Rematerialization

Preston Briggs
Keith D. Cooper
Linda Torczon

Department of Computer Science*
Rice University
Houston, TX 77251–1892

Abstract

This paper examines a problem that arises during global register allocation – *rematerialization*. If a value cannot be kept in a register, the allocator should recognize when it is cheaper to recompute the value (rematerialize it) than to store and reload it. Chaitin's original graph-coloring allocator handled simple instances of this problem correctly. This paper details a general solution to the problem and presents experimental evidence that shows its importance.

Our approach is to tag individual values in the procedure's SSA graph with information specifying how it should be spilled. We use a variant of Wegman and Zadeck's *sparse simple constant* algorithm to propagate tags throughout the graph. The allocator then splits live ranges into values with different tags. This isolates those values that can be easily rematerialized from values that require general spilling. We modify the base allocator to use this information when estimating spill costs and introducing spill code.

Our presentation focuses on rematerialization in the context of Chaitin's allocator; however, the problem arises in any global allocator. We believe that our approach will work in other allocators – while the details of implementation will vary, the key insights should carry over directly.

1 Introduction

In the past decade, the literature on register allocation has focused largely on *global* allocation – methods that factor information about the entire procedure into the decision process. Because the problem of optimal register allocation is NP-complete [19], compilers employ heuristic techniques to approximate its solution. In general, today's generation of global allocators produces "good" approximations; however, careful examination of the output of these allocators reveals that there is still room for improvement.

This paper examines a specific problem that arises in global register allocation – *rematerialization*. When a value must be spilled, the allocator should recognize those cases when it is cheaper to recompute the value than to store and retrieve it from memory. While our discussion is set in the context of a Chaitin-style graph-coloring allocator [6, 5, 3, 1], the same questions seem to arise in all global allocators.

Consider the code fragments shown in Figure 1 (the notation [p] means "the contents of the memory location addressed by p"). Examining the *Source* column, we note that p is constant in the first loop, but varying in the second loop. The register allocator should take advantage of this situation.

Imagine that high demand for registers in the first loop forces p to be spilled; the *Ideal* column shows the desired result. In the upper loop, p is loaded just before it is needed (using some sort of "load-immediate" instruction). For the lower loop, p is loaded just before the loop, again using a load-immediate.

The *Chaitin* column illustrates the code that would be produced by a Chaitin-style allocator. The entire live range of p has been spilled to memory, with loads inserted before uses and stores inserted after definitions.

The final column shows code we would expect from a "splitting" allocator [8, 17, 16, 4]; the actual code might be worse. In fact, our work on rematerialization was motivated by problems observed during our own experiments with live range splitting. Unfortunately, examples of this sort are not discussed in the literature on splitting allocators and it is unclear how best to extend these techniques to achieve the *Ideal* solution.

We begin our discussion with a brief review of Chaitin-style allocators. Section 3 gives a high-level view of our approach to rematerialization. Section 4 describes the low-level modifications to the allocator required to support our approach. Experimental results are presented in Section 5. Subsequent sections suggest extensions and make comparisons with other work.

*This work has been supported by DARPA through ONR grant N00014-91-J-1989 and by the IBM Corporation.

ACM SIGPLAN '92 PLDI-6/92/CA
© 1992 ACM 0-89791-476-7/92/0006/0311...$1.50

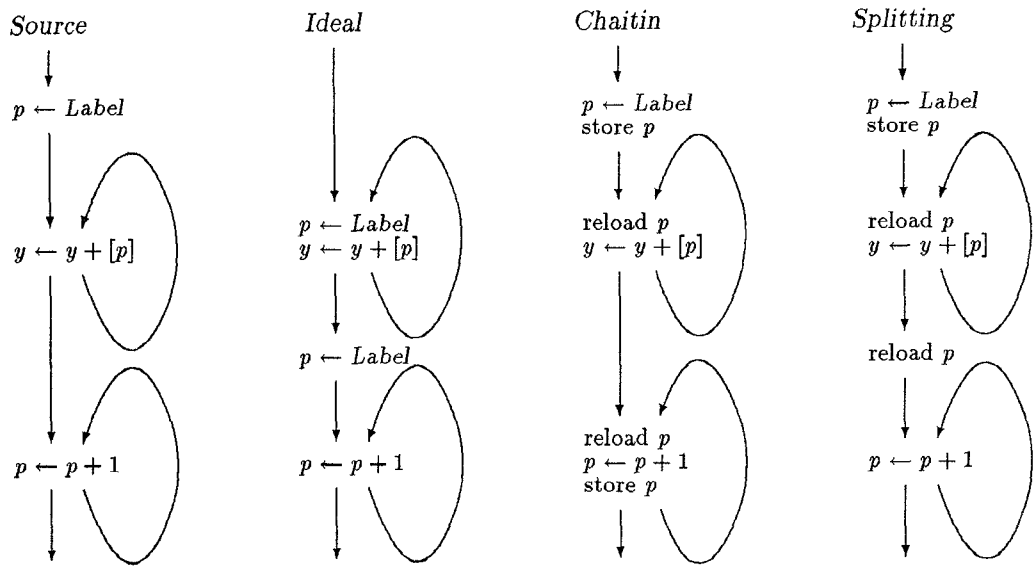

Figure 1: Rematerialization versus Spilling

2 Background

The notion of modeling register allocation as a graph coloring problem descends from very early work on storage allocation [18]. The first actual implementation was done by Chaitin *et al.* in the PL.8 compiler [6]. Chow and Hennessy described a priority-based scheme built on a coloring paradigm [8]. Our own work has built on Chaitin's approach [3]. To distinguish our allocator from Chaitin's, we call it the *optimistic* allocator.

