Improvements to Graph Coloring Register Allocation

PRESTON BRIGGS, KEITH D. COOPER, and LINDA TORCZON
Rice University

We describe two improvements to Chaitin-style graph coloring register allocators. The first, *optimistic coloring*, uses a stronger heuristic to find a k-coloring for the interference graph. The second extends Chaitin's treatment of *rematerialization* to handle a larger class of values. These techniques are complimentary. Optimistic coloring increases the number of procedures that color without spilling and lowers total spill costs when spilling is unavoidable. Rematerialization lowers the cost of spilling some values.

This paper describes both the techniques themselves and our experience building and using register allocators that incorporate them. It provides a detailed description of optimistic coloring and rematerialization. It presents experimental data to show the performance of several versions of the register allocator on a suite of FORTRAN programs. It discusses several insights that we discovered only after repeated implementation of these allocators.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors – compilers, optimization

General Terms: Languages

Additional Keywords and Phrases: Register allocation, code generation, graph coloring

1 INTRODUCTION

The relationship between run-time performance and effective use of a machine's register set is well understood. In a compiler, the process of deciding which values to keep in registers at each point in the generated code is called *register allocation*. Values in registers can be accessed more quickly than values in memory – on high-performance, microprocessor-based machines, the difference in access time can be an order of magnitude. Thus, register allocation has a strong impact on the run-time performance of the code that a compiler generates. Because memory latencies are rising while register latencies are not, the impact of allocation on performance is increasing. In addition, features like superscalar instruction issue increase a program's absolute demand for registers – if the machine issues two instructions in a single cycle, it must have two sets of operands ready and in place at the start of the cycle. This naturally increases the demand for registers.

Popular techniques for performing register allocation are based on a graph coloring paradigm. These allocators construct a graph representing the constraints that the allocator must preserve. Using graph coloring techniques, they discover a mapping from values in the procedure to registers in the target machine; the mapping must observe the constraints. The first graph coloring allocator was built by Chaitin and his colleagues [8]. Another approach, called *priority-based coloring*, was described by Chow and Hennessy [10, 11]. These two techniques have formed the core around which a rich literature has emerged (see Section 7).

The techniques used in building graph coloring allocators can be improved. In recent years, several important extensions to Chaitin's basic techniques have appeared [2, 28]. Nevertheless, problems remain. In practice, most of these problems appear as either over-spilling or a poor spill choice. In the former case, the allocator fails to keep some value in a register, even though a register is available throughout its lifetime. In the latter case, the allocator chooses the "wrong" value to keep in a register at some point in the code.

Preliminary versions of these results have appeared in the SIGPLAN *Conference on Programming Language Design and Implementation* in 1989 and 1992.
This research has been supported by IBM Corporation and by DARPA through ONR Grant N00014-91-J-1989.
Authors' address: Department of Computer Science, Rice University, Houston, TX 77251–1892.

This paper presents two improvements to existing techniques for register allocation via graph coloring. The improvements address specific problems that we have observed in our use of both Chaitin-style and Chow-style allocators.

2 BACKGROUND

The notion of abstracting storage allocation problems to graph coloring dates from the early 1960's [26] (see Section 7). The first implementation of a graph coloring register allocator was done by Chaitin and his colleagues in the PL.8 compiler [8, 6]. Chow and Hennessy later described a priority-based scheme for allocation based on a coloring paradigm [10, 11]. Almost all subsequent work on coloring-based allocation has followed from one of these two papers. Our own work follows Chaitin's scheme.

Any discussion of register allocation will contain several implicit assumptions. For our work, we assume that the allocator works on low-level intermediate code or assembly code. The code has been shaped by an optimization phase. Before allocation, the code can reference an unlimited number of registers. We call these "pre-allocation" registers *virtual registers*. The allocator does not work with the virtual registers; instead, it works with the distinct *live ranges* in a procedure. A single virtual register can have several distinct values that are interesting in different parts of the program – each of these values will become a separate live range. The allocator discovers all the separate live ranges and allocates them to physical registers on the target machine.

To model register allocation as a graph coloring problem, the compiler first constructs an *interference graph* G. The nodes in G correspond to live ranges and the edges represent *interferences*. Thus, there is an edge in G from node i (live range l_i) to node j if and only if l_i *interferes* with l_j; that is, they are simultaneously live at some point and cannot occupy the same register.[1] The live ranges that interfere with a particular live range l_i are called neighbors of l_i in the graph; the number of neighbors is the *degree* – denoted l_i°.

