Live Range Splitting in a Graph Coloring Register Allocator*

Keith D. Cooper[1] and L. Taylor Simpson[2]

[1] Rice University, Houston, Texas, USA
[2] Trilogy Development Group, Austin, Texas, USA

Abstract. Graph coloring is the dominant paradigm for global register allocation [8, 7, 4]. Coloring allocators use an interference graph, $\mathcal{I}$, to model conflicts that prevent two values from sharing a register. Nodes in $\mathcal{I}$ represent live ranges. An edge between two nodes indicates that they are simultaneously live and, thus, cannot share a register. The allocator tries to construct a k-coloring of $\mathcal{I}$, for k equal to the number of registers on the target machine. If it succeeds, it maps the colors onto the machine registers to produce an allocation. Unfortunately, it may not discover a k-coloring. In that case, it *spills* some live ranges by saving their values to memory. Early coloring allocators spilled live ranges completely – at each definition and use. This approach often introduces more spill code than necessary. This paper presents a global approach that avoids some spill code by *splitting* the live range, breaking it into smaller pieces.

We are not the first to study this problem. Bergner *et al.* describe a heuristic called *interference region spilling* that reduces the amount of code needed to spill a live range [1]. Briggs experimented with an aggressive form of live range splitting; he saw mixed results [3, see Chapter 6]. This paper presents a passive form of live range splitting that uses splitting as an alternative to spilling. The allocator finds regions where splitting breaks the interferences that cause a spill; it uses estimated costs to choose between splitting the live range and spilling it. We present experimental evidence that this technique is effective. We have seen reductions in the amount of dynamic spill overhead as high as 78% for non-trivial routines. Our technique can easily be coupled with Bergner's to create an allocator that makes a cost-based choice between splitting, interference region spilling, and spilling completely for each spilled live range.

1 Introduction

Chaitin *et al.* first used graph coloring as a paradigm for register allocation and assignment in a compiler [8, 7]. Coloring allocators approach the problem by building a graph that models when two live ranges cannot reside in the same location; we call the graph an *interference graph* ($\mathcal{I}$) because the live ranges would interfere with each other if they shared a register. The allocator attempts

* This work was supported by Darpa through contract Dabt63-95-C-0115 and by the Trilogy Development Group.

to discover a k-coloring of $\mathcal{I}$; that is, an assignment of k colors to the nodes of $\mathcal{I}$ in such a way that no adjacent nodes have the same color. If it can find a k-coloring, for k equal to the number of registers on the target machine, it can map the colors into registers and its task is done. If, however, it cannot discover such a coloring, it selects one or more live ranges to store in memory, or *spill*. It inserts code to spill those live ranges and tries to color the interference graph for the resulting, modified procedure.

Chaitin's basic scheme has been improved by other authors. Briggs *et al.* describe variations on the coloring heuristic that increase the number of live ranges that can be colored [4, 5]. Bernstein *et al.* showed that different heuristics for choosing spill candidates can improve the results [2]. These modifications have reduced the cost of spilling, but they have not eliminated it. The remaining problem is not a poor coloring heuristic; these routines have regions where they need more registers than the hardware provides.

Once it chooses a live range to spill, Chaitin's allocator spills that value everywhere. It places a STORE instruction after each definition of the value and a LOAD instruction before each use of the value. Local heuristics can reduce the number of spill instructions inserted into a single basic block [7, 2]. These methods eliminate some redundant spill instructions in a block that contains several references to the spilled value. They do nothing for problems that arise across multiple blocks.

Bergner *et al.* introduced a global technique for reducing spill code that they called interference region spilling [1]. Rather than spill the live range everywhere, their method chooses a color for it and only spills it in areas where that color is unavailable. The allocator picks a color for the spilled live range by estimating the costs that would be incurred for each color; it selects the color with the smallest estimated cost.