Throughout this paper, we assume that the allocator works on either low-level intermediate code or assembly code. Before allocation, the code can reference an unlimited number of virtual registers. A single virtual register can have disconnected lifetimes in distinct parts of the procedure. Rather than map virtual registers directly onto physical registers, the allocator discovers the distinct *live ranges* in a procedure and allocates them to physical registers.

To model register allocation as a graph coloring problem, the compiler first constructs an interference graph G. Nodes in G represent live ranges; edges represent *interferences*. Thus, there is an edge 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. Live ranges that interfere with l_i are its *neighbors* in the graph; the *degree* of l_i is the number of neighbors it has in the graph.

To find an allocation from G, the compiler looks for a k-coloring of G; that is, an assignment of k colors to the nodes of G such that neighboring nodes always have distinct colors. If we choose k to match the number of machine registers, then we can map a k-coloring for G into a feasible register assignment for the underlying code. Because finding a k-coloring of an arbitrary graph 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 a k-coloring is not discovered, some live ranges are *spilled*; *i.e.*, the values are kept in memory rather than registers.

Spilling one or more live ranges changes both the code and the interference graph. The compiler proceeds by iteratively spilling some live ranges and attempting to color the resulting new graph. This process is guaranteed to terminate. In practice, this process converges quickly [5, 3] (see also Table 2). Figure 2 illustrates the overall flow of the optimistic allocator.

Renumber finds the live ranges and gives them unique names. It creates a new live range for each definition point and unions together the live ranges that reach each use point.

Build constructs the interference graph using the dual representations, a triangular bit-matrix and a set of adjacency vectors, advocated by Chaitin [5].

Coalesce attempts to combine live ranges. Two live ranges l_i and l_j are combined, giving l_{ij}, if the initial definition of l_j is a copy from l_i and there is no interference between l_i and l_j. This has several beneficial effects; most importantly, it eliminates the copy and reduces the total degree of the graph. Since some coalesces can preclude others, the allocator should work "inside out," examining deeply-nested blocks first.

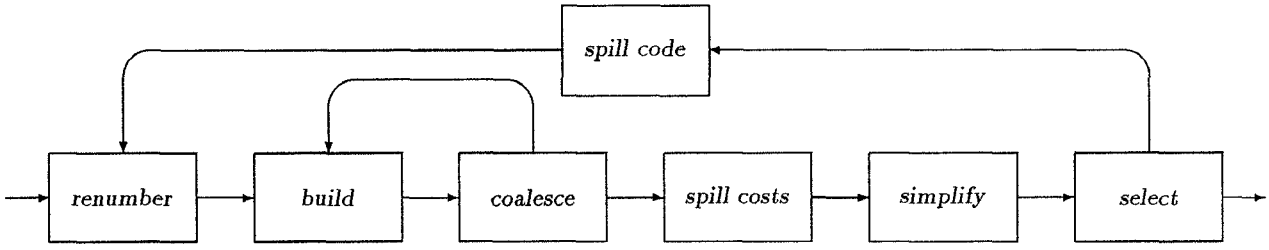

Figure 2: The Optimistic Allocator

Spill Costs estimates the cost associated with spilling each live range. Spill cost is computed as the cost of the memory accesses required to spill the live range, each weighted by 10^d where d is the instruction's loop nesting depth.

Simplify constructs an ordering of the nodes. It removes nodes with current degree less than k from G, pushes them on a stack, and decrements the degree of their neighbors. If all remaining nodes have k or more neighbors, it chooses a *spill candidate*, removes it from G, and pushes it on the stack.

The metric for picking spill candidates is critical. Chaitin suggested choosing the node that minimizes spill cost divided by degree [5].

Select assigns colors to the nodes of G in the order determined by *simplify*. *Select* repeatedly pops a node from the stack and attempts to give it a color distinct from its colored neighbors. If no color is available for a node, it is left uncolored. If all nodes receive colors, allocation is complete.

Spill Code is invoked if *select* left a node uncolored. It converts each such node into a collection of tiny live ranges by inserting a load or store at each use and definition.

Detailed descriptions of these processes can be found in Chaitin's work and our earlier paper [5, 6, 3].

