the shortest paths, the allocator scans through the block, maintaining an allocation for every live variable. At each program point (a level in the network), the allocation of every live variable is updated to follow the previously computed shortest path (the common case is for a variable to remain in its current location). If necessary, variables may be evicted. The cost of evicting a variable from its current location is computed by finding the shortest path in the network to a valid eviction edge. When a variable is defined, the minimum cost allocation is computed using the shortest path information and the cost of any necessary eviction.

The simultaneous allocator is effective at minimizing the total priced cost of the final solution, in large part because the eviction mechanism allows it to undo poor previous allocation decisions. However, allocation decisions become fixed at basic block boundaries. The simultaneous allocator relies on prices on nodes at block boundaries to avoid locally good, globally poor, allocation decisions.

2 Trace-Based Allocation

In an attempt to improve upon the simultaneous allocator we have developed a trace-based simultaneous allocator. Instead of processing each basic block independently, the trace-based allocator decomposes the control flow graph into linear traces of basic blocks, which may contain internal and external control flow, and allocates each trace similarly to how a single basic block is allocated by the simultaneous allocator. To construct our traces we simply find the longest possible traces using depth first search while ensuring that loop headers start a new trace (as in the example in Figure 1).

The presence of control flow within each trace creates some complications. When computing shortest paths care must be taken to take the correct edge spanning basic blocks within a trace (there may be holes in a trace where a variable is not live). When an allocation decision is made at a block boundary, that decision must be propagated to all connected

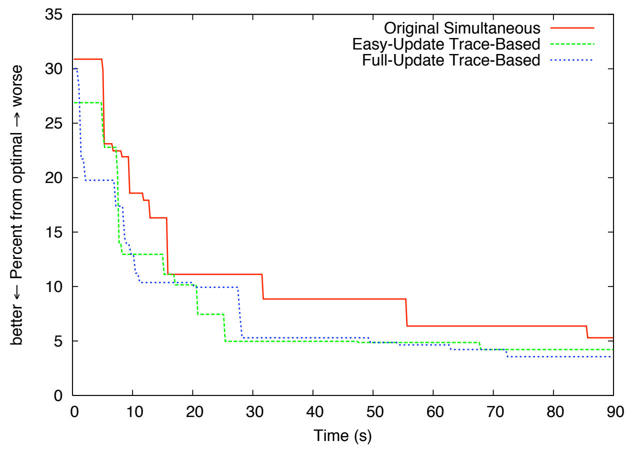

Figure 2: The performance of trace based allocators compared to the original simultaneous allocator when compiling the function quicksort.

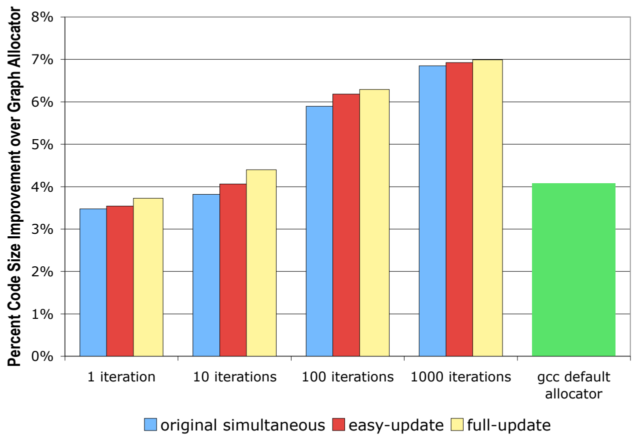

Figure 3: Average code size improvement relative to a graph coloring allocator. Measurements are taken immediately after allocation to eliminate downstream noise.

blocks within the trace. For example, the exit allocation of block 1 in Figure 1 fixes the starting allocation of both blocks 5 and 7 and the exit allocation of block 6 in the same trace. Similarly, it may not be straightforward to evict a variable across block boundaries if doing so affects other blocks in the trace.

We consider two techniques for propagating boundary allocation decisions within a trace. The first, *easy-update*, does the minimal amount of recomputation necessary for correctness. Only blocks directly effected by the boundary allocation have their shortest path computations redone. For example, in Figure 1, after allocating block 1, only block 6 would have to be recomputed as its exit allocations have changed. Although these recomputations result in extra work compared to the original simultaneous allocator, they are necessary for the correct allocation of the trace. The second technique, *full-update*, recomputes the shortest paths for all unallocated blocks prior to and including the blocks effected by the boundary allocation. The full-update technique is computationally more expensive (potentially quadratically more updates) but provides more up-to-date information for the simultaneous allocator in blocks not immediately affected by the boundary allocation. For example, in Figure 1, if a variable were to spill to memory and then be loaded back into a register in block 5, it would likely be best for the variable to be loaded into the same register it was allocated to at the exit of block 1 (to avoid a move into that register before the exit of block 6). With full-update the allocator would be aware of this cost since both blocks 5 and 6 would have been recomputed after the allocation of block 1.

3 Results

We compare the two trace-based allocators and the original simultaneous allocator on a representative function in Figure 2. The trace-based allocators are clearly better than the original allocator despite requiring more computation per an iteration. The easy-update technique takes 26% longer than the simultaneous allocator and the full-update technique takes 71% longer for this function. On a per-iteration basis the full-update technique outperforms easy-update, but because easy-update is much faster the actual benefit is less pronounced.

We evaluated our allocators by compiling a large selection of benchmarks from the SPEC-2000, SPEC95, MediaBench, and MiBench benchmark suits using our allocator implemented in `gcc` 3.4.4. We choose to optimize for code size since this metric, in addition to being important in the embedded community, can be exactly represented and measured statically. As expected, the trace-based allocators outperform the original simultaneous allocator when compared on a per-iteration basis (Figure 3). After only 10 iterations, the full-update technique gets an additional .55% average code size improvement over the original allocator. As more time is permitted for compilation and the Lagrangian prices converge, the difference between the trace-based allocators and the original allocator decreases, most likely because the more accurate prices push the original allocator closer towards the optimal solution (which the trace-based allocators may have already found).

Although slower than the original allocator, the traced-based allocators find better solutions in less time. When compiling for 10 iterations the easy-update and full-update allocators take 18% and 60% longer overall than the original allocator. After 12 and 16 iterations the original allocator has achieved an average code size improvement of 3.98% and 4.18% compared to improvements of 4.06% and 4.40% for the easy-update and full-update techniques after only 10 iterations implying that the benefits of the trace-based allocators outweigh the cost.

4 Future Work

We have presented an extension to prior work that allows a fundamentally local register allocator to better incorporate global information into its allocation decisions. The use of traces instead of basic blocks allows the allocator to perform well despite the initially approximate nature of the Lagrangian pricesin the GMCNF model. In addition to exploring other techniques for finding better allocations, future work will need to focus on reducing the computational overhead of these allocators. Some possible directions include: simplifying the GMCNF model (for example, reducing the amount of detail at points without register pressure), constructing traces that are informative but require less recomputation during allocation, and investigating more efficient methods of recomputation.

It is hopeful that with further algorithmic and code improvements register allocation based on a GMCNF model coupled with a Lagrangian solver will be both compile-time and code-quality competitive with current allocators while also being progressive, allowing users to trade compile-time for optimal or near-optimal solutions.

References

[Koes06] D. KOES AND S. GOLDSTEIN. A Global Progressive Register Allocator. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, June 2006.

[Trau98] O. TRAUB, G. HOLLOWAY, AND M. SMITH. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 142–151, New York, NY, USA, 1998. ACM Press.