In this paper, we present another global method for reducing spill code, called *live range splitting*. Chaitin-style allocators use maximal-length live ranges as the basic unit of allocation. It has long been recognized that breaking a live range into smaller pieces may allow some, or all, of the resulting pieces to be colored [13, 9]. Chow used this observation in his priority-based coloring scheme; when his allocator encountered a live range that could not be kept in a register, it broke it into smaller pieces. Briggs experimented with an aggressive form of live range splitting in his Chaitin-style allocator [3, Chapter 6]. His algorithm aggressive split many long live ranges before it tried to color the graph; he added several mechanisms to the allocator that could recombine the smaller pieces when doing so would not cause a spill. The method produced both large gains and large losses; sometimes, it aggressively inserted splits that were both unneeded and beyond its power to remove.

Our approach overcomes this difficulty by being passive. It only considers splitting a live range l after l has been selected for spilling. To split l, it looks for a color where splitting will succeed – that is, either all live ranges of that color can be split around l, or l can be split around all live ranges of that color. If such a color exists, and the estimated cost of splitting is less than the cost

of spilling l, our method will split rather than spill. Because it compares the estimated cost of splitting and spilling, this method can easily be combined with Bergner's method. The resulting allocator would use cost estimates to choose between live range splitting, interference region spilling, and complete spilling for each live range that must be spilled.

1.1 Example

To understand the benefits of live range splitting, consider the code on the left side of Figure 1. If only one register is available, then the allocator must spill one of l_1 or l_2. The "spill everywhere" method would place spill code inside one of the loops. Assume that the spilling heuristic chose l_2; the middle column shows the result of spilling l_2 entirely. Notice that a LOAD instruction gets inserted into the loop. A second problem with this choice is that the two small live ranges that result from spilling still interfere with l_1. The next round of spilling will still need to address the underlying problem; in a spill-everywhere scheme, l_1 will be spilled. Splitting l_1 across l_2, shown in the right column, produces a much better result. All the spills occur outside the loop. To split l_1 across l_2, we insert a STORE for l_1 before each definition of l_2, and a LOAD of l_1 after each death of l_2.

Normally, a live range dies after its last use. The exception occurs in the presence of control-flow – the flow may branch to one path where a value is live and to another path where the value is dead. Intuitively, the death occurs along the second edge. In our example, l_2 dies along the edge that exits the first loop, so we insert a LOAD for l_1 in the successor block.

Splitting in this way lets us allocate l_1 and l_2 to the same register, without inserting spill code inside the loop. In the example, l_1 is split across l_2 because

Fig. 1. Example of live range splitting

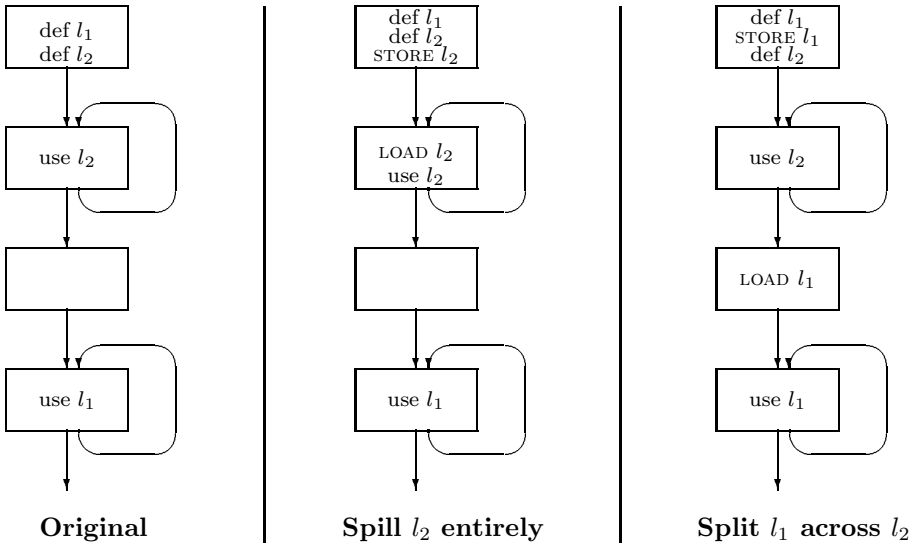
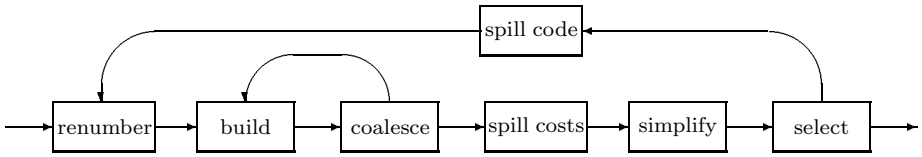