3 Rematerialization

Chaitin *et al.* discuss several ideas for improving the quality of spill code [6]. They point out that certain values can be recomputed in a single instruction and that the required operands will always be available for the computation. They call these exceptional values *never-killed* and note that such values should be recalculated instead of being spilled and reloaded. They further note that an uncoalesced copy of a never-killed value can be eliminated by recomputing it directly into the desired register [6]. Together, these techniques are termed *rematerialization*. In practice, opportunities for rematerialization include:

- immediate loads of integer constants and, on some machines, floating-point constants,
- computing a constant offset from the frame pointer or the static data area pointer,
- loads from a known constant location in either the frame or the static data area, and
- loading non-local frame pointers from a *display*.

The values must be cheaply computable from operands that are available throughout the procedure.

It is important to understand the distinction between *live ranges* and *values*. A live range may comprise several values connected by common uses. In the *Source* column of Figure 1, p denotes a single live range composed from three values: the address *Label*, the result of the expression $p + 1$, and (more subtly) the merge of those two values at the head of the second loop.

Chaitin's allocator correctly handles rematerialization when spilling live ranges with a single value, but cannot handle more complex cases; *e.g.*, the variable p in Figure 1. Our task is to extend Chaitin's work to take advantage of rematerialization opportunities for complex, multi-valued live ranges. Our approach is to tag each value with enough information to allow the allocator to handle it correctly. To achieve this, we

1. split each live range into its component values,
2. propagate rematerialization tags to each value, and
3. form new live ranges from connected values having identical tags.

This approach allows correct handling of rematerialization, but introduces the new problem of minimizing unnecessary splits. The following sections describe how to find values, how to propagate tags, how to split the live ranges, and how to remove unproductive splits.

3.1 Discovering Values

To find values, we construct the procedure's *static single assignment* (SSA) graph, a representation that transforms the code so that each use of a value references exactly one definition [11]. To achieve this goal, the construction technique inserts special definitions called ϕ-nodes at those points where control-flow paths join and different values merge. We actually use the *pruned* SSA, with dead ϕ-nodes eliminated [7].

A natural way to view the SSA graph for a procedure is as a collection of values, each composed of a single definition and one or more uses. Each value's definition is either a single instruction or a ϕ-node that merges two or more values. By examining the defining instruction for each value, we can recognize never-killed values and propagate this information throughout the SSA graph.

3.2 Propagating Rematerialization Tags

To propagate tags, we use an analog of Wegman and Zadeck's *sparse simple constant* algorithm [21]. We modify their lattice slightly to represent the necessary rematerialization information. The new lattice elements may have one of three types:

$\top$ *Top* means that no information is known. A value defined by a copy instruction or a ϕ-node has an initial tag of $\top$.

inst If a value is defined by an appropriate instruction (*never-killed*), it should be rematerialized. The value's tag is simply a pointer to the instruction.

$\bot$ *Bottom* means that the value must be spilled and restored. Any value defined by an "inappropriate" instruction is immediately tagged with $\bot$.

Additionally, their *meet* operation $\sqcap$ is modified correspondingly. The new definition is:

any	$\sqcap$	$\top$	=	any	
any	$\sqcap$	$\bot$	=	$\bot$	
$inst_i$	$\sqcap$	$inst_j$	=	$inst_i$	if $inst_i = inst_j$
$inst_i$	$\sqcap$	$inst_j$	=	$\bot$	if $inst_i \neq inst_j$

Note that $inst_i = inst_j$ compares the instructions on an operand-by-operand basis. Since our instructions have at most 2 operands, this modification does not affect the asymptotic complexity of propagation.

During propagation, each value will be tagged with a particular *inst* or $\bot$. Values defined by a copy instruction will have their tags *lowered* to *inst* or $\bot$, depending on the value that flows into the copy. Tags for values defined by ϕ-nodes will be lowered to *inst* if and only if all the values flowing into the node have equivalent *inst* tags; otherwise, they are lowered to $\bot$.

This process tags each value in the SSA graph with either an instruction or $\bot$. If a value's tag is $\bot$, spilling that value requires a normal, heavyweight spill. If, however, its tag is an instruction, it can be rematerialized by issuing the instruction specified by the tag. The tags are used in two phases of the allocator: *spill costs* uses the tags to compute more accurate spill costs and *spill code* uses the tags to emit the desired code.

3.3 Inserting Splits

After propagation, the ϕ-nodes must be removed and values renamed to recreate an executable program. Consider the example in Figure 3. The *Source* column simply repeats the example introduced in Figure 1. The *SSA* column shows the effect of inserting a ϕ-node for p and renaming the different values comprising p's live range. The *Splits* column illustrates the copies necessary to distinguish the different values without ϕ-nodes. The final column (*Minimal*) shows the single copy required to isolate the never-killed value p_0 from the other values comprising p. We avoid the extra copy by noting that p_1 and p_2 have identical tags after propagation (both are $\bot$) and may be treated together as a single live range p_{12}. Similarly, two connected values with the same *inst* tag would be combined into a single live range.

For the purposes of rematerialization, the copies are placed perfectly – the never-killed value has been isolated and no further copies have been introduced. The algorithm for removing ϕ-nodes and inserting copies is described in Section 4.1. In Section 6, we discuss the possibility of including *all* the copies suggested in the *Splits* column.

