Some Comments on 'The Priority-Based Coloring Approach to Register Allocation'

Arthur Sorkin
Tesuji, Inc.
12340 Indian Trail Road
Los Gatos, CA 95030
art@fuseki.com

Abstract: There is a problem with the definition of constrained live range and unconstrained live range in Chow and Hennessy's paper on priority-based register allocation that unnecessarily pessimizes the performance of the register allocator. Some corrections to the algorithm are suggested. Some problems with Chow and Hennessy's analysis of caller-save vs. callee-save are also discussed.

Categories and Subject Descriptors: D.3.4 {Programming Languages} Processors - compilers; optimization

General Terms: Languages, Theory

Additional Keywords and Phrases: register allocation, graph coloring, priority-based coloring, live-ranges

1. Constrained Live Ranges

In Chow and Hennessy[1] there is a problem with the definition of "constrained live range" and "unconstrained live range" that unnecessarily pessimizes the performance of the register allocator. Intuitively speaking, a live range is constrained if the live range has so many neighbors that there aren't enough registers available to allocate to all of them. A live range is unconstrained if it is guaranteed that there will always be enough registers available for it and all its neighbors. Chow and Hennessy follow Chaitin[2,3] in defining neighbors in terms of an interference graph, and this is the source of the problem. This definition also appears in previous versions of the paper[4,5].

In Figure 1, we have three live ranges, LR1, LR2, and LR3. Live range LR1 has two neighbors, LR2 and LR3 as shown by the interference graph. Chow and Hennessy state that "unconstrained live ranges have a number of neighbors in the interference graph less than the original number of registers available." Assuming that there are exactly two registers available and that each live range requires one register, then live range LR1 is constrained by Chow and Hennessy's definition. However, examining the basic blocks, it is obvious that no basic block requires more than two registers. Since LR2 and LR3, the neighbors of LR1, are disjoint, only two registers are required for all three live ranges, and LR1 is unconstrained.

A further illustration of this error occurs in the discussion of splitting live ranges. In Section 5.3 paragraph (5), Chow and Hennessy state that "because of the introduction of new live ranges, it is also possible that some unconstrained live range is made constrained." Consider the live ranges LR1 and LR2 in Figure 2. If we split LR2 into two pieces, LR3 and LR4, then LR1 has two neighbors instead of one, as shown in the interference graphs. Assuming again that there are exactly two registers and that each live range requires one register, the split would make LR1 constrained according to Chow and Hennessy's definition. However, since the pieces of LR2 after the split are disjoint, the number of registers needed by the live ranges remains the same both before and after the split. This can be verified by examining the basic blocks. In fact, splitting a

live range never increases the number of live ranges present in any basic block (precisely because the pieces of a split live range are always disjoint), and, therefore, cannot increase the number of registers needed in any basic block, i.e. splitting a live range can never cause unconstrained live ranges to become constrained.

This observation leads us to the corrected definition of constrained and unconstrained live ranges. A live range is *unconstrained* if for each of its basic blocks, the number of registers required by the live ranges defined in that block (including itself) is less than or equal to the number of registers available; otherwise, the live range is *constrained*. Unconstrained live ranges are guaranteed to have enough registers available for allocation, and can be dealt with separately after allocating all the constrained live ranges.

The constrained live ranges can be allocated according to priority, as stated in Chow and Hennessy. A priority queue of unprocessed constrained live ranges allows live ranges to be extracted in priority order in O(n log n) time and also to be inserted in O(n log n) time, and is, therefore, a good data structure for the unprocessed list. In each cycle of the algorithm, a constrained live range is extracted from the unprocessed list and allocated registers if it is possible to do so for that live range. It is not possible to allocate the registers required by a constrained live range if one of the two following conditions holds: (a) at least one of its basic blocks has fewer registers available than are required by the live range, or (b) for each individual basic block the number of registers available is greater than or equal to the number of registers required by the live range, but the set of specific registers that is available for the entire live range is smaller than the number of registers required by the live range, e.g., for a live range requiring one register, there is no *single* register that is available in every one of the live range's basic blocks, even though there *is* a register available in each of it's basic blocks.. One algorithm for determining if an unprocessed live range can be allocated is to iterate over its basic blocks and take the union of the set of registers already allocated in each block.