An attractive feature of Chaitin's approach is that machine-specific constraints on register use can be represented directly in the graph [8]. Thus, the graph represents both the constraints embodied in the program and those presented by the target architecture in a single, unified structure. This is one of the key underlying insights in graph coloring allocators: the interference graph represents *all* the constraints.

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 adjacent nodes always have distinct colors. If we choose k to match the number of machine registers, then we can map a k-coloring of G into a feasible register assignment for the underlying code. Because graph coloring is NP-complete, the compiler uses a heuristic method to search for a coloring; it is not guaranteed to find a k-coloring for all k-colorable graphs. If a k-coloring is not discovered, some values are *spilled;* that is, the values are kept in memory rather than in registers.

Spilling one or more live ranges creates a new and different interference graph. The compiler proceeds by iteratively spilling some live ranges and attempting to color the resulting new graph. In practice, a Chaitin-style allocator rarely requires more than three trips through this loop. Figure 1 illustrates a Chaitin-style allocator. It proceeds in seven phases.

1. *Renumber* systematically renames live ranges. It creates a new live range for each definition point. At each use point, it unions together the live ranges that reach the use. Our implementation models this as an example of the classical disjoint set union-find problem. (In the papers on the PL.8 compiler, this type of analysis is called "getting the right number of names"[8]. The Hewlett-Packard Precision Architecture compiler papers refer to this type of analysis as "web analysis"[23].)

[1] See Chaitin *et al.* for a complete discussion of interference [8].

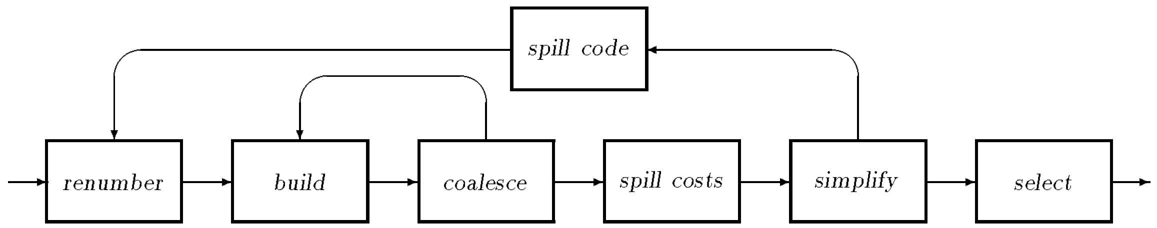

Figure 1 Chaitin's Allocator

2. *Build* constructs the interference graph. Our implementation closely follows the published descriptions of the PL.8 allocator [8, 6]. The interference graph is simultaneously represented as a bit-matrix and as a collection of adjacency lists.

3. *Coalesce* attempts to shrink the number of live ranges. Two live ranges l_i and l_j are combined if the initial definition of l_j is a copy from l_i and they do not otherwise interfere. Combining the two live ranges eliminates the copy instruction. We denote the new live range l_{ij}.

   When the allocator combines l_i and l_j, it can construct an imprecise but conservative approximation to the set of interferences for l_{ij}. The conservative update lets the allocator batch together many combining steps. It performs all the coalescing possible with the update, then repeats both *build* and *coalesce* if coalescing has changed the graph.

4. *Spill costs* estimates, for each live range, the run-time cost of any instructions that would be added if the item were spilled. Spill cost is estimated by computing the number of loads and stores that would be required to spill the live range, with each operation weighted by $c \times 10^d$, where c is the operation's cost on the target architecture and d is the instruction's loop-nesting depth.

5. *Simplify* constructs an ordering of the nodes. It creates an empty stack, then repeats the following two steps until the graph is empty:

   (a) If there exists a node l_i with $l_i^\circ < k$, remove l_i and all of its edges from the graph. Place l_i on the stack for coloring.

   (b) Otherwise, choose a node l_i to spill. Remove l_i and all of its edges from the graph. Mark l_i to be spilled.

   If any node is marked for spilling, the allocator inserts spill code (see 7 below) and repeats the allocation process. If no spilling is required, it proceeds to *select* (see 6 below).

6. *Select* assigns colors to the nodes of the graph in the order determined by *simplify*. It repeats the following steps until the stack is empty:

   (a) pop a live range from the stack,

   (b) insert its corresponding node into *G*, and

   (c) give it a color distinct from its neighbors.

7. *Spill code* is invoked if *simplify* decides to spill a node. Each spilled live range is converted into a collection of tiny live ranges by inserting loads before uses and stores after definitions, as required.