3.4 Removing Unproductive Splits

Our approach inserts the minimal number of copies required to isolate the never-killed values. Nevertheless, coloring can make some of these copies superfluous. Recall the *Minimal* column in Figure 3. If neither p_0 nor p_{12} are spilled and they both receive the same color, the copy connecting them is unnecessary. Because it has a real run-time cost, the copy should be eliminated whenever possible. Of course, *coalesce* would remove *all* of the copies, losing the desired separation between values with different tags. So, we use a pair of limited coalescing mechanisms to remove unproductive copies:

Conservative coalescing is a straightforward modification of Chaitin's *coalesce* phase. Conceptually, we add a single constraint to *coalesce* – only combine two live ranges if the resulting single live range will not be spilled.

Biased coloring increases the likelihood that live ranges connected by a copy get assigned to the same register. Conceptually, *select* tries to assign the same color to two live ranges connected by a copy.

Taken together, these two mechanisms remove most of the unproductive copies.

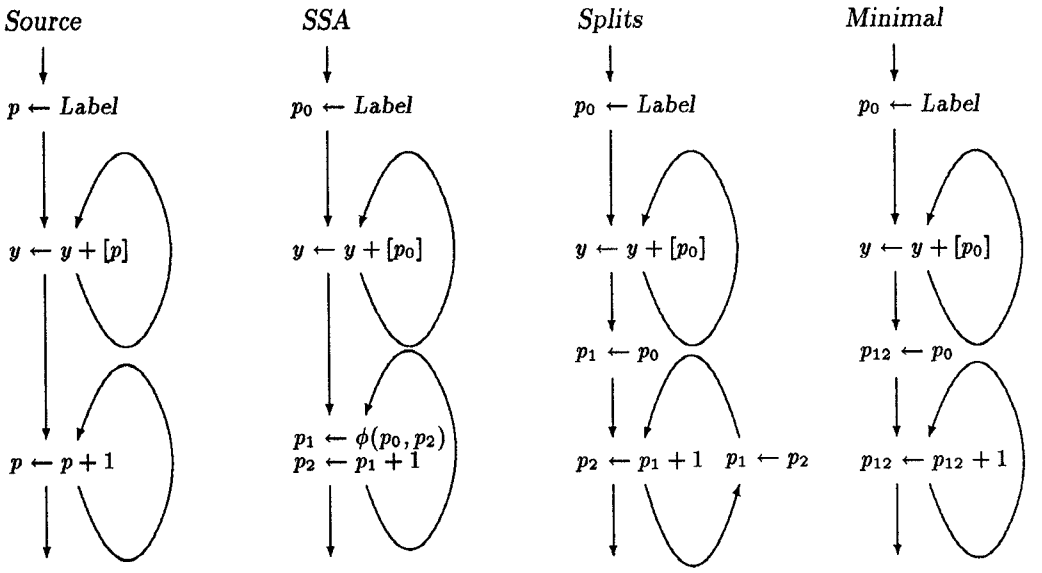

Figure 3: Introducing Splits

4 Implementation

Chaitin-style allocators can be extended naturally to accommodate our approach. The high-level structure depicted in Figure 2 is unchanged, but a number of low-level modifications are required. The next sections discuss the enhancements required in *renumber*, *coalesce*, and *select*.

4.1 Renumber

Chaitin's version of *renumber* (termed "getting the right number of names") was based on def-use chaining [6]. Long before our interest in rematerialization, we adopted an implementation strategy for *renumber* based on the pruned SSA graph. Conceptually, the old implementation has four steps:

1. Determine liveness at each basic block using a sparse data-flow evaluation graph [7].
2. Insert ϕ-nodes based on dominance frontiers [11]. Avoid inserting dead ϕ-nodes.
3. Renumber the operands in every instruction to refer to values instead of the original virtual registers. At the same time, accumulate availability information for each block. The intersection of *live* and *avail* is needed at each block to allow construction of a precise interference graph [6].
4. Form live ranges by unioning together all the values reaching each ϕ-node using a fast disjoint-set union. The disjoint-set structure is maintained while building the interference graph and coalescing (where coalesces are further union operations).

In our implementation, steps 3 and 4 are performed during a single walk over the dominator tree. Using these techniques, *renumber* completely avoids the use of *bit-vectored* flow analysis. Despite the apparent complexity of the algorithms involved, it is very fast in practice and requires only a modest amount of space.

Because *renumber* already uses the SSA graph, only modest changes are required to support rematerialization. The modified *renumber* has six steps:

1. Determine liveness at each basic block using a sparse data-flow evaluation graph.
2. Insert ϕ-nodes based on dominance frontiers, still avoiding insertion of dead ϕ-nodes.
3. Renumber the operands in each instruction to refer to values. At the same time, initialize the rematerialization tags for all values.
4. Propagate tags using the sparse simple constant algorithm as modified in Section 3.2.
5. Examine each copy instruction. If the source and destination values have identical *inst* tags, we can union them and remove the copy.
6. Examine the operands of each ϕ-node. If an operand value has the same tag as the result value, union the values; otherwise, insert a *split* (a distinguished copy instruction) connecting the values in the corresponding predecessor block.

