Live Range Splitting in a Graph Coloring Register Allocator^{*}

Keith D. Cooper¹ and L. Taylor Simpson²

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 to model the conflicts that prevent two values from sharing a register. Nodes in the graph represent live ranges, or values. 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 the graph, for k equal to the number of registers on the target machine. If it succeeds, it maps the k colors onto the k 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. In this paper, we present a global approach to reducing spill code based on live range splitting.

We are not the first to study this problem. Bergner *et al.* describe a heuristic called *interference region spilling* that reduces the amount of spill code necessary for spilled live ranges [1]. Briggs experimented with an aggressive form of live range splitting and saw mixed results [3, Chapter 6]. This paper presents a more passive form of live range splitting. The allocator uses splitting as an alternative to spilling. It looks for regions where splitting can break the interferences responsible for the 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. Because our splitting technique chooses between splitting and spilling on the basis of estimated costs, combining it with Bergner's technique will create an allocator that chooses the better spill strategy, on a live range by live range basis.

1 Introduction

Chaitin *et al.* first used graph coloring as a paradigm for register allocation and assignment in a compiler [8, 7]. Coloring allocators operate by building a graph that shows when two values cannot reside in the same location; we call the graph an *interference graph* because the values would interfere with each other if they

^{*} This work was supported by DARPA through contract DABT63-95-C-0115 and by the Trilogy Development Group.

shared a register. The allocator attempts to discover a k-coloring of the graph that is, an assignment of k colors to the nodes of the graph 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, the allocator cannot discover such a coloring, it must select one or more values to store in memory, or *spill*. It inserts code to spill those values and tries to color the resulting, modified procedure.

Chaitin's basic scheme has been improved by other authors. Briggs *et al.* describe a variation on the coloring heuristic that increases 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]. Even with these improvements, some routines still require spill code. The problem is not poor coloring; these routines simply need more registers than the hardware provides.

Once it chooses a live range to spill, Chaitin's original scheme spills the 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 exist to reduce the number of spill instructions inserted into a single basic block [7, 2]. These methods reduce spills within a block that contains multiple references to the spilled value. They do nothing for problems that arise across multiple blocks.

Bergner *et al.* introduced a technique called interference region spilling that takes a global approach to reducing the amount of spill code introduced [1]. Rather than spill a live range everywhere, their method chooses a color for the live range 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



Fig. 1. Example of live range splitting

incurred for each color; it selects the color with the smallest estimated cost.

In this paper, we explore another global method to 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 subsequent 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]. Prior to coloring, his aggressive splitting broke every live range into 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 these difficulties by being more passive. When the allocator selects a live range l as a candidate for spilling, our method considers splitting the live range as an alternative to spilling it. We look 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 everywhere, we perform the splitting instead of spilling. Because it compares the estimated cost of splitting and spilling, our method can easily be combined with Bergner's method. The resulting allocator would either split live ranges to reduce spill costs, or perform interference region spilling to reduce spill costs. It would choose between these alternatives on a live range by live range basis.

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 for either l_1 or l_2 , one of them must be spilled. 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 exist 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 l_1





completely contains l_2 . To let the allocator detect this situation, we introduce a new data structure called the containment graph. Section 3 describes this graph in detail. Section 4 explains how to insert the code that splits a live range.

2 Briggs' Allocator

Because our live range splitting procedure is an extension of a Briggs-style allocator, we will first provide 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 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 renumbering 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 contains a node for each live range and an edge between each pair of live ranges that are simultaneously live. The graph is represented by both a triangular bit matrix and a collection of adjacency lists. We build the graph 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 the interference graph, 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 assign colors to live ranges in the order they are popped from the coloring stack and put back into the interference graph. If no color is available for a live range, it is marked for spilling.⁴ 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 any live ranges are marked 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 the *spill costs* phase. The difference is that it actually inserts LOAD and STORE instructions for any live range marked for spilling.