Fig. 2. Briggs' allocator



l_1 completely contains l_2. To let the allocator detect this situation, we introduce a new data structure, the *containment graph*. Section 3 describes this graph in detail. Section 4 explains how to insert the code that splits a live range. Sections 5 and 6 put the algorithm together. Section 7 presents some experimental results showing the efficacy of splitting.

2 Briggs' Allocator

Because our live range splitting procedure extends a Briggs-style allocator, we will begin with an overview of that allocator. Figure 2 shows a flow chart of a Briggs-style allocator [4]. It is composed of seven major phases:

Renumber The symbolic, or virtual, registers in the routine are renamed to create live ranges. A live range is a collection of definitions that reach a common use. Briggs accomplishes this by converting the routine to pruned static single assignment form [12] and then combining all names mentioned in each ϕ-node.

Build The interference graph, $\mathcal{I}$, contains a node for each live range and an edge between each pair of live ranges that are simultaneously live. $\mathcal{I}$ is represented by both a triangular bit matrix and a collection of adjacency lists. We build $\mathcal{I}$ by traversing the instructions in the routine; at each definition, we add an edge between the defined name and all live ranges that are currently live.

Coalesce If the source and destination of a copy do not otherwise interfere, the two live ranges can be assigned the same register, and the copy can be removed. When two live ranges are coalesced, we add an edge between the new live range and each neighbor of the two live ranges. This approach may be overly conservative, so we repeat the *build* and *coalesce* phases until no more coalescing is possible.

Spill costs We estimate the cost of spilling each live range by counting the instructions (weighted by instruction cost and by loop nesting depth) required to spill that live range. The effect of any local heuristic to reduce the number of LOADs is included in the estimated spill cost for each live range.

Simplify Coloring is a two step process. During the first phase, we repeatedly remove a node with degree less than k from $\mathcal{I}$, and push it onto the coloring stack. If the process reaches a point where no such node exists, a live range

is chosen heuristically to be a spill candidate. *Simplify* pushes the spill candidate onto the stack, and optimistically hopes that it can receive a color during the *select* phase.[3]

Select We repeatedly pop a live range from the coloring stack, insert it back into the interference graph, and assign it a color different from those of its neighbors. If no color is available, the live range is left uncolored and marked for spilling. An uncolored live range is always one of the spill candidates chosen by *simplify* [4, 5]. If we are able to assign a color to every live range, this corresponds to a valid allocation, and the algorithm terminates.

Spill code If *select* marked any live ranges for spilling, we must update the code to keep these values in memory and repeat the entire allocation process. This phase traverses the instructions in the same manner as *spill costs*, inserting the actual LOADs and STOREs for spilled live ranges.

3 The Containment Graph

Chaitin first observed that spilling a live range does not break all its interferences; this is one reason that the allocator must repeat the coloring process after spill code is inserted. Spilling merely breaks a live range into multiple tiny live ranges. These tiny live ranges still interfere with values that are live across them. The middle column of Figure 1 shows this situation. The allocator has spilled l_2 entirely, producing two short live ranges – one at l_2's definition and one at its use. Both these live ranges still interfere with l_1, because l_1 is live across each of them.[4] If, as shown in the right column of Figure 1, the allocator spills l_1 instead of l_2, the new live ranges do not interfere with l_2 because l_2 is not live across either of them. Spilling l_1 breaks the interference between l_1 and l_2, where spilling l_2 does not.