Steps 5 and 6 are performed in a single walk over the dominator tree.

4.2 Conservative Coalescing

To prevent coalescing from removing the splits that have been carefully introduced in *renumber*, we must limit its power. Specifically, it should never coalesce a split instruction if the resulting live range may be spilled. In normal coalescing, two live ranges l_i and l_j are combined if l_j is defined by a copy from l_i and they do not otherwise interfere. In conservative coalescing, we add an additional constraint: combine two live ranges connected by a split if and only if l_{ij} has $< k$ neighbors of "significant degree," where significant degree means a degree $\geq k$.

To understand why this restriction is safe (indeed, it is conservative), recall Chaitin's coloring heuristic [5]. Before any spilling, nodes of degree $< k$ are removed from the graph. When a node is removed, the degrees of its neighbors are reduced, perhaps allowing them to be removed. This process repeats until the graph is empty or all remaining nodes have degree $\geq k$. Therefore, for a node to be spilled, it must have at least k neighbors with degree $\geq k$ in the initial graph.

In practice, we perform two rounds of coalescing. Initially, all possible copies are coalesced (but not split instructions). The graph is rebuilt and coalescing is repeated until no more copies can be removed. Then, we begin conservatively coalescing split instructions. Again, we repeatedly build the interference graph and attempt further conservative coalescing until no more splits can be removed.

In theory, we should not intermix conservative coalescing with unrestricted coalescing, since the result of an unrestricted coalesce may be spilled. For example, l_i and l_j might be conservatively coalesced, only to have a later coalesce of l_{ij} with l_k provoke the spilling of l_{ijk} (since the significant degree of l_{ijk} may be quite high). In practice, this may not prove to be a problem, permitting a simplification of the entire process.

Conservative coalescing directly improves the allocation. Each coalesce removes an instruction from the resulting code – a split instruction that was introduced by the allocator. In regions where there is little competition for registers (a region of low register pressure), conservative coalescing undoes all splitting. It cannot, however, undo all of the non-productive splits by itself.

4.3 Biased Coloring

The second mechanism for removing useless splits involves changing the order in which colors are considered for assignment. Before coloring, the allocator finds *partners* – values connected by splits. When *select* 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 conservative coalescing cannot. For example, l_i might have $2k$ neighbors of significant degree; but these neighbors might not interfere with each other and thus might all be colored identically. Conservative coalescing cannot combine l_i with any of its partners; the resulting live range would have too many neighbors of significant degree. Biasing may be able to combine l_i and its partners because it is applied after the allocator has shown that both live ranges will receive colors. At that late point in allocation, combining them is a matter of choosing the right colors. By virtue of its late application, the biasing mechanism uses a detailed level of knowledge about the problem that is not available any earlier in the process – for example, when coalescing is performed.

To increase the likelihood that biasing will match partners, we can add *limited lookahead*. When picking a color for l_i, if it has an uncolored partner l_j, the allocator can look for a color that is still available for l_j. On average, l_i has a small number of partners; thus, we can add *limited lookahead* to biased coloring without increasing the asymptotic complexity of *select*.

5 Experimental Results

To support our research, we have written an optimizing compiler for FORTRAN. The compiler is part of the ParaScope programming environment and includes support for interprocedural analysis and a variety of traditional optimizations [9]. We currently generate code for the IBM RT/PC and have experimental code generators for the Sparc, i860, and RS/6000. To experiment with register allocation, we have built a series of allocators that are independent of any particular architecture [2].

Our experimental allocators work with routines expressed in ILOC, a low-level intermediate language designed to allow extensive optimization. An ILOC routine that assumes an infinite register set is rewritten in terms of a particular target register set, with spill code added as necessary. The target register set is specified in a small table and may be varied to allow convenient experimentation with a wide variety of register sets.

After allocation, each ILOC routine is translated into a complete C routine. Each C routine is compiled and the resulting object files are linked into a complete program. There are several advantages to this approach:

- By inserting appropriate instrumentation during the translation to C, we are able to collect accurate, *dynamic* measurements.
- Compilation to C allows us to test a single routine in the context of a complete program running with real data.
- We are able to perform our tests in a machine-independent fashion, potentially using a variety of register sets.

```
LLE3:  nop
LLA4:  ldi   r14   8
       add   r9    r15   r11
       mvf   f15   f0
       bc    L0023

L0023: lddrr f14   r14   r9
       dabs  f14   f14
       dadd  f15   f15   f14
       addi  r14   r14   8
       sub   r7    r10   r14
       br    ge    r7    N6    N7
```

```
LLE3:
LLA4:  r14 = (int) (8); i++;
       r9 = r15 + r11;
       f15 = f0; c++;
       goto L0023;

L0023: f14 = *((double *) (r14 + r9)); l++;
       f14 = fabs(f14);
       f15 = f15 + f14;
       r14 = r14 + (8); a++;
       r7 = r10 - r14;
       if (r7 >= 0) goto N6; else goto N7;
```

Figure 4: ILOC and C

Simply timing actual machine code is inherently machine-dependent and tends to obscure the effects of allocation. During the translation into C, we can add instrumentation to count the number of times any specific ILOC instruction is executed. For comparing register allocators, we are interested in the number of loads, stores, copies, load-immediates, and add-immediates.