To understand why this works, consider the actions of *simplify* and *select*. *Simplify* only moves l_i from the graph to the stack if $l_i^\circ < k$. Any live range that meets this condition is trivially colorable – that is, it will receive a color independent of the colors assigned to its neighbors. Thus, *simplify* only removes a node when it can prove that the node will get assigned a color. As each live range is removed, the degrees of its neighbors are lowered. This, in turn, may prove that they can be assigned colors.

Select assigns colors to the nodes in reverse order of removal. Thus, it colors each node in a graph where it is trivially colorable; *simplify* ordered the stack so that this must be true. In one sense, the ordering colors the most constrained nodes first – l_i gets colored before l_j precisely because *simplify* proved that l_j was colorable independent of the specific color chosen for l_i.

As an example, consider finding a three-coloring for the "simple graph" shown in Figure 2. First, *simplify* removes all the nodes; it does not need to spill any of them. One possible sequence of removals is a, c, b, d, e. Next, *select* reinserts nodes into the graph, assigning colors. Node e is inserted first and can be given any color, say *red*. Next, d is added; it can get any color except *red*. Running through the entire stack might result in the assignment: $e \leftarrow$ *red*, $d \leftarrow$ *blue*, $b \leftarrow$ *green*, $c \leftarrow$ *red*, and $a \leftarrow$ *blue*.

The decision to spill a live range is made in *simplify*. When the allocator cannot find a node that is trivially colorable, it selects a node to spill. The metric for picking spill candidates is important. Chaitin suggests choosing the node with the smallest ratio of spill cost divided by current degree [6].

Chaitin's heuristic is not guaranteed to find the minimal coloring nor can it be guaranteed to find a k-coloring if it exists; after all, graph coloring is NP-complete. For example, suppose we want to find a two-coloring of the "diamond graph" shown in Figure 2. Clearly, one exists; for example $w \leftarrow$ *red*, $x \leftarrow$ *blue*, $y \leftarrow$ *blue*, and $z \leftarrow$ *red*. Applying *simplify* to the diamond graph presents an immediate problem; there are no nodes with degree less than two. Thus, some node is selected for spilling. If all spill costs are equal, the allocator will make an arbitrary choice; for example, x. After x is removed from the graph and marked for spilling, *simplify* will remove the remaining three nodes without further spilling. Since a node was marked to spill, the allocator must insert spill code, rebuild the interference graph, and try again.

Despite the existence of examples like the diamond graph, Chaitin's technique produces good allocations in practice. Several factors contribute to its success. Allocation is based on global information in the form of a precise interference graph. It includes a powerful mechanism to remove unneeded copies – coalescing. Finally, it uses spill costs to guide the generation of spill code; those spill costs encode simple information about the control-flow graph. Any improvements to Chaitin's work should retain these properties.

3 OPTIMISTIC COLORING

As part of the ParaScope programming environment, we built an optimizing compiler for FORTRAN running on uniprocessors [12]. The initial implementation included a register allocator that used Chaitin's technique as described in Section 2. The allocator worked well and seemed to produce satisfactory allocations. It

Figure 2 Example Graphs

```
subroutine SVD(M, N, ...)
do I = 1, N
  do J = 1, M
    A(I, J) = B(I, J)
  enddo
enddo

do
  many deeply-nested loops
enddo

do
  ...
enddo

do
  ...
enddo

do
  ...
enddo
```

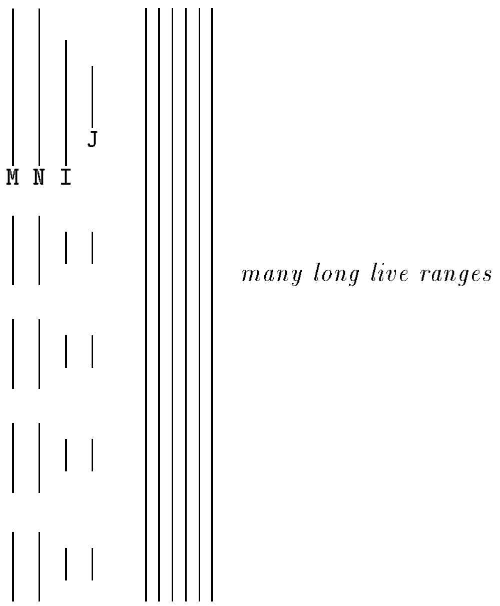

Figure 3 The Structure of SVD

required modest amounts of time and space at compile time. However, as we debugged other parts of the compiler, we discovered several cases where it produced obviously flawed allocations.