In general, spilling a live range l_i does not break its interference with any live range l_j that is live at either a definition of a use of l_i. If, on the other hand, l_i and l_j interfere, but l_j is not live at any definition or use of l_i, then spilling l_i breaks the interference between them. In this case, we say that l_i *contains* l_j. The allocator can use this knowledge to split l_i around l_j by storing l_i before definitions of l_j and restoring it after l_j dies. Splitting l_i in this manner eliminates the interference with l_j.

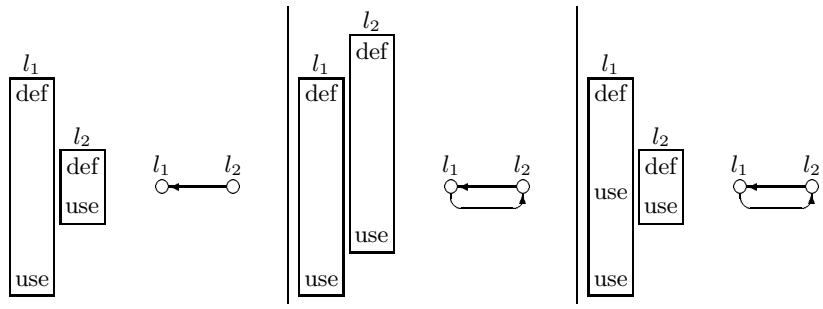
To capture this knowledge, we introduce the containment graph ($\mathcal{C}$), a directed analog of the interference graph ($\mathcal{I}$). Nodes in $\mathcal{C}$ represent live ranges. An edge from l_j to l_i in $\mathcal{C}$ indicates that l_i is live at a definition or use of l_j. We represent $\mathcal{C}$ with a square bit matrix. It is twice as large as the triangular bit matrix used to represent $\mathcal{I}$; our method does not need adjacency lists for $\mathcal{C}$.

Figure 3 shows some examples that illustrate the utility of the containment graph. The left column depicts the situation from Figure 1; l_2 is not live at either

[3] Chaitin's algorithm always spilled the candidate live range. Briggs' method lets *select* see if a color is available.

[4] In fact, since both live ranges created for l_2 interfere with l_1, spilling l_2 actually increases l_1's degree in $\mathcal{I}$.

Fig. 3. Examples of the containment graph



the definition or the use of l_1, so $\langle l_2, l_1 \rangle \in \mathcal{C}$. In the middle column, l_1 and l_2 overlap, so both $\langle l_1, l_2 \rangle \in \mathcal{C}$ and $\langle l_2, l_1 \rangle \in \mathcal{C}$. The right column is similar to the left column except that there is a use of l_1 while l_2 is live. This additional use adds the edge $\langle l_1, l_2 \rangle$ to $\mathcal{C}$.

These examples illustrate how we can use $\mathcal{C}$ for live range splitting. Given two live ranges l_i and l_j, with $\langle l_i, l_j \rangle \in \mathcal{I}$, one of the three cases must occur:

Edges in $\mathcal{C}$	*Impact on spilling*
$\langle l_i, l_j \rangle$	Spilling l_i leaves the interference with l_j. Splitting l_j around l_i eliminates the interference.
$\langle l_j, l_i \rangle$	Spilling l_j leaves the interference with l_i. Splitting l_i around l_j eliminates the interference.
$\langle l_i, l_j \rangle$ & $\langle l_j, l_i \rangle$	Spilling either l_i or l_j leaves the interference between them. Splitting does not help.

The fourth case, where $\langle l_i, l_j \rangle \notin \mathcal{C}$ and $\langle l_j, l_i \rangle \notin \mathcal{C}$, would imply that $\langle l_i, l_j \rangle \notin \mathcal{I}$, which contradicts our premise.

