

Aggressive Live Range Splitting[†]

Preston Briggs
Keith D. Cooper
Linda Torczon

Rice University
Houston, TX 77251-1892

1. The Problem

Over the past three years, we have built a series of register allocators based on the graph coloring paradigm. Over the same period, colleagues have involved us in two different studies involving source-to-source transformations. These studies have made us much more familiar with the shortcomings of the register allocators in several commercial compilers [CaCK 90, CoHT 90]. Some of these use the coloring paradigm, others do not.

In general, allocators based on coloring do a good job of keeping values in registers and keeping spill costs low. Nonetheless, we have *observed* a number of problems with the allocations produced by our allocator [BCKT 89], by our implementation of Chaitin's method [CACC 81, Chai 82], and by the implementation of Chow's method in the MIPS FORTRAN compiler [ChHe 90]. We believe that these misallocations are not caused by specific details of the implementations, but rather by the loss of information during the mapping from allocation into coloring.

These problems usually manifest themselves as a bad spill choice – the allocator picks the wrong live range or ranges to spill at some particular point in the code. These misallocations are becoming increasingly important. For example, on the Intel 860 microprocessor, a sixty-four bit floating point multiply takes two cycles. Loading a sixty-four bit number from cache takes two cycles. If the load causes a cache miss, the cost rises to twenty or more cycles. For the advanced microprocessors that are moving into the market today, memory traffic is more expensive than computation.

We are building a compiler that restructures loop nests to improve both cache and register locality. These transformations can drastically increase the ratio of useful computation to memory traffic, producing improvements of two to three hundred percent in running time [CaCK 90]. The compiler tries to minimize the number of loads and stores in the inner loops and to tune the register pressure to match the target machine's resources. If, after these transformations, the allocator unnecessarily inserts new loads and stores into that inner loop, in the form of spill code, it has negated the effects of the optimization. Further, because spilling moves unanticipated values into the cache, the spill code causes cache pollution and destroys the carefully planned locality.

The aim of this work is to address the sources of these misallocations. We believe that these problems arise due to information loss in the transformation of the problem from a register allocation problem to a graph coloring problem. Simply put, allocation is like coloring, but not identical to coloring. The mapping from allocation to coloring used in previous work discards information that would allow the allocator to avoid these observed problems.

2. Background

The notion of modeling register allocation as a graph coloring problem goes back to the early sixties.¹ The first actual implementation was done by Chaitin *et al.* in the PL.8 compiler [CACC 81]. Our own work has

[†] This work has been supported by the National Science Foundation, through grants CCR 86-19893 and CCR 87-06229.

¹ In the early sixties Ershov and his colleagues on the ALPHA project solved storage allocation problems by building an interference graph and using a packing algorithm on it [Ersh 62, Ersh 66]. Ershov credits the insight to Lavrov; his 1961 paper appears to be the first reference that makes the link between allocation problems and graph coloring [Lavr 61]. By the late sixties, Cocke was clearly talking about register allocation as a coloring problem; both Kennedy and Schwartz credit him with this insight [Kenn 71, Schw 73].

built on Chaitin’s scheme [BCKT 89].

To model register allocation as a graph coloring problem, the compiler first constructs a register interference graph G . The nodes in G represent live ranges and the edges represent *interferences*. Thus, there is an edge in G from node i to node j if and only if live range l_i *interferes* with live range l_j ; that is, they are simultaneously live at some point and cannot occupy the same register.²

To find an allocation from G , the compiler looks for a k -coloring of G , that is, an assignment of colors to the nodes of G such that adjacent nodes always have distinct colors. If we choose k to match the number of machine registers, then we can map a coloring into a feasible register assignment. Because graph coloring is NP-complete, the compiler uses a heuristic method to search for a coloring. It is not guaranteed to find a k -coloring for all k -colorable graphs. If it cannot find a k -coloring, it chooses some values to throw out of registers, an action called *spilling* in the jargon of allocation.

Spilling one or more live ranges creates a new and different interference graph. The compiler proceeds by iteratively spilling some live ranges and attempting to color the resulting new graph. This process must produce an allocation; on each iteration of the process, it spills some live ranges and reduces the size of the interference graph. In practice, our allocator rarely requires more than two trips through this loop.

Figure 1 shows how our allocator works. It proceeds in six phases.