3.1 A Motivating Problem

A particularly interesting case arose in the code generated for the singular value decomposition (SVD) of Golub and Reinsch. The actual code was from the software library distributed with Forsythe, Malcolm, and Moler's book on numerical methods [18]. It has two hundred fourteen lines of code, excluding comments. The code contains thirty-seven DO-loops organized into five different loop nests. The first loop nest is a simple array copy; four larger and more complex loop nests follow. Figure 3 shows its structure.

In SVD, the allocator spilled many short live ranges in deference to the larger, longer live ranges. In the array-copy loops, it spilled the loop indices and limits even though several registers were unused in the loop. After some study, we understood both why the register allocator over-spilled so badly and what situations provoked this behavior.

After optimization, about a dozen long live ranges extend from the initialization portion of the code down into the large loop nests. As the figure shows, the long live ranges span the small array-copy loops. Because they span so much of the code, they all have high degree in the interference graph. Additionally, they have large spill costs because they are referenced often inside deeply nested loops. They restrict the graph so much that the allocator must spill some live ranges.

Initially, the allocator chooses to spill the indices and limits on the array-copy loops. This choice is correct; these values have smaller estimated spill costs than the longer live ranges; the allocator *should* choose them first. Unfortunately, spilling them does not help; the problem is in the later loop nests. To alleviate the demand for registers in the large loop nests, the allocator must spill more values. As it proceeds, it eventually spills most of the longer live ranges. As a result, there are unused registers in the array-copy loops (and the indices and limits are kept in memory).

3.2 An Improvement

Knowing that the allocator over-spilled on both the diamond graph and the SVD, we reconsidered the allocation process. Each of the examples highlights a different problem.

1. The diamond graph is two-colorable; we can see that by inspection. The allocator fails because the approximation that it uses to decide whether x will get a color is too weak.

   In looking for a k-coloring, the allocator approximates "x gets a color" by "x has degree less than k." This is a sufficient condition for x to get a color but not a necessary condition. For example, x may have k neighbors, but two of those neighbors may get assigned the same color. This is precisely what happens in the diamond graph.

2. In SVD, the allocator must spill some live ranges. The heuristic for picking a spill candidate selects the small live ranges used in shallow loop nests because they *are* less expensive to spill. Unfortunately, spilling them is not productive – it does not alleviate register pressure in the major loop nests.

   At the time that the allocator makes the decisions, it cannot recognize that the spills will not help. Similarly, the allocator cannot retract the decisions later. Once spilled, a live range stays spilled.

The coloring heuristic explored by Matula and Beck finds a two-coloring for the diamond graph [27]. Their algorithm differs only slightly from Chaitin's approach. To simplify the graph, they repeatedly remove the node of smallest current degree, versus Chaitin's approach of removing *any* node n where $n^\circ < k$. After all nodes have been removed, they select colors in the reverse of the order of deletion, in the same fashion as Chaitin.

On the diamond graph, this heuristic generates a two-coloring. Chaitin's heuristic fails because it pessimistically assumes that all the neighbors of a node will get different colors. Matula and Beck's heuristic discovers a two coloring because it can capitalize on the case when two neighbors receive the same color.

Unfortunately, Matula and Beck's scheme simply finds *a* coloring; there is no notion of finding a k-coloring for some particular k and no mechanism for producing spill code. In real programs, this is a serious problem. Many procedures require spill code – their interference graphs are simply not k-colorable. For example, the SVD routine must spill some live ranges; an optimal coloring would not eliminate all spills.

We wanted an algorithm that combined Matula and Beck's stronger coloring heuristic with Chaitin's mechanism for cost-guided spill selection. To achieve this effect, we made two modifications to Chaitin's original algorithm:

1. In *simplify,* the allocator removes nodes that have degree less than k in arbitrary order. Whenever it discovers that all the remaining nodes have degree greater than k, it chooses a spill candidate. That node is removed from the graph; but instead of marking it for spilling, *simplify* optimistically pushes it on the stack, hoping a color will be available in spite of its high degree. Thus, it removes nodes in the same order as Chaitin's allocator, but spill candidates are included on the stack for possible coloring.

2. In *select,* the allocator may discover that no color is available for some node. In that case, it leaves the node uncolored and continues with the next node. Any uncolored node must be a node that Chaitin's method would spill. To see this, consider the graph in which each node is colored. In both methods, a node n is inserted into the graph from which it was removed. Chaitin's method guarantees that a color is available for n. Our method guarantees this property for all nodes except spill candidates. Thus, an uncolored node must be a spill candidate; that is, a node that Chaitin would have spilled.

   If all nodes receive colors, the allocation has succeeded. If any nodes are uncolored, the allocator inserts spill code for the corresponding live ranges, rebuilds the interference graph, and tries again.