In the event that (a) holds, then the block or blocks in question must be deleted from the live range and spill code must be inserted for the deleted blocks, i.e., the live range is treated as a local in the deleted blocks. Deleting blocks from the live range will separate it into disjoint pieces. If any of the remaining pieces of the live range contains only a single basic block, then that piece should be deleted as well, since the live range is no longer global in that block. It is possible that by spilling some of the basic blocks in a live range that the live range is converted from a constrained live range into an unconstrained live range. In that case, it can be added to the unconstrained list for processing after all the constrained live ranges have been done.

In the event that (b) holds, then there are two obvious strategies. One is to split the live range into connected pieces where the same register is available and then reinsert the pieces of the live range into the unprocessed list for later processing according to the priorities of the new pieces. Ultimately, loads and stores may need to be inserted at the entries and exits of some or all of the new pieces. This is basically the strategy of Chow and Hennessy. It is possible that by splitting a constrained live range, some (but not all) of the new pieces of the live range can become unconstrained live ranges. In that case, they can be added to the unconstrained list for processing after all the constrained live ranges have been done.

The other strategy is to insert register-to-register copies at the boundaries of the connected pieces and to deal with the entire live range at once. A register copy is only required at a boundary between two basic blocks where a store on exit and then a load on entry would have been required. For most architectures, the register copy strategy should be cheaper than inserting loads and stores since it requires fewer operations to be inserted. It will also improve the running time of the register allocator since it doesn't create as many new unprocessed live ranges as always splitting live ranges. It would be possible to use a combination of both strategies using such factors as

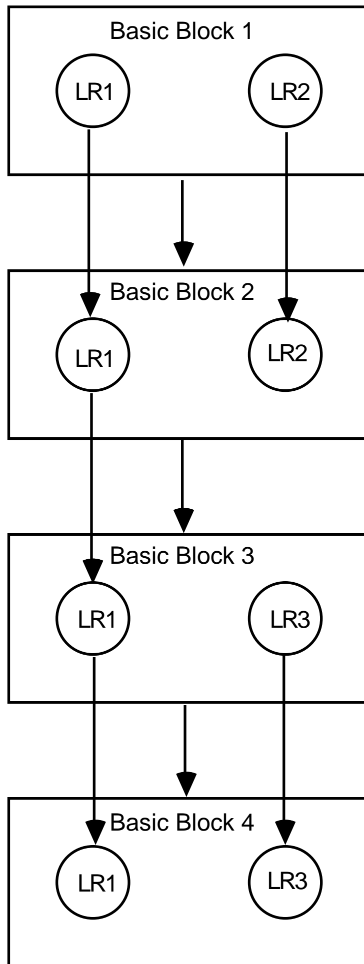
relative priorities and lifetimes to help make the choice of strategy.

2. Caller-Save vs. Callee-Save

Chow and Hennessy state that a mixed caller-save/callee-save strategy is superior to caller-save, and that caller-save is superior to callee-save, and they present experimental evidence for their position. One must be very careful to expose all assumptions before drawing such conclusions. For example, the Cydra 5 mini-supercomputer was a pipelined machine capable of initiating one operation per cycle in uni-op mode. However, the memory latency was on the order of 20 cycles. Since updating the stack frame required two memory references for a call and return, there were more than enough cycles available to completely overlap the saving and restoring of 32 registers. On the other hand, to completely overlap saves before a call was difficult because they had to complete before the call was completed so that the registers were free when the callee was entered. It was even more difficult to overlap the restores, since they couldn't begin until after the return to the caller was completed and operations following the call were dependent upon the registers being restored before they were initiated. In the case of the Cydra 5, callee-save was clearly better than caller-save, and this *was* with a compiler that did global register allocation.