- (1) *Renumber* systematically renames live ranges. It assigns a new virtual register number to each definition point. Then, it unions together the live ranges that reach each use point.³
- (2) *Build graph* constructs the interference graph. Our implementation closely follows the published descriptions of the PL.8 allocator.
- (3) *Coalesce* attempts to shrink the number of live ranges. It subsumes unneeded copies, eliminating the copy instruction itself and combining the live ranges. If coalescing changes the graph, we repeat *build graph* and *coalesce*. Coalescing proceeds from live ranges in inner loops to those in outer loops.
- (4) *Simplify graph* constructs an ordering over the nodes of G . It removes from G nodes with current degree less than k , pushes them on a stack, and adjusts the degree of their neighbors. If no such

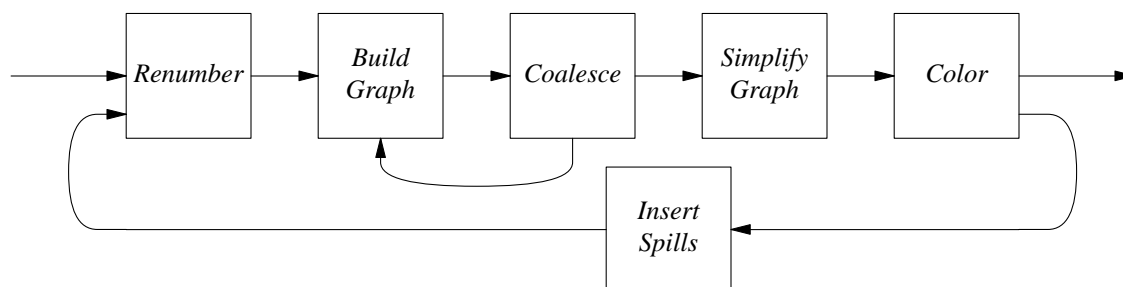


Figure 1 — the base allocator

² This is a simplified definition. For the ultimate definition of interference, see the 1981 PL.8 paper [CACC 81].

³ In the papers on the PL.8 compiler, this type of analysis is referred to as getting “the right number of names”. The HP Spectrum compiler papers refer to this type of analysis as “web analysis”. Our implementation models this as an example of the classic union-find problem.

node remains, it chooses a “spill candidate”, removes it from G , and pushes it on the stack.⁴

- (5) *Color* assigns colors to the nodes of the graph in the order determined by *simplify graph*. It repeats the simple loop: (a) pop a node from the stack; (b) insert it in G ; (c) give it a color distinct from its neighbors. If no color is available in step (c), it leaves the node uncolored.
- (6) *Insert spills* is invoked if *color* has left any node uncolored. If all nodes received colors, the allocator has succeeded. If needed, *insert spills* converts the single live range being spilled into a collection of tiny live ranges, one around each use or definition of the value.

More detailed descriptions of these processes can be found in Chaitin’s work and our own SIGPLAN 89 paper [CACC 81, Chai 82, BCKT 89].

3. Live Range Splitting

In Chaitin’s scheme, a live range that cannot be colored gets spilled everywhere. Chow proposed a simple alternative. He observed that splitting a live range into several pieces and considering these new live ranges separately can produce an interference graph that colors with less spilling [ChHe 90]. Thus, when his allocator cannot assign a color to some live range l_i , it splits l_i into smaller live ranges, one for each basic block in which l_i appears. This technique is called *live range splitting*.

To decrease the amount of fragmentation introduced by splitting, Chow also included a method for combining some of these small live ranges. After splitting a live range, the allocator examines the resulting set of smaller live ranges. If it finds two adjacent live ranges that would have $degree < k$ when combined, it pastes them together.

Live range splitting has several merits. The splitting process often creates live ranges of lower degree; the limitation on combining keeps degrees low. If an entire live range is spilled, as in Chaitin’s work, its value will reside in a register only for trivial periods around each definition or use. Splitting allows the live range to stay in a register over longer intervals – often an entire block or, if combinations are possible, over several blocks. With luck, the new live range can be large enough to extend over all of an important construct, like an inner loop.

As proposed by Chow, however, this idea has several shortcomings. Because it only splits a live range after the allocator has failed to assign the live range a color, it ends up using Chow’s priority-based coloring heuristic to select the live ranges that get split. Because it splits at each basic block boundary and then combines some of these small live ranges, it doesn’t always select good split points. Our technique, described in the next section, attacks both of these problems with a simple but powerful set of heuristics.

4. Aggressive Live Range Splitting

In an allocator that tries to use live range splitting, the implementor faces two fundamental problems:

- (1) picking live ranges to split, and
- (2) picking places to split them.