Figure 4 shows the algorithm for building $\mathcal{C}$; it is similar to the algorithm for building the interference graph. For clarity, we describe the algorithm as if it must build both graphs separately and at different times. The implementor might elect to build just the bit-matrix for $\mathcal{C}$; the allocator could consult $\mathcal{C}$ twice to infer the entry in $\mathcal{I}$.

The primary drawback to using $\mathcal{C}$ is the space required to hold the bit-matrix. Two different facts should moderate this problem.

1. $\mathcal{C}$ contains all the information found in $\mathcal{I}$. Thus, we do not need the lower-triangular bit-matrix form of $\mathcal{I}$. If the edge $\langle l_i, l_j \rangle \in \mathcal{I}$, then one or both of $\langle l_i, l_j \rangle$ and $\langle l_j, l_i \rangle$ must be in $\mathcal{C}$.
2. A bit matrix may be space inefficient for $\mathcal{C}$. A recent study of techniques for building interference graphs showed that a closed hash table implementation can use less space for sufficiently large graphs [11]. $\mathcal{C}$ should reach that threshold in half the time.

These suggestions should reduce the space impact of building $\mathcal{C}$ rather than $\mathcal{I}$.

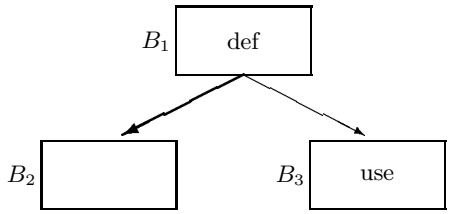
Fig. 4. Building the containment graph

```
buildContainmentGraph()
    Allocate the square bit matrix
    For each block b
        live ← liveOut_b
        For each instruction, i, in b in reverse order
            For each live range, l, defined in i
                For each m ∈ live
                    Add edge ⟨l, m⟩ to C
            Update the live set
            For each live range, l, used in i
                For each m ∈ live
                    Add edge ⟨l, m⟩ to C
```

4 Computing Split Costs and Inserting Split Code

The containment graph tells the allocator when it is *possible* to split one live range across another. The next step is to determine when this splitting is *profitable*. Estimating the cost of splitting is similar to estimating spill costs. We compute the number of LOAD and STORE instructions required to split across each live range. Spilling a live range requires a STORE before each definition and a LOAD after each death. Definitions are easy to identify; deaths require a bit more effort. We can traverse the instructions in each block in reverse order and follow the effect that each instruction has on the *live* set. Initially, the *live* set is the *liveOut* set for the block. At each instruction, we remove any defined live ranges and add any used live ranges. When a live range, l, is added to the set for the first time, we have identified a death of l. Deaths can also occur at branch points in the control-flow graph. The example in Figure 5 illustrates how this can happen. The live range is defined in block B_1 and used in B_3. Clearly, the use in B_3 is a death, but the value also dies if flow of control transfers from B_1 to B_2. In this situation, we think of the death as occurring along the edge. Formally, the set of live ranges that die along an edge $\langle i, j \rangle$ is $liveOut_i - liveIn_j$.

Fig. 5. Death along an edge in the CFG



The algorithm is shown in Figure 6. The *range* array keeps an estimate of the number of LOADs and STOREs required to split around each live range. The estimates are weighted by nesting depth. When we choose a color to split around the live range, we multiply these estimates by the cost of each instruction.

Once we have selected which live ranges to split (see Section 5), we must insert the necessary LOAD and STORE instructions. The routine to insert the split code follows exactly the same logic as the cost calculation, except that it inserts the code for any live ranges marked for splitting. Whenever we encounter a death of a live range, l, we insert a LOAD for any live range that is split around l. Similarly, when we encounter a definition of l, we insert a STORE instruction for any live range that is split around l.

5 Finding a Color

The previous sections explained how we determine if one live range can be split around another and how we estimate the cost of splitting around each live range. When a live range, l, is chosen for spilling during the *select* phase, we attempt to split one or more live ranges across l. The goal is to find a color which can be made available to hold l. We group all the neighbors of l by color and look for a color such that all the neighbors can be split across l. We total the cost of splitting each neighbor.[5] We also check for a color where l can be split across all those neighbors. If a color is found whose split cost is less than the cost of spilling l entirely, we assign l that color and record which live ranges will be split around l (or which live ranges to split l around).