3. Summary and Conclusions

Using the corrected definition of constrained and unconstrained, it appears that the interference graph is unnecessary and should be discarded. In fact, the corrected algorithm seems to have nothing to do with graph coloring, and it is, at best, confusing to call it a coloring algorithm. It is an algorithm that maps registers to live ranges based upon priorities. What is minimally required is a mapping from each live range to its basic blocks and another mapping from each basic block to its live ranges.



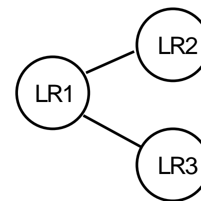
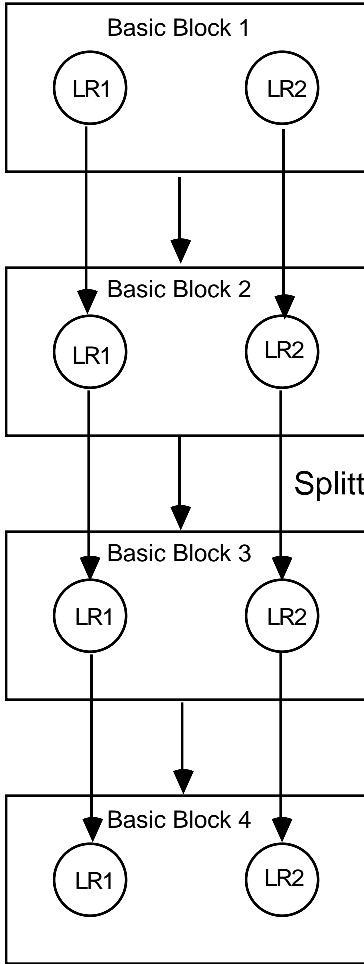


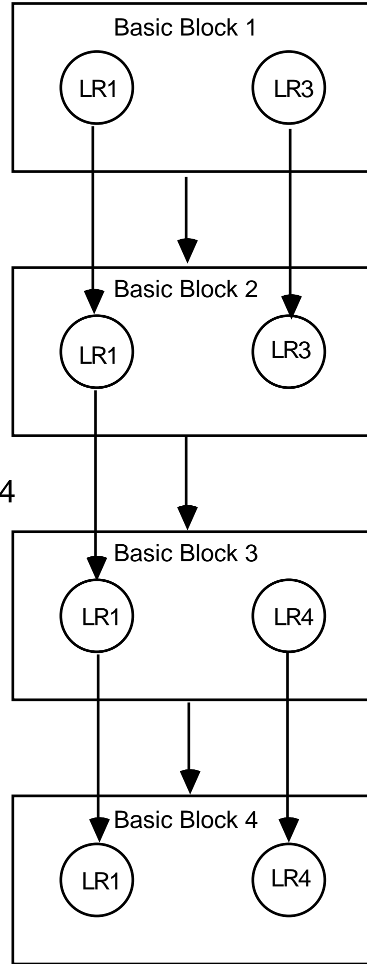
Figure 1

Basic Blocks and Live Ranges
Before Split

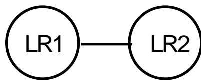


Splitting LR2 Into LR3 and LR4

Basic Blocks and Live Ranges
After Split



Interference Graph
Before Split



Splitting LR2 Into LR3 and LR4

Interference Graph
After Split

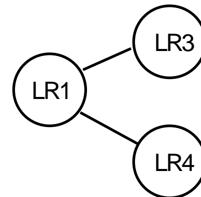


Figure 2

REFERENCES

1. Chow, F., and Hennessy, J. The Priority-Based Coloring Approach to Register Allocation, *TOPLAS* 12,4 (1990) pp. 501-536.

2. Chaitin, G. J. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction* (Boston, June 1982). ACM, New York, 1982, pp. 22-31.

3. Chaitin, G. J., Auslan der, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. Register allocation via coloring. *Comput. Lang.* 6 (1981), pp. 47-57.

4. Chow, F., and Hennessy, J. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction* (Montreal, June 1984). ACM, New York, 1984, pp. 222-232.

5. Chow, F. A portable machine-independent global optimizer -- Design and measurements. Ph.D. thesis and Tech. Rep. 83-254, Computer System Lab, Stanford Univ., Stanford, Calif., Dec. 1983.