It is clear that the allocator could spend exponential time picking the optimal set of live ranges to split and the optimal points at which to split them. Our quest has been for reasonably efficient techniques that provide good answers to these two questions. We have developed, implemented, and tested heuristic techniques to address each of these problems. This section describes our approach. The first subsection discusses our aggressive approach to splitting. The second subsection describes the two mechanisms that we use to moderate the effects of aggressive splitting. The third subsection deals with spilling and the computation of spill costs. The fourth subsection relates some of our experience with the implementation. The final subsection gives some data on the behavior of the new allocator.

⁴ The metric for picking spill candidates is important. For the purposes of this paper, assume that the allocator uses a single metric, Chaitin’s metric of spill cost divided by current degree [Chai 82]. The spill cost is computed as the number of loads and stores that would be required to spill the live range, with each operation weighted by 8^d where d is the instruction’s loop nesting depth.

4.1. Splitting

In our search for a splitting technique that produced good results with a reasonable running time, we were forced to reconsider the fundamental ideas of the coloring paradigm for register allocation. The key insight behind our work is that *the interference graph captures none of the structure of the control flow graph*. In translating the allocation problem into a coloring problem, the compiler loses almost all information about the topography of the code. There is no representation for locality. Estimates of execution frequency get factored into estimated spill costs, but because the information is computed over the whole procedure, it gives equal weight to both near and distant references. Thus, a live range that is heavily used in some critical inner loop may get spilled in deference to a value that is live across the loop and used in one or more distant but deeply nested loops.

To recapture geographic locality, we advocate (1) finding those points in the code where we would like splitting to occur, and (2) splitting *every* live range at those points. Thus, we avoid the problem of picking a live range by splitting all the live ranges that cross a split point. To locate split points, we use a simple and efficient examination of the control-flow graph.

Selection of good split points is critical. Relying on our fundamental insight, we return to the geography of the procedure. Conceptually, we would like to spill before and after each loop. In practice, we find the strongly connected regions in the control flow graph. For those regions where register pressure is high, we insert split points along all entering and exiting edges.⁵

Accomplishing the splits is simple. The allocator breaks each live range at the split point and inserts a copy from the incoming live range to the outgoing live range. Then it runs *renumber*, which naturally renames the two new live ranges.

This strategy has two major effects.

- (1) First, it shifts the basis of the spill competition from global allocation towards local allocation. In an innermost loop, the competition is effectively local. Moving out through the loop nest, the competition shifts from local allocation toward more global allocation.
- (2) Second, because most spill code is generated at the split points, it gives us a great deal of control over spill code placement. Many live ranges will spill only over inactive regions. These spills require only a single load or load/store pair at the endpoints. The strategy encourages this spilling of inactive regions – once the first inactive region is spilled, adjacent inactive regions have spill costs of zero.

4.2. Coalescing

Our technique breaks every live range at every split point. A more selective strategy might be able to generate the same allocation while generating fewer splits. Rather than spend time being selective, we choose to split aggressively and let the allocator recombine some of the extraneous splits. Of course, the design must maintain a delicate balance to avoid negating the benefits achieved through splitting. We use two techniques to maintain this balance: limited coalescing and color biasing. Taken together, these two methods approach the effectiveness of the complete coalescing performed in a Chaitin-style allocator without undoing the benefits of splitting.

4.2.1. Limited Coalescing

Essentially, coalescing is the inverse of splitting. Coalescing merges disjoint live ranges (when that operation is legal). Splitting takes a single live range and creates disjoint live ranges. Given an interference graph G , if we applied aggressive splitting followed by the normal coalescing, we would expect to get back

⁵ Originally, we considered splitting around each strongly connected region. Matt Zaleski of IBM pointed out that splitting in areas of low pressure is both unnecessary (it does not improve the allocation) and counterproductive (it increases the size of the graph). We call this improvement *Zaleski's finesse*.

the same interference graph (within an isomorphism over register names). Therefore, we apply a limited form of coalescing.

In normal coalescing, two live ranges are combined if the initial definition of the second is a copy from the first and they do not otherwise interfere. In limited coalescing, we add an additional constraint: only coalesce two live ranges if the resulting live range has “low degree”. Intuitively, we want to coalesce two live ranges when the resulting live range is no harder to color than the original live ranges.

Initially, we considered the following notion: two live ranges are coalesced only when the resulting live range would have degree less than k . While this approach cannot introduce new spills, it is probably too conservative. It will not coalesce any live ranges that are not trivially colored – that is, live ranges with initial degree $> k$. In practice, many such live ranges receive colors.