3 The Containment Graph

Chaitin first observed that spilling a live range does not break all its interferences – this is one of the reasons that we must repeat the coloring process after spill code is inserted. In fact, spilling merely breaks a live range into multiple tiny live ranges. The middle column of Figure 1 depicts this situation. The allocator has spilled l_2 entirely, producing two short live ranges—one at the definition and one at the use. Both these live ranges still interfere with l_1 . If, on the other hand, the allocator had spilled l_1 , the live ranges at l_1 's definition and use would not interfere with l_2 .

More formally, spilling a live range l_i does not break the interference with any live range l_j that is live at either a definition or a use of l_i . If, on the other hand, l_i and l_j interfere, but l_j is not live at a definition or a use of l_i , then l_i contains l_j and the allocator should consider splitting l_i around l_j . To detect this situation, we introduce a new data structure called the *containment graph*. The containment graph is a directed analog of the interference graph. Nodes in the containment graph represent live ranges. An edge from l_j to l_i in the graph indicates that l_i is live at a definition or use of l_j . In other words, if the allocator spills l_j , the edges leaving l_j in the interference graph correspond to edges in the interference graph that will remain unbroken by the spill.⁵ The allocator uses the graph to determine when one live range is wholly contained in another. We represent the containment graph as a square bit matrix. It is twice as large as the triangular bit matrix used for the interference graph, but we do not require the adjacency list representation of the containment graph.

³ Chaitin's algorithm always spilled the chosen spill candidate. Briggs' method lets the select phase see if a color is available. If no color is available when the spill candidate is popped from the stack in select, it is left uncolored and it is spilled.

⁴ An uncolored live range must have been chosen as a spill candidate in simplify. See Briggs *et al.* for a detailed discussion of this point [4, 5].

⁵ In other words, the tiny live ranges left at definitions and uses of l_j will continue to interfere with l_i if and only if $\langle l_j, l_i \rangle$ is an edge in the containment graph.

Fig. 3. Examples of the containment graph



Figure 3 shows some examples that illustrate the utility of the containment graph. In the left column, l_2 is not live at either the definition or the use of l_1 . Therefore, there is no edge from l_1 to l_2 . In the middle column, l_1 and l_2 overlap so there is an edge in the containment graph in both directions. The right column is similar to the left column except that there is a use of l_1 while l_2 is live. Therefore, there is an edge in both directions.

These examples illustrate how we can use the containment graph for live range splitting. If there is no edge in the interference graph between l_1 and l_2 , then either l_1 and l_2 do not interfere or l_2 is completely contained inside l_1 . Therefore, we can split l_1 across l_2 if and only if l_1 and l_2 interfere and there is no edge from l_1 to l_2 in the containment graph. Figure 4 shows the algorithm for building the containment graph; it is very similar to the algorithm for building the interference graph.

The primary drawback to using the containment graph is the space required to hold the bit-matrix. Two different facts should moderate this problem.

- 1. The containment graph contains all the information found in the interference graph. Thus, we do not need the lower-triangular bit-matrix form of the interference graph. If the edge $\langle l_i, l_j \rangle$ is the interference graph, then one or both of $\langle l_i, l_j \rangle$ and $\langle l_j, l_i \rangle$ must be in the containment graph. The containment graph is no harder to build than the lower-triangular bit-matrix.
- 2. A bit matrix may be space inefficient for the containment graph. A recent study of techniques for building the interference graph showed that a closed hash table implementation can use less space for sufficiently large graphs [11]. The containment graph should reach that threshold much earlier than the lower-triangular bit-matrix.

Taken together, these suggestions should reduce the space impact of building the containment graph rather than an undirected interference graph.

For clarity of exposition, we describe the algorithm as if it must build the graphs separately, and at different times. However, the implementor can easily use a single graph and build it during the **build** phase of the original Chaitin-Briggs scheme.

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 prof*itable*. 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. It takes 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 $liveOut_i - liveIn_i$.

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, n 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.

Fig. 4. Building the containment graph

buildContainmentGraph() Allocate the square bit matrix For each block b $live \leftarrow liveOut_b$ For each instruction, i, in b in reverse order For each live range, l, defined in iFor each $m \in live$ Add edge $\langle l, m \rangle$ to containment graph Update the live set For each live range, l, used in iFor each $m \in live$ Add edge $\langle l, m \rangle$ to containment graph

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.⁶ 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 splitling 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.