The resulting allocator is shown in Figure 4. We call it an *optimistic* allocator. The decision to insert spill code now occurs in *select,* rather than in *simplify*. The rest of the allocator is unchanged from Chaitin's scheme. In this form, the allocator can handle both of the problems described at the head of this section.

Deferring the spill decision has two powerful consequences. First, it eliminates some non-productive spills. In Chaitin's scheme, spill decisions are made during *simplify,* before any nodes are assigned colors. When it selects a node as a spill candidate, that live range is spilled. In our scheme, spill candidates get placed on the stack with all the other nodes. Only when *select* discovers that no color is available is the live range actually spilled. This mechanism, in effect, allows the allocator to reconsider spill decisions.

Second, late spilling capitalizes on specific details of the color assignment to provide a stronger coloring heuristic. In selecting a color for node x, it examines the colors of all x's current neighbors. This provides a direct measure of "does x get a color?" rather than estimating the answer with "is $x^\circ < k$?" In particular, if two or more of x's neighbors receive the same color, then x may receive a color even though $x^\circ \geq k$. The optimistic allocator finds a two-coloring for the diamond graph.

Recall SVD. The live ranges for `I`, `J`, `M`, and `N` are early spill candidates because their spill costs are small. However, spilling them does not alleviate register pressure inside the major loop nests. Thus, the allocator must spill some of the large live ranges; this happens after the small live ranges have been selected as spill candidates and placed on the stack. When the small live ranges come off the stack in *select,* some of these large live ranges have been spilled. The allocator can easily determine that colors are available for these small live ranges in the early array-copy loops.

Optimistic coloring is a simple improvement to Chaitin's pessimistic scheme. Assume that we have two allocators, one optimistic and one pessimistic, and that both use the same spill metric – for example, Chaitin's metric of spill cost divided by current degree. The optimistic allocator has a stronger coloring heuristic, in the following sense: it will color any graph that the pessimistic allocator does and it will color some graphs that the pessimistic allocator will not. If spilling is necessary, the optimistic allocator will spill a subset of the live ranges spilled by the pessimistic allocator.

Optimistic coloring helps generate better allocations. In a few cases, this eliminates all spilling; the diamond graph is one such example. In many cases, the cost of spilling is reduced; that is, the procedure executes fewer cycles due to register spilling. Section 5 presents dynamic measurements of the improvement. An implementation of this technique in the back-end of the IBM XL compiler family for the RS/6000 architecture resulted in a decrease of about twenty percent in estimated spill costs over the SPEC benchmark suite [22].

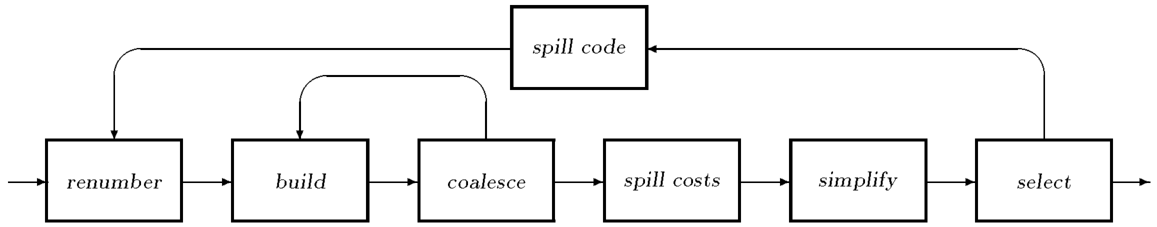

Figure 4 The Optimistic Allocator

4 REMATERIALIZATION

Even with optimistic coloring, the allocator must spill some live ranges. When this happens, the allocator should choose the least expensive mechanism to accomplish the spill. In particular, it should recognize cases where it is cheaper to recompute the value than to store and retrieve it from memory. Consider the code fragments shown in Figure 5 (the notation $[p]$ means "the contents of the memory location addressed by p").

Source Note that p is constant in the first loop, but varying in the second loop. The register allocator should take advantage of this situation.

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

Chaitin This 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.

Splitting The final column shows code we would expect from a "splitting" allocator [11, 25, 21, 5]; 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 [3]. 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.

This section divides into two major subsections. The first presents a conceptual view of our approach to rematerialization. The second discusses the actual implementation in our allocator.