This observation led us to refine our restriction.⁶ Two live ranges l_i and l_j are combined only when $l_{ij}^o \leq \max(l_i^o, l_j^o, k - 1)$. This formulation allows the allocator to coalesce live ranges that are not trivially colored. The increased power to coalesce does have a downside. For example, if l_i would spill and l_j would not, and $l_{ij}^o = l_i^o$ then the second technique would coalesce them, with the result that l_j is spilled.

Limited coalescing has no deleterious effects on neighboring live ranges. Assume that we have two live ranges l_i and l_j that interfere. Coalescing l_i with another live range l_k cannot raise l_j^o . If l_j and l_k do not interfere, then l_j^o is unchanged. Furthermore, if l_j interferes with both l_i and l_k , then combining l_i and l_k reduces l_j^o by one.

Both of these strategies shrink the number of live ranges, reduce the number of copies, undo some of the excess splitting, and help lower the overall degree of the graph. By placing restrictions on the degree of the resulting live range, we ensure that limited coalescing will not create new spills. Thus, limited coalescing pastes live ranges back together in a conservative fashion, undoing splitting in some of the cases where it did not improve the allocation. However, limited coalescing cannot undo all of the non-productive splitting introduced by our aggressive strategy.

4.2.2. Biased Coloring

Our second mechanism for pasting live ranges back together involves biasing the order in which colors are considered for assignment. As live ranges are split, the allocator constructs a list of *partners*, new live ranges that are all split off from a single original live range. When the allocator assigns a color to l_i , it first tries colors already assigned to one of l_i ’s partners. With a careful implementation, this is no more expensive than picking the first available color; it really amounts to biasing the spectrum of colors by previous assignments to l_i ’s partners.

The biasing mechanism can combine live ranges that limited coalescing cannot. In particular, it can combine two live ranges that receive colors when treated separately, but would likely be spilled if combined during limited coalescing. By virtue of the time of its application – after the allocator has decided that both live ranges will receive colors – it takes advantage of deep knowledge about the interference graph.

For example, at the time that limited coalescing is performed, the allocator has no knowledge about interferences between the neighbors of some live range l_i . Thus, limited coalescing must rely on l_i^o as a crude approximation to l_i ’s colorability. It may be the case that l_i has $2k$ neighbors, but that they require only three colors between them because few of them interfere with each other. The high degree will prevent limited coalescing from combining l_i with one of its partners l_j , particularly if l_j ’s neighbors are disjoint from l_i ’s neighbors. The biasing mechanism, by virtue of its late application, is only invoked when both live ranges will be assigned colors. This detailed level of knowledge is not available earlier in the process – for example, when coalescing is performed.

⁶ To simplify the text, we will use l_i^o to denote $\text{degree}(l_i)$. Similarly, l_{ij} denotes the combination of l_i and l_j .

4.3. Spilling

When a live range is split, the component live ranges are called *partners*. Partners are attached to one another via copies at each split point. Since each set of partners is split from one live range, members of the same set can spill to the same location. The recognition and proper handling of partners is essential for obtaining high-quality spill code.

Consider the example in Figure 2. The single live range in (a) is split in (b) by the introduction of a copy. The resulting live ranges, l_j and l_k , are partners. If l_j is spilled, we should get (c). Alternatively, (d) illustrates the result of spilling l_k . Note that each partner spills to the same location. Finally, (e) shows the result of spilling both partners.

Now consider the costs for the sequence from (b) through (c) to (e). Moving from (b) to (c) costs one store and one load, but saves one copy.⁷ The transition from (c) to (e) saves one load (at the split point) and costs one load (at the use point). No new instructions are required; instead, the load is effectively moved. Therefore, the cost of spilling l_k at (c) is determined by the relative loop nesting depth of the split point and the use. If the split point is nested more deeply than the use, it will be *profitable* to spill l_k .

We account for these situations while computing spill costs and inserting spill code. Additionally, we update spill costs incrementally while simplifying and coloring. In terms of the example in Figure 2, if l_j cannot be colored and must be spilled, the cost of spilling its partner is immediately adjusted, increasing the probability of spilling l_k . Further, if any live range has a negative spill cost, it will be spilled immediately and its partners' costs updated appropriately.

The effect of this careful handling of partners is important. Aggressive splitting divides long live ranges into long chains of partners. If one partner is spilled, it tends to drag its immediate partners along.