We will explain how the process works when applied to 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 in the containment graph from l_1 to l_2 , so the splitting is possible. Since the cost of the split is less than the cost of spilling l_2 entirely, we choose color cfor 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.

⁶ 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 \leftarrow 10^{depth(b)}
       live \leftarrow live Out_b
       For each successor, s, of b
           deaths \leftarrow liveOut_b - liveIn_s
           For each m \in deaths
               range[m].loads \gets 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 \leftarrow range[l].stores + weight
           For each live range, l, used in i
               if l \notin live
                   range[l].loads \leftarrow range[l].loads + weight
           Update the live set
splitCode()
   For each block b
       live \leftarrow liveOut<sub>b</sub>
       For each successor, s, of b
           deaths \leftarrow liveOut_b - liveIn_s
           For each m \in 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 \neg rematerializable(s)
                       Insert a STORE s
           For each live range, l, used in i
               if l \notin 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
find Color(l)
    bestCost \leftarrow range[l].cost
    splitFound \leftarrow FALSE
    For each color c
        /* Try to split c around l */
        splitOK \leftarrow TRUE
        cost \gets 0
        For each neighbor, n, of l with colors[n] = c
            if \langle n, l \rangle \in \text{containment graph}
                splitOK \leftarrow \texttt{FALSE}
            else if rematerializable(n)
                cost \leftarrow cost + range[l].loads \times rematCost
            else
                cost \leftarrow cost + range[l].stores \times storeCost +
                     range[l].loads \times loadCost
        if splitOK and cost < bestCost
            bestCost \gets cost
            bestColor \leftarrow c
            splitDir \leftarrow splitAroundName
            splitFound \leftarrow TRUE
        /* Try to split l around c */
        splitOK \leftarrow \texttt{TRUE}
        cost \leftarrow 0
        For each neighbor, n, of l with colors[n] = c
            if \langle l, n \rangle \in \text{containment graph}
                splitOK \leftarrow FALSE
            else if rematerializable(l)
                cost \leftarrow cost + range[n].loads \times rematCost
            else
                cost \leftarrow cost + range[n].stores \times storeCost +
                     range[n].loads \times loadCost
        if splitOK and cost < bestCost
            bestCost \gets cost
            bestColor \leftarrow c
            splitDir \leftarrow splitAroundColor
            splitFound \leftarrow TRUE
   if splitFound
        colors[l] \leftarrow bestColor
        if splitDir = splitAroundName
            For each neighbor, n, of l with colors[n] = bestColor
                Mark n to be split around l
        else
            For each neighbor, n, of l with colors[n] = bestColor
                Mark l to split around n
```



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. During this phase, we also build the containment graph.⁷ This computation could easily be folded into the *spill costs* phase of the Briggsstyle allocator, but we show it as a separate phase for clarity. We do not build the containment graph during the *Build* phase because it is not needed during the *Build/Coalesce* loop.
- 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 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.

⁷ If the implementor is using a single graph to represent both the interference graph and the containment graph, it will already have been built.

	Briggs	Splits	%	Bergner
field	191870	174725	8.94	186191
smooth	52260	51338	1.76	38499
init	50301	50107	0.39	50303
vslv1p	28121	5980	78.73	23035
parmvr	3456	1108	67.94	3378
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

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

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 Acknowledgements

This work was supported by DARPA through Army contract DABT63-95-C-0115. 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 did much of the implementation work that supports this effort; without his patient support, this work would not have been done. Nat McIntosh provided the test code 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

 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.
- Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. SIG-PLAN Notices, 27(7):311-321, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.
- Steve Carr. Memory-Hierarchy Management. PhD thesis, Rice University, Department of Computer Science, September 1992.
- 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.
- 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.
- 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.
- 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), 1997. Available on the web at http://softlib.rice.edu/MSCP/publications.html.
- 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.
- 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 IAT_FX macro package with LLNCS style