Figure 4 shows a small sample of ILOC code and the corresponding C translation. Usually there is a one-to-one mapping between the ILOC statements and the C translations, though some additional C is required for the function header and declarations of the "register" variables; *e.g.*, `r14` and `f15`. Also note the very simple instrumentation appearing immediately after several of the statements. Of course, this code is very simple, but the majority of ILOC is no more complex.

5.1 The Target Machine

For the tests reported here, our target machine is defined to have sixteen integer registers and sixteen floating-point registers. Each floating-point register can hold a double-precision value, so no distinction is made between single-precision and double-precision values once they are held in registers. Up to four integer registers may be used to pass arguments (recall that arguments are passed by reference in FORTRAN; therefore, the argument registers hold pointers to the actual values); any remaining arguments are passed in the stack frame. Function results are returned in an integer or floating-point register, as appropriate. Ten of each register class are designated as callee-saves; the remaining six (including the argument registers) are not preserved by the callee.

When reporting costs, we assume that each load and store requires two cycles; all other instructions are assumed to require one cycle. Of course, these are only simple approximations of the costs on any real machine.

5.2 Spill Costs

Since our instrumentation reports dynamic counts of *all* loads, stores, *etc.*, we need a mechanism for isolating the instructions due to allocation. A difficulty is that some spills are profitable. In other cases, the allocator removes instructions; *e.g.*, copy instructions. Therefore, we tested each routine on a hypothetical "huge" machine with 128 registers, assuming this would give a nearly perfect allocation. The difference between the "huge" results and the results for one of the allocators targeted to our "standard" machine should equal the number of cycles added by the allocator to cope with insufficient registers.

5.3 The Test Suite

Our test suite is a collection of seventy routines contained in eleven programs. Eleven routines are from Forsythe, Malcolm, and Moler's book on numerical methods [13]. They are grouped into seven programs with simple drivers. The remaining fifty-nine routines are from the SPEC benchmark suite [20]. Four of the SPEC programs were used: `doduc` (41 routines), `fpppp` (12 routines), `matrix300` (5 routines), and `tomcatv` (1 routine). The two other FORTRAN programs in the suite (`spice` and `nasa7`) require language extensions not yet supplied by our front-end.

Table 1 summarizes the effect of our new approach to rematerialization. It compares two versions of the optimistic allocator that differ only in their handling of never-killed values. The column labeled *Optimistic* gives data for a version that uses Chaitin's limited approach to rematerialization. The column labeled *Rematerialization* gives data for a version incorporating our new method. The table shows only routines where a difference was observed.

The first two columns give the program and subroutine name. The third and fourth column give the observed spill costs for the two allocators being compared.

		Cycles of Spill Code		Percentage Contribution					
program	*routine*	*Optimistic*	*Rematerialization*	*load*	*store*	*copy*	*ldi*	*addi*	*total*
rkf45	fehl	68	50	26		7	−7		27
seval	spline	117	102	10	2	2	−1		13
solve	decomp	305	286	4	3		−1		6
svd	svd	1,977	1,966	1		0	−0		1
zeroin	zeroin	236	234	2		−1			1
doduc	bilan	1,046	966	5	3				8
	bilsla	16	15				6		6
	colbur	19	24	−11	−11	−5			−26
	ddeflu	335	375	−5	−7	1	1		−12
	debico	459	418	6	0	1	2		9
	deseco	4,957	4,636	7	2		−2	0	7
	drepvi	218	175	4	14	0	2		20
	drigl	32	31				3		3
	heat	34	31	6		1			9
	ihbtr	400	395	1	0		−0		1
	inideb	50	48				4		4
	inisla	31	28		6		3		10
	inithx	579	437	17	10		−2		25
	integr	502	372	18	12		−3		26
	lectur	221	166			2	23		25
	orgpar	39	35		5	−3	8		10
	paroi	1,433	1,383	8	0	−1	−4		4
	pastem	289	220	20	10	13	−19		24
	prophy	1,531	1,525		0		0		0
	repvid	599	404	9	13	11			33
fpppp	d2esp	35	34	6			−3		3
	main	210	199			0	5		5
	twldrv	11,311,624	11,198,058	2	0		−1		1
matrix300	sgemm	9,905	8,398	12	6		−3		15
tomcatv	tomcatv	367,995,733	355,039,258	4	0	−0			4

Table 1: Effects of Rematerialization

These costs are calculated from dynamic counts of instructions as described earlier. The last column (*total*) gives the percentage improvement in spill costs due to improved rematerialization – large positive numbers indicate significant improvements. The middle columns show the contribution of each instruction type to the total.