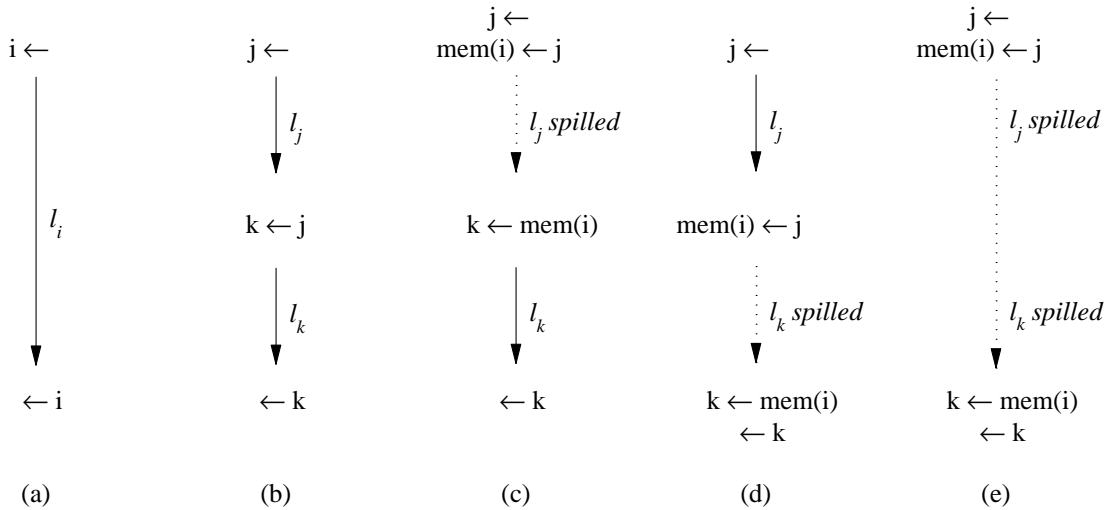


Figure 2 — spilling partners

⁷ Naturally, each instruction's cost should be weighted according to its loop nesting depth.

Conversely, when a partner is kept in a register, it tends to hold its immediate partners in registers. These tendencies, together with register pressure from competing live ranges, work to force spill points (both loads and stores) out of loop nests.

4.4. Implementation

Integrating these ideas into our allocator was a major task. Conceptually, the task is fairly straightforward; Figure 3 shows a high-level view of the resulting allocator. Several points have changed from the allocator depicted in Figure 1.

- (1) *Split* finds strongly connected regions in the procedure’s control flow graph and splits live ranges along each edge that enters a region. This may introduce some new basic blocks to hold the copies that break the live ranges.
- (2) *Limited coalesce* combines live ranges using one of the strategies described in section 4.2.1. In effect, this phase undoes some splits that are unproductive.
- (3) *Biased color* is almost identical to the *color* phase of the original allocator. The biased method uses a dynamically determined ordering to choose colors rather than a static ordering. The dynamic ordering gives preference to colors already assigned to one of the live range’s partners.

Unfortunately, Figure 3 is too pretty; reality is somewhat uglier. Figure 4 shows the allocator as actually implemented.

We have peeled off the first iteration of the allocate-spill loop. In the first iteration, *coalesce* and *color* are the originals shown in Figure 1. Any procedure that generates no spill code avoids the expense of splitting; it will take the exit after *color* (called “Markstein’s exit”). If the first iteration must generate spill code, control passes on to *split* and subsequent iterations are handled by the modified allocator. *Limited coalesce* and *biased color* implement the coalescing mechanisms described in Section 4.2.

Finally, Figure 4 includes an edge that was omitted in both earlier figures. The loop from *biased color* back to *simplify graph* shows that the allocator implements the “best of three spill choice” computation suggested by Bernstein *et al.* [BGGK 89]. They observed that the cost of simplifying and coloring is much less than the cost of building the graph. They suggested trying three different spill choice heuristics and using the result that produces the lowest spill cost.

The initial iteration has one other effect. Because it uses the full strength *coalesce* from the original allocator, and repeats the *build graph-coalesce* loop until the graph stabilizes, it gets rid of any extraneous copy operations introduced as a by-product of other optimizations. Thus, before the splitter is ever invoked, the allocator will reduce the number of live ranges to some canonical set. In this reduced graph, any remaining copy instructions are meaningful.