Figure 7 shows the algorithm used to find a color. The *findColor* routine will be called from *select* whenever a live range, l, is chosen for spilling. We look for a color to assign l by splitting. First we try to split the color around l, then we try to split l around the color. At each point, we keep track of the color with the smallest estimated cost. If a color is found for l, we assign it to $colors[l]$ so that other neighbors of l colored later will not receive that color.

To see how this process works, consider the example in Figure 1. First, assume that l_1 is removed from the stack and assigned a color, c. When l_2 is removed from the stack, it cannot receive a color, so we search for a color to split around l_2. The color c is assigned to neighbor l_1, and there is no edge $\langle l_1, l_2 \rangle \in \mathcal{C}$, so splitting is possible. Since the cost of the split is less than the cost of spilling l_2 entirely, we choose color c for l_2.

In the alternative scenario, l_2 is removed from the stack before l_1. When l_1 is removed from the stack, it cannot receive a color so we search for color to split around l_1 *and* for a color to split l_1 around. We will discover that l_1 can be split around the color of l_2. In other words, our algorithm will split l_1 around l_2 regardless of which live range is assigned a color first.

[5] For normal live ranges, this is the cost of a STORE instruction before each definition and a LOAD instruction after each death. However, if a live range is rematerializable [5], we need only restore its value after each death.

Fig. 6. Computing split costs and inserting split code

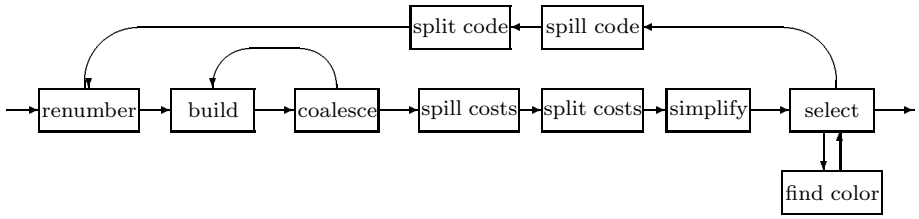
```
splitCosts()
    buildContainmentGraph()
    For each block b
        weight ← 10^depth(b)
        live ← liveOut_b
        For each successor, s, of b
            deaths ← liveOut_b − liveIn_s
            For each m ∈ deaths
                range[m].loads ← range[m].loads + 10^depth(s)
        For each instruction, i, in b in reverse order
            For each live range, l, defined in i
                range[l].stores ← range[l].stores + weight
            For each live range, l, used in i
                if l ∉ live
                    range[l].loads ← range[l].loads + weight
            Update the live set

splitCode()
    For each block b
        live ← liveOut_b
        For each successor, s, of b
            deaths ← liveOut_b − liveIn_s
            For each m ∈ deaths
                For each live range, l, split around m
                    if rematerializable(l)
                        Insert a LOAD-IMMEDIATE for l
                    else
                        Insert a LOAD for l
        For each instruction, i, in b in reverse order
            For each live range, l, defined in i
                For each live range, s, split around l
                    if ¬rematerializable(s)
                        Insert a STORE s
            For each live range, l, used in i
                if l ∉ live
                    For each live range, s, split around l
                        if rematerializable(s)
                            Insert a LOAD-IMMEDIATE for s
                        else
                            Insert a LOAD for s
            Update the live set
```