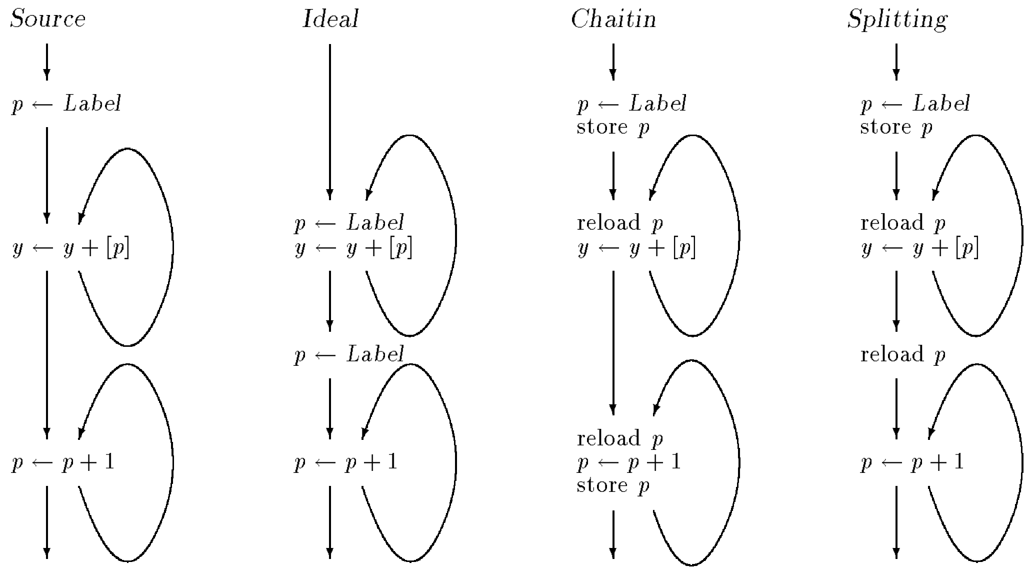

Figure 5 Rematerialization versus Spilling

4.1 The Ideas

Chaitin *et al.* discuss several ideas for improving the quality of spill code [8]. 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 [8]. 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 5, 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 a live range with a single value, but cannot handle more complex cases, like the variable p in Figure 5. We have extended Chaitin's work to take advantage of rematerialization opportunities for complex, multi-valued live ranges. Our method tags 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. tag each value with rematerialization information, and
3. form new live ranges from connected values having identical tags.

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

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 [13]. 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 graph, with dead ϕ-nodes eliminated [9].

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.

Propagating Rematerialization Tags

To propagate tags, we use an analog of Wegman and Zadeck's *sparse simple constant* algorithm [31].[2] We modify their lattice slightly to represent the necessary rematerialization information. The new lattice

[2]The more powerful *sparse conditional constant* algorithm is unnecessary; earlier optimization has eliminated any control-flow paths that it would detect as non-executable.

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

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

$$\begin{array}{ccccl} \text{any} & \sqcap & \top & = & \text{any} \\ \text{any} & \sqcap & \perp & = & \perp \\ inst_i & \sqcap & inst_j & = & inst_i \quad \text{if } inst_i = inst_j \\ inst_i & \sqcap & inst_j & = & \perp \quad \text{if } inst_i \neq inst_j \end{array}$$

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 $\perp$. Values defined by a copy instruction will have their tags *lowered* to *inst* or $\perp$, 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 $\perp$.

This process tags each value in the SSA graph with either an instruction or $\perp$. If a value's tag is $\perp$, 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 – a lightweight spill. The tags are used in two later 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.

Inserting Splits

After propagation, the ϕ-nodes must be removed and values renamed to recreate an executable program. Consider the example in Figure 6. The *Source* column simply repeats the example introduced in Figure 5. 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 $\perp$) 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 4.2. Of course, the allocator could insert *all* the copies suggested in the *Splits* column as a form of *live range splitting.* This idea has been discussed elsewhere [3].

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 6. If neither p_0 nor p_{12} are spilled and 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. We cannot simply use *coalesce;* it would remove *all* of the copies, losing the desired separation between values with different tags. Therefore, we use a pair of limited coalescing mechanisms to remove unproductive copies:

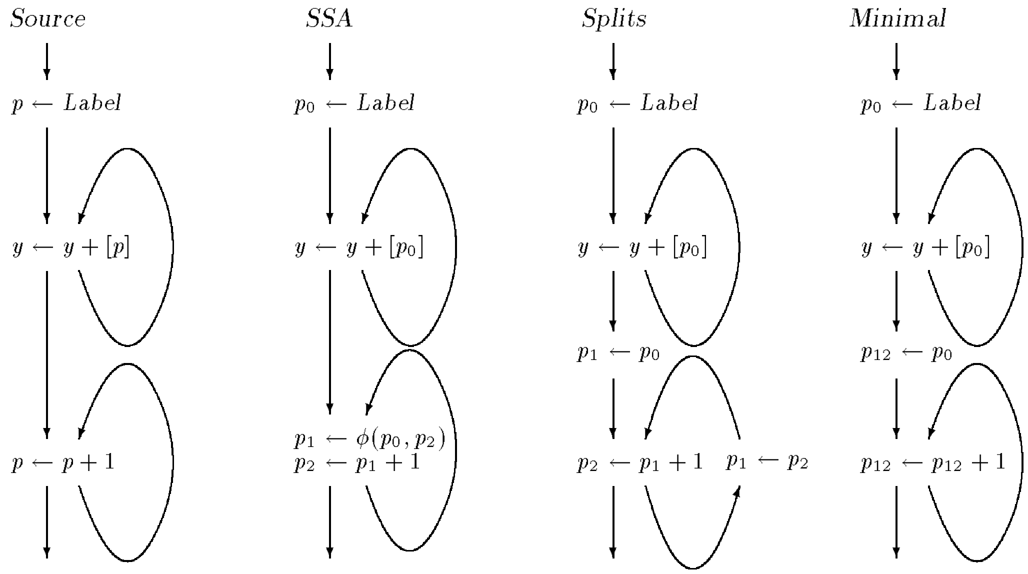

Figure 6 Introducing Splits

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.

4.2 The Implementation

Chaitin-style allocators can be extended naturally to accommodate our approach. The high-level structure depicted in Figure 4 is unchanged, but several low-level modifications are required. The next sections discuss the enhancements required in *renumber, coalesce,* and *select.*

Renumber

Chaitin's version of *renumber* was based on def-use chains [8]. Long before our interest in rematerialization, we adopted an implementation strategy for *renumber* based on the pruned SSA graph. Conceptually, that implementation has four steps:

1. Determine liveness at each basic block using a sparse data-flow evaluation graph [9].
2. Insert ϕ-nodes based on dominance frontiers [13]. 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 [8].
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 4.1.
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.

Conservative Coalescing

To prevent *coalesce* from removing the splits so 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 [6]. 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; we have not measured this effect. If intermixing conservative coalescing with unrestricted coalescing does not produce worse allocations, it would simplify and speed 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, conservative coalescing undoes all splitting. However, it cannot undo all the non-productive splits by itself.

Biased Coloring

The second mechanism for removing useless splits involves changing the order in which colors are considered for assignment. As *renumber* inserts splits, it marks pairs of values connected by a split as *partners*. 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

We have implemented these techniques in our optimizing compiler for FORTRAN. The compiler has experimental code generators for the RT/PC, the Sparc, the i860, and the RS/6000. To experiment with register allocation, we have built a series of allocators that are independent of any particular architecture [3]. 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 can 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 can perform our tests in a machine-independent fashion, potentially using a variety of register sets.

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 7 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 simple instrumentation appearing immediately after several of the statements. Of course, this sample is very simple, but the majority of ILOC is no more complex.

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

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

Figure 7 ILOC and C

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 Results

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 [18]. They are grouped into seven programs with simple drivers. The remaining fifty-nine routines are from the SPEC benchmark suite [30]. 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.

Our results are presented as a sequence of comparisons in two tables. 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. These costs are calculated from dynamic counts of instructions as described

		Cycles of Spill Code		Percentage Contribution					
program	*routine*	*Original*	*Optimistic*	*load*	*store*	*copy*	*ldi*	*addi*	*total*
fmin	fmin	551	370	16	16		0		32
seval	spline	125	117	5	2				6
solve	decomp	362	305	10	6	0			16
svd	svd	2,509	1,977	16	7	−2			21
doduc	colbur	25	19	8	8		8		24
	dcoera	29	15	21	21		7		48
	ddeflu	443	335	13	12	−0			24
	debflu	1,939	1,131	20	20				42
	debico	463	459	0	0				1
	deseco	5,500	4,957	4	2	−1	3	0	10
	drepvi	252	218	7	6				14
	ihbtr	452	400	6	6				12
	inithx	714	579	11	4		4		19
	integr	526	502				5		5
	lectur	257	221	11	2		2		14
	paroi	1,780	1,433	9	6			5	20
	prophy	1,954	1,531	13	9			0	21
	repvid	651	599	4	4				8
	supp	146	149	−1	−1		0		−2
fpppp	d2esp	51	35	16	16	−2	2		31
	efill	173	94	21	23		2		46
	fpppp	1,472	1,444	1	1				2
	twldrv	13,731,802	11,311,624	12	7	0			18
matrix300	lbmk14	136	132	1	1				3
	sgemm	12,321	9,905	10	10				20
	sgemv	3,027	1,808	40	0	−0			40
tomcatv	tomcatv	394,397,732	367,995,733	3	3	−0			7