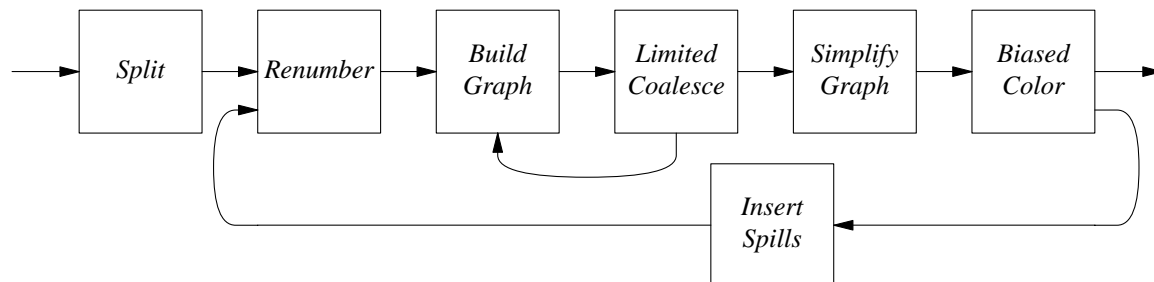


Figure 3 — the splitting allocator: conceptual view

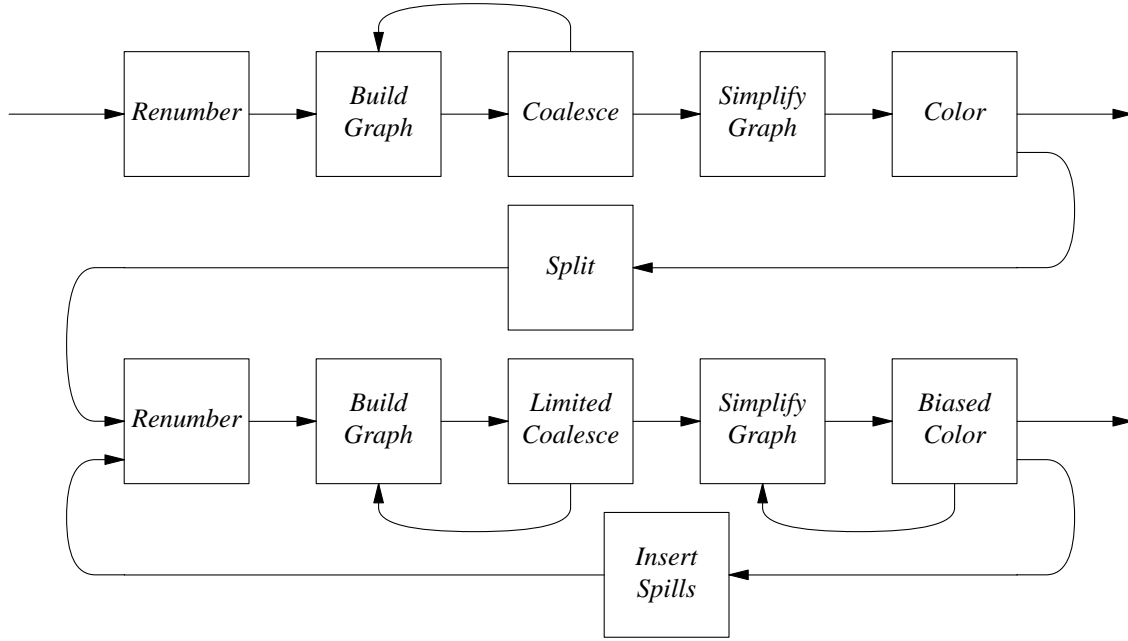


Figure 4 — the splitting allocator: realistic view

As we built the splitting allocator, we encountered one additional opportunity for improvement. Because splitting increases the number of live ranges (see Section 4.5), the number of candidates that must be examined on any spill decision grows. Our original allocator simply searched through the remaining nodes; this did not cause significant performance problems. In the splitting allocator, it proved to be excessively slow.

Of course, using a more complex data structure to order the spill candidates can decrease the asymptotic cost of this operation. Two possibilities are interesting:

- (1) a single heap for all nodes, or
- (2) a collection of heaps, each containing all nodes of a given degree.

The former scheme is simpler, in that the allocator need only manage a single heap. The latter scheme is attractive because it decreases the number of division operations required – nodes of the same degree can be compared without division.

From a performance perspective, the key issue is clearly updating the heaps. As each node is removed from the graph, it must be removed from its heap and the degrees of all its neighbors updated. In both schemes, removing the spill candidate is inexpensive – $O(\log_2 n)$ where n is the number of nodes in the heap from which it is removed.

The cost for updating the degrees of its neighbors is radically different in the two schemes.