Fig. 7. Finding a color for splitting

```
findColor(l)
    bestCost ← range[l].cost
    splitFound ← FALSE
    For each color c
        /* Try to split c around l */
        splitOK ← TRUE
        cost ← 0
        For each neighbor, n, of l with colors[n] = c
            if ⟨n, l⟩ ∈ C
                splitOK ← FALSE
            else if rematerializable(n)
                cost ← cost + range[l].loads × rematCost
            else
                cost ← cost + range[l].stores × storeCost +
                    range[l].loads × loadCost
        if splitOK and cost < bestCost
            bestCost ← cost
            bestColor ← c
            splitDir ← splitAroundName
            splitFound ← TRUE
        /* Try to split l around c */
        splitOK ← TRUE
        cost ← 0
        For each neighbor, n, of l with colors[n] = c
            if ⟨l, n⟩ ∈ C
                splitOK ← FALSE
            else if rematerializable(l)
                cost ← cost + range[n].loads × rematCost
            else
                cost ← cost + range[n].stores × storeCost +
                    range[n].loads × loadCost
        if splitOK and cost < bestCost
            bestCost ← cost
            bestColor ← c
            splitDir ← splitAroundColor
            splitFound ← TRUE
    if splitFound
        colors[l] ← bestColor
        if splitDir = splitAroundName
            For each neighbor, n, of l with colors[n] = bestColor
                Mark n to be split around l
        else  /* splitDir = splitAroundColor */
            For each neighbor, n, of l with colors[n] = bestColor
                Mark l to split around n
```

Fig. 8. Splitting allocator



6 Putting It Together

Figure 8 shows a flow chart for our new splitting allocator. Three new phases are added to the Briggs-style allocator.

Split costs We estimate the cost of splitting around each live range by counting a STORE instruction before each definition and a LOAD instruction after each death. This computation could easily be folded into the *spill costs* phase of the Briggs-style allocator, but we show it as a separate phase for clarity. Our implementation builds the containment graph in *split costs*.[6]

Find color When a live range, l, is chosen for spilling during the *select* phase, it calls the *findColor* routine. This phase selects a color for the live range based on either the cost of splitting that color across l or the cost of splitting l across that color. If a color is found, l is assigned that color and the appropriate live ranges are marked for splitting.

Split code We must insert the LOAD and STORE instructions according to the selections made by the *findColor* routine. This process could easily be folded into the *spill code* phase of the Briggs-style allocator, but we show it as a separate phase for clarity.

7 Experiments

To assess the impact of our technique, we have implemented it in our experimental Fortran compiler. The compiler is centered around our intermediate language, called ILOC (pronounced "eye-lock"). ILOC is a pseudo-assembly language for a RISC machine with an arbitrary number of symbolic registers. LOAD and STORE operations are provided to access memory, and all computations operate on symbolic registers. The front end translates Fortran into ILOC. The optimizer transforms the ILOC and hands the results to the register allocator. The back end produces code instrumented to count the number of spill instructions executed.

[6] It could be built earlier. We build it here to avoid the extra space overhead during the *build/coalesce* loop. Since coalescing shrinks the interference graph, this reduces the space requirements for the containment graph.

Table 1. Allocating for 32 integer + 32 float registers (dynamic spill operations)

	Briggs	*Splits*	%	*Bergner*
`field`	191,870	174,725	8.94	186,191
`smooth`	52,260	51,338	1.76	38,499
`init`	50,301	50,107	0.39	50,303
`vslv1p`	28,121	5,980	78.73	23,035
`parmvr`	3,456	1,108	67.94	3,378
`radf4`	382	372	2.62	297
`radb4`	382	376	1.57	301
`energy`	296	295	0.34	292
`radb2`	172	146	15.12	114
`radf2`	163	143	12.27	108
`fftb`	128	128	0.00	128
`fftf`	128	128	0.00	128
`radf5`	123	132	-7.32	96
`radb5`	123	118	4.07	82
`putb`	43	44	-2.33	38
`getb`	26	22	15.38	20
`rffti1`	24	19	20.83	20
`slv2xy`	11	9	18.18	11
`pdiag`	6	0	100.00	6

Our initial interest in this problem arose from several studies in which we examined code that resulted from automatic application of aggressive program transformations [10, 6, 14]. As these techniques become more widely applied, compilers will need to deal with their consequences. For this study, we focused on routines from the program `wave5` in the SPEC95 benchmark suite. These routines had been transformed by the insertion of advisory prefetch instructions intended to improve cache behavior [14]. The transformations increased register pressure to the point where spilling was a recognizable performance problem, even on a machine with thirty-two integer and thirty-two floating-point registers.