All percentages have been rounded to the nearest integer. Insignificant improvements are reported as 0 and insignificant losses are reported as −0. In cases where the result *is* zero, we simply show a blank. Since results are rounded, a *total* entry may not equal the sum of the contributing entries.

Consider the first row in Table 1. This row presents results for the routine `fehl` from the program `rkf45`. The optimistic allocator generated an allocation requiring 68 cycles of spill code; the enhanced allocator required only 50 cycles. 26% of the savings came from having to execute fewer loads and 7% arose from fewer copies. There was a 7% degradation due to more load-immediates. The total improvement was 27%.

From the entire suite of 70 routines, we observed improvements in 28 cases and degradations in only 2 cases. One loss was very small (2 loads, 2 stores, and an extra copy); the other was somewhat larger. Improvements ranged from tiny to reasonably large, with many greater than 20%. Of course, adjusting the relative costs of each instruction, especially loads and stores, will change the amount of improvement.

As expected, we see a pattern of fewer load instructions and more load-immediates. Typically, the number of stores and the number of copies are also reduced. The reduced number of copy instructions suggests that our heuristics for removing unhelpful splits are adequate in practice. Note that this reduction is obtained in spite of the extra copies introduced by *renumber*.

5.4 Allocation Costs

The improved support for rematerialization comes at a cost in allocation time. An extra pass over the code is required to initialize rematerialization tags before prop-

	repvid		tomcatv		twldrv	
Phase	*Old*	*New*	*Old*	*New*	*Old*	*New*
cfa	.00	.00	.00	.00	.01	.01
renum	.03	.05	.06	.10	.57	.91
build	.17	.17	.39	.43	10.27	8.81
costs	.01	.01	.02	.02	.16	.14
color	.02	.02	.04	.04	1.16	1.21
spill	.01	.01	.02	.02	.17	.16
renum	.02	.03	.02	.04	.10	.17
build	.06	.05	.09	.12	.63	.83
costs	.01	.01	.01	.01	.07	.06
color	.01	.02	.02	.03	.14	.21
spill	.01	.00	.01	.01	.03	.04
renum	.01	.03	.02	.04	.10	.17
build	.03	.06	.05	.09	.60	.82
costs	.01	.01	.01	.01	.07	.06
color	.01	.01	.02	.02	.13	.21
spill			.01	.01		
renum			.02	.03		
build			.05	.09		
costs			.01	.01		
color			.01	.02		
total	.40	.49	.89	1.13	14.19	13.80

Table 2: Allocation Times in Seconds

agation and further time is required to propagate the tags throughout the routine. Finally, at least one extra pass is required to accomplish *conservative coalescing*. On the other hand, the *build–coalesce* process may be slightly faster since we are able to eliminate some copies during *renumber* (recall step 5 in Section 4.1).

Table 2 shows comparative timings for the two allocators on three routines from the SPEC suite. Times are given in seconds and were measured with a 100 hertz clock on an unloaded IBM RS/6000 Model 540. Each run was repeated 10 times and the results averaged. The first column shows the phase of allocation, where *cfa* stands for control-flow analysis and includes the time required to compute forward and reverse dominators and dominance frontiers, *build* includes the entire *build–coalesce* loop, and *color* includes both *simplify* and *select*. Note that `tomcatv` required an additional round of spilling. For each routine, the *Old* column gives times required by the optimistic allocator with Chaitin's scheme and the *New* column gives times required by the same allocator with improved rematerialization.

We selected three routines to illustrate performance over a range of sizes. The first routine is `repvid`, from the program `doduc`, with 144 non-comment lines of FORTRAN. It compiles to a `.text` size of 1284 bytes using IBM's `xlf` compiler with full optimization. The second routine is `tomcatv`, with 133 lines and a `.text` size of 3064 bytes. The largest routine is `twldrv` from the program `fpppp`, with 881 lines and a `.text` size of 15,616 bytes. All three routines appear in Table 1.

An obvious conclusion to draw from the data in Table 2 is that support for rematerialization can require a small amount of additional compile-time. Occasionally, the new allocator may even be faster, though our experience suggests that `twldrv` is an exceptional case.

The results in Table 2 also illuminate a number of interesting details about the behavior of both allocators.

- The inital pass of the *build–coalesce* loop dominates the overall cost of allocation (as noted by Chaitin). In comparison, additional iterations of the *color–spill* process are quite inexpensive.
- In each case, the cost of *renumber* is higher for the *New* allocator, reflecting the cost of propagating rematerialization tags.
- In all but one case, the cost of the *build–coalesce* loop is higher for the *New* allocator, due to the additional passes of *conservative coalescing*.
- The very low costs of control-flow analysis illustrates the speed and practicality of the algorithm for calculating dominance frontiers [11].
- The higher cost of coloring in the first pass arises from the cost of choosing nodes to spill. While the cost of coloring is linear in the size of the graph, spill selection is $O(s \cdot n)$, where s is the number of spill choices and n is the number of nodes. With a large number of spills, this term dominates the cost of coloring.