- (1) With a single heap, the weight, or key, of each neighbor must be adjusted. If the magnitude of the change is small relative to the surrounding keys, this should involve a couple of comparisons and a pointer swap. In the worst case, it requires $O(\log_2 n)$ operations.
- (2) With a distinct heap for each degree, each neighbor must be removed from its heap and inserted into the heap of next lower degree. Each of these operations costs $O(\log_2 n)$ operations, where n is the number of nodes in the involved heap.

In computing the key for each node, the degree is used in the divisor. Nodes in the heap must have degree of k or greater. Thus, we expect the change in the key's magnitude to be small.⁸ We are implementing the single heap solution in the allocator; we expect it to be fast in practice.

4.5. Experimental Results

We have implemented a prototype splitting allocator. It builds on our earlier work for the IBM RT/PC [BCKT 89]. The new allocator has a variety of options, allowing us to experiment with various heuristics for splitting and coalescing. While we have only conducted a limited set of experiments to date, the results are gratifying.

Our best result was a 27 percent reduction in estimated spill costs. This improvement was measured against our best previous efforts on the FORTRAN routine, SVD.⁹ It has 37 DO-loops, organized into five loop-nests. As a result, it is ideally suited to our loop-based splitting techniques.

Obviously, our scheme increases the number of live ranges, which increases the number of nodes and edges in the interference graph. In our experiments, the number of nodes grew by a factor of 1.5 to 5. The number of edges in the graph increased by a factor of 3 to 10. For SVD, the number of live ranges grew from 705 to 3076, a factor of 4.4, and the number of edges grew from over 38,000 to over 367,000, a factor of 9.5. The space requirements for the graph grew by a factor of 10, to just over one megabyte. We feel that one megabyte for the graph is reasonable. While the growth on SVD is high, it is important to remember that the structure of SVD provokes a large amount of splitting.

We are constructing a new version of our allocator as part of a compiler for the Intel i860. The complete paper will include detailed measurements (including compile-times, run-times, and graph sizes) from a broad range of test cases.

5. Other Work

Other researchers have noted weaknesses in Chaitin's register allocator and have developed new approaches to the allocation problem. Since a complete survey of the many different approaches would be difficult, we offer a limited comparison of related work. It seems helpful to characterize the various approaches based on the amount of structural information derived from the control flow graph and employed during allocation.

- The earliest work on graph coloring register allocation emphasizes the coloring problem with little consideration for the questions of spill choice and placement. Algorithms by Cocke and Ershov (as reported by Schwartz [Schw 73]) are concerned exclusively with minimizing the number of colors required. There is no discussion of spill code and the flow graph is ignored entirely.
- The first complete register allocator based upon graph coloring is described in [CACC 81]. Spilling is handled by a variety of heuristics, some based upon an interval analysis of the flow graph. Unfortunately, these *ad hoc* techniques are expensive and not always effective. In a subsequent paper, Chaitin introduces a simpler technique that attempts to solve the spilling problem based on the interference graph and spill cost estimates for each live range [Chai 82]. The cost estimates are based on the loop structure of the flow graph; but spill code is introduced without regard for any control structure. Our 1989 paper describes an improved coloring heuristic, but is otherwise identical to Chaitin's 1982 technique [BCKT 89].
- The *cleaning* heuristic introduced by [BGGK 89] is an attempt to avoid problems arising from the very conservative spill techniques employed by Chaitin. In this case, basic blocks are recognized while

⁸ In Chaitin's original spill choice heuristic, the key has *degree* in the divisor [Chai 82]. Of the three heuristics proposed by Bernstein *et. al.*, one has *degree* in the divisor, while two have *degree*² in the divisor [BGGK 89]. For these latter two spill choice heuristics, the variation in magnitude will be larger. We hope that further measurements with the allocator will show whether this increases the average number of comparisons required to reposition the node after a recomputation of its key.

⁹ SVD is a version of the singular value decomposition from Forsythe, Malcolm, and Moler [FoMM 77].

spilling, but no distinction is made for higher level constructs.

- The notion of live range splitting was introduced in Chow and Hennessy's 1984 paper [ChHe 90]. Loop nesting is used to help prioritize coloring and basic block boundaries are used to establish split points. Larus and Hilfinger refined this approach to extend live ranges after splitting using a breadth-first traversal of the blocks comprising the old live range [LaHi 86]. Gupta, Soffa, and Steele describe a variation of Chow's approach that partitions the interference graph into subgraphs that are colored individually and later merged [GuSS 89]. While the partitioning is performed by examining the code, no particular attention is paid to the structure of the flow graph.
- Meltzer describes an interesting approach to global register allocation that is *not* based on coloring [Melt 89]. He performs a detailed analysis of the flow graph, constructing a control tree containing a hierarchical description of the control constructs used in the graph [Shar 80]. Register allocation is performed in two passes over the tree. Callahan and Koblenz take a similar tack [CaKo 91]. They construct a fine-grained hierarchical decomposition of the flow graph, a *tiling*. Coloring is performed for individual tiles and the results are merged in two passes over the tree.