Table 1 shows the results of our experiment. The *Briggs* column shows the number of spill instructions executed when the code is allocated using the Briggs-style allocator. Our version of the Briggs-style allocator includes optimistic coloring, rematerialization, and biased coloring [4, 5]. The *Splits* column shows the spill code executed using our splitting allocator. The *Bergner* column shows how Bergner *et al.*'s interference region spilling performs on the same code.

In some cases, splitting produces a drastic reduction in the number of operations introduced for spilling. We reduced the spill overhead of `vslv1p` and `parmvr` by 78.73% and 67.94%, respectively. The improvement in `field` is the largest in absolute terms. For the `pdiag` routine, we reduced the dynamic spill overhead by 100%. This does *not* mean that we removed all the spill code from the routine; we simply placed the spill code on paths that were not exercised by this set of input data.

Unfortunately, we did see an increase in the amount of spill code for two routines. Two situations can produce this problem. First, the estimated spill costs may not accurately reflect the true cost at run time. This is the case for both the `radf5` and `putb` routines in our test. Second, spill decisions change the problem seen by subsequent passes of the allocator. This can produce significantly different allocations. In other words, when we cycle around the main loop in Figure 2 or 8, we insert different spill code. Therefore, the next attempt at coloring will have a different interference graph.

Comparing splitting against interference region spilling, it is clear that each technique has its strengths. Splitting outperforms IR spilling on `field` and `vslv1p`, while IR spilling wins on `smooth`. We believe that the two techniques are complimentary; an allocator that trades off the cost of splitting against the cost of IR spilling should produce the better code for each example, moderated, of course, by the fact that the comparison is based on estimated costs rather than actual costs.

8 Summary and Conclusions

Global techniques for the reduction of spill code can reduce the number of memory operations introduced by the register allocator. The potential for live range splitting to reduce spill code has long been recognized; the details of how to implement it in a Chaitin-style register allocator have not. In this paper, we showed that a relatively passive approach to splitting can produce dramatic positive results. The technique is easy to add to an existing Briggs-style allocator. Because our splitting algorithm chooses between splitting and spilling on the basis of costs, it can be combined with Bergner's interference region spilling to create an allocator that captures the improvements of both techniques.

9 Acknowledgments

This work was supported by DARPA through Army contract DABT63-95-C-0115 and by the Trilogy Development Group. The work described in this paper has been done as part of the Massively Scalar Compiler Project at Rice University. The many people who have contributed to that project deserve our gratitude. Preston Briggs of Tera Computer Company has acted as a sounding board for many of our ideas in this area. Tim Harvey contributed substantially to the development of this work with his ideas and his implementations; without his patient support, this work would not have been done. Nat McIntosh transformed `wave5` for prefetching, generating the version that we used in this experiment. Peter Bergner of The University of Minnesota deserves our gratitude for helping us understand the details of his work on interference region spilling.

References

1. Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. *SIGPLAN Notices*, 32(6):287–295,

June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.

2. David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.
3. Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.
4. Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.
5. Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *SIGPLAN Notices*, 27(7):311–321, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.
6. Steve Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, Department of Computer Science, September 1992.
7. Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.
8. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.
9. Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Notices*, 19(6):222–232, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*.
10. Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software – Practice and Experience*, 21(6):581–601, June 1991.
11. Keith D. Cooper, Timothy J. Harvey, and Linda Torczon. How to build an interference graph. *Software–Practice and Experience* (to appear), 1998. Available on the web at `http://softlib.rice.edu/MSCP/publications.html`.
12. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 1989.
13. Janet Fabri. Automatic storage optimization. *SIGPLAN Notices*, 14(8):83–91, August 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction*.
14. Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Notices*, 27(9):62–75, September 1992. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

This article was processed using the LaTeX macro package with LLNCS style