We are pleased with the overall speed of both allocators. Our results appear to be slightly faster than the times reported by IBM's `xlf` compiler for register allocation and comparable to the times reported for optimization. In an extensive comparison with priority-based coloring, our allocators appeared much slower on very small routines, but much faster on very large routines [2]. Of course, these speeds are not competitive with the fast, local techniques used in non-optimizing compilers [14, 15]; however, we believe that global optimizations require global register allocation.

6 Extensions

Of course, rematerialization is not the only reason for splitting live ranges; others have observed that splitting a live range can improve the allocation [12, 8]. A natural extension to the scheme described in Section 3 is to split at *all* ϕ-nodes. This lets the allocator pick and choose among all the values in the SSA graph. The machinery used to support rematerialization can easily handle additional splitting.

While inserting copies at all ϕ-nodes introduces additional splitting, it misses a particularly important case. Consider the value p_0 in Figure 3. Because it is unmodified in the first loop, no ϕ-node is created at the loop

header. Assume that additional code exists between the definition of p_0 and the first loop. If this causes p_0 to be spilled, we would like the allocator to consider the loop body separately from the code that precedes it, even though they are part of the same value in the SSA graph. If possible, the allocator should rematerialize p_0 in the loop header, where it will be defined exactly once. This suggests adding extra splits at the top of the loop.

We have experimented with a number of alternative splitting schemes. These include:

1. splitting all live ranges around all loops,
2. splitting all live ranges around outer loops,
3. splitting live ranges around the outermost loop where they are neither used nor defined,
4. splitting along the forward dominance frontiers (at all ϕ-nodes), and
5. splitting based on both forward and reverse dominance frontiers.

Conceptually, it seems easy to incorporate these ideas into our allocator; however, experience has shown that significant engineering is required.

The complete results of our experiments with loop-based splitting are presented in Briggs' thesis [2]. Each scheme had several major successes; each had several equally dramatic failures. While the improvements are large enough to warrant further study, the failures are significant enough to discourage adoption in a production compiler. Of course, we are holding the allocator to a high standard – the results of splitting are compared to the results presented in Section 5. Thus, we immediately notice both improvements and degradations. We intend to continue our search for a consistently profitable approach.

7 Related Work

Our work extends the work described by Chaitin *et al.* and recalls an approach suggested by Cytron and Ferrante. Chaitin et al. introduce the term *rematerialization* and discuss the problem briefly [6]. Because their allocator cannot split live ranges, they handle only the simple case where all definitions contributing to a live range are identical. Our work is a direct extension and is able to handle each component of a complete live range separately and correctly. Cytron and Ferrante suggest splitting based on (the equivalent of) the SSA [10]. Their goal is minimal coloring in polynomial-time – achieved at the cost introducing extra copies. There is no direct discussion of rematerialization; indeed, the discussion of spilling is very sketchy. In contrast, we are concerned primarily with quality of spill code.

It is also interesting to compare our approach to other published alternatives, particularly the splitting allocator of Chow and Hennessy and the hierarchical coloring allocator of Callahan and Koblenz [8, 4]. The published work does not indicate how they handle rematerialization. It is possible that they make no special provisions, trusting their splitting algorithm to do an adequate job. While we can imagine constructing implementations of their techniques to allow more direct comparisons, it is unlikely that the results would accurately reflect the potential of their schemes. In any case, it may be possible to modify their allocators to take advantage of our approach.

8 Summary

The primary contribution of this paper is a natural extension of Chaitin's ideas on rematerialization. In particular, we show how to handle complex live ranges that may be completely or partially rematerialized. We describe a technique for tagging the component values of a live range with correct rematerialization information. We introduce heuristics, especially conservative coalescing and biased coloring, that are required for good results. Finally, we present experimental results that show the effectiveness and practicality of our extensions.

9 Acknowledgements

Greg Chaitin, Ben Chase, John Cocke, Marty Hopkins, Bob Hood, Ken Kennedy, Chuck Lins, Peter Markstein, Tom Murtagh, Randy Scarborough, Rick Simpson, Tom Spillman, and Matthew 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.

References

[1] 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.*

[2] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[3] 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.*

[4] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *SIGPLAN Notices*, 26(6):192–203, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[5] 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.*

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[7] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[8] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[9] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization on the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[10] Ron Cytron and Jeanne Ferrante. What's in a name? The value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[12] Janet Fabri. Automatic storage optimization. *SIGPLAN Notices*, 14(8):83–91, August 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*

[13] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[14] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[15] Christopher W. Fraser and David R. Hanson. Simple register spilling in a retargetable compiler. *Software – Practice and Experience*, 22(1):85–99, January 1992.

[16] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. *SIGPLAN Notices*, 24(7):264–274, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[17] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. *SIGPLAN Notices*, 21(7):255–263, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[18] S. S. Lavrov. Store economy in closed operator schemes. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 1(4):687–701, 1961. English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3, 1962.

[19] Ravi Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, 1975.

[20] SPEC release 1.2, September 1990. Standards Performance Evaluation Corporation.

[21] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.