Our approach fits in the middle. We perform structural analysis of the control flow graph to find the natural spill points – on the edges leading in and out of high-pressure loops. We split all live ranges at these potential spill points and then color a global interference graph. While coloring, we remove splits when profitable via conservative coalescing and biased coloring. Spill costs are incrementally adjusted while coloring, allowing accurate placement of spill code.

6. Acknowledgements

Greg Chaitin, Ben Chase, John Cocke, Marty Hopkins, Bob Hood, Ken Kennedy, Peter Markstein, Tom Murtagh, Randy Scarborough, Rick Simpson, Tom Spillman and Matt Zaleski have all contributed to this work through encouragement and enlightened discussion. Our colleagues on the ParaScope project at Rice have provided us with an excellent testbed for our ideas. To all these people go our heartfelt thanks.

7. References

- [BGK 89] D. Bernstein, D. Goldin, M. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, R. Pinter. "Spill code minimization techniques for optimizing compilers," *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 24(7), July, 1989.
- [BCKT 89] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon. "Coloring heuristics for register allocation," *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 24(7), July, 1989.
- [CaCK 90] D. Callahan, S. Carr, and K. Kennedy. "Improving register allocation for subscripted variables," *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 25(6), June, 1990.
- [CaKo 91] D. Callahan and B. Koblenz. "Register allocation via tiling," *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation (to appear)*, SIGPLAN Notices 26(6), June, 1991.
- [Chai 82] G.J. Chaitin. "Register allocation and spilling via graph coloring," *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction*, SIGPLAN Notices 17(6), June, 1982.
- [CACC 81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. "Register allocation via coloring," *Computer Languages* 6, January, 1981.
- [ChHe 90] F. Chow and J. Hennessy. "The priority-based coloring approach to register allocation," *ACM TOPLAS* 12(4), October, 1990.
- [CoHT 90] K.D. Cooper, M. Hall, and L. Torczon. "An experiment with inline substitution," *Software-Practice and Experience*, to appear. Also available as *Computer Science Technical Report 90-128*, Rice University, August, 1990.

- [Ersh 62] A.P. Ershov. “Reducing the problem of memory allocation when compiling programs to one of coloring the vertices of graphs,” *Doklady Akademii Nauk S.S.S.R.* 142(4), 1962; English translation in *Soviet Math* 3, 1962.
- [Ersh 66] A.P. Yershov.¹⁰ “ALPHA – an automatic programming system of high efficiency,” *Journal of the ACM* 13(1), January, 1966.
- [FoMM 77] G.E. Forsythe, M.A. Malcolm, and C.B. Moler. *Computer Methods for Mathematical Computations*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.
- [GuSS 89] R. Gupta, M.L. Soffa, T. Steele. “Register allocation via clique separators,” *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 24(7), July, 1989.
- [Kenn 71] K. Kennedy. “Global flow analysis and register allocation for simple code structures,” Ph.D. Thesis, Courant Institute, New York University, October, 1971.
- [LaHi 86] J.R. Larus and P.N. Hilfinger. “Register allocation in the SPUR Lisp compiler,” *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices* 21(7), June, 1986.
- [Lavr 61] S.S. Lavrov, “Store Economy in Closed Operator Schemes,” *Journal of Computational Mathematics and Mathematical Physics*, 1(4), 1961, pp. 687-701. (English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3, 1962)
- [Melt 89] A. Meltzer. “Control tree based register allocation,” *unpublished abstract*, COMPASS, 1989. (a full version is in preparation).
- [Schw 73] J.T. Schwartz. “On programming: An interim report on the SETL project. Installment II: The SETL language and examples of its use,” Computer Science Department, Courant Institute of Mathematical Sciences, October, 1973.
- [Shar 80] M. Sharir. “Structural analysis: a new approach to flow analysis in optimizing compilers,” *Computer Languages* 5, 1980.

¹⁰ There appear to be two accepted spellings for Ershov’s name. We use the more modern spelling for references, but spell the citations as they originally appeared.