Table 1 Effects of Optimistic Coloring

earlier. The last column (*total*) gives the percentage improvement in spill costs from the old allocator to the new one – 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. Each table shows only routines where a difference was observed.[3]

Consider the first row in Table 1. This row presents results for the routine `fmin` from the program `fmin`. The allocator using Chaitin's heuristic generated an allocation requiring 551 cycles of spill code; the optimistic allocator required only 370 cycles. 16% of the savings came from having to execute fewer loads and 16% arose from fewer stores. There was a further insignificant improvement due to executing fewer load-immediates. The total improvement was 32%.

[3]The raw data for the tables is given in Appendix A of Briggs' thesis [3].

Optimistic Coloring

Table 1 shows a comparison of test results for our allocator using two different coloring heuristics. The column labeled *Original* gives data for a version using Chaitin's coloring heuristic; the column labeled *Optimistic* gives data for a version using our optimistic coloring heuristic.

In our test suite of 70 routines, we measured improvements in 26 cases and a single loss (an extra load and store were required). Improvements ranged from tiny to quite large, sometimes reducing spill costs by over 40%. The single loss was disappointing, since we have claimed that the optimistic coloring heuristic can never spill more than Chaitin's heuristic. However, we must recall the structure of the allocator. After each attempt to color, spill code is inserted and the entire *build-coalesce-color* process is repeated. The optimistic coloring heuristic will perform at least as well as Chaitin's heuristic on any graph; but after spilling, the two allocators will be facing different problems.

Rematerialization

Table 2 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.

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 allocations come at a cost in compile-time. In the case of the optimistic allocator, the coloring phase can be more expensive due to the need to attempt a coloring even if spill candidates are discovered. Support for rematerialization requires several extra steps. An extra pass over the code is required to initialize rematerialization tags before propagation 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 can eliminate some copies during *renumber* (recall step 5 in Section 4.2).

Table 3 shows comparative timings for the three 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 were averaged. The first column shows the phase of allocation. *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. *Color* includes both *simplify* and *select.* Note that **tomcatv** required extra rounds of spilling with the original allocator and the optimistic allocator. For each routine, the *Orig.* column gives times required by the allocator with Chaitin's original coloring heuristic, the *Opt.* column gives the times required by the optimistic allocator, and the *Remat.* column gives times required by the optimistic allocator with improved rematerialization.

We selected three routines to illustrate performance over a range of sizes. The first routine is **repvid**,

		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 2 Effects of Rematerialization

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 3 is that support for rematerialization can require a small amount of additional compile-time. On the other hand, the optimistic coloring heuristic has very little cost and is sometimes faster than Chaitin's pessimistic heuristic.

The results in Table 3 also illuminate a number of interesting details about the behavior of all three allocators.

- The initial 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 *Remat.*, reflecting the cost of propagating rematerialization tags.

	`repvid`			`tomcatv`			`twldrv`		
Phase	*Orig.*	*Opt.*	*Remat.*	*Orig.*	*Opt.*	*Remat.*	*Orig.*	*Opt.*	*Remat.*
cfa	.00	.00	.00	.00	.00	.01	.01	.01	.01
renum	.04	.03	.04	.06	.06	.07	.56	.56	.62
build	.16	.16	.16	.38	.38	.42	9.88	9.84	8.47
costs	.01	.01	.01	.02	.02	.02	.14	.16	.14
color	.01	.02	.03	.03	.04	.06	1.12	1.08	1.61
spill	.01	.01	.01	.02	.02	.02	.18	.16	.16
renum	.02	.02	.02	.02	.02	.03	.08	.10	.14
build	.06	.06	.06	.09	.09	.12	.62	.61	.81
costs	.01	.01	.01	.01	.01	.01	.07	.07	.06
color	.01	.01	.02	.01	.02	.04	.11	.14	.31
spill	.01	.00	.01	.09	.01	.01	.04	.03	.04
renum	.02	.01	.02	.02	.02	.03	.09	.10	.14
build	.04	.04	.06	.05	.05	.09	.40	.59	.80
costs	.01	.01	.01	.01	.01	.01	.07	.07	.06
color	.01	.01	.02	.02	.02	.04	.19	.15	.31
spill				.01	.00	.01			
renum				.02	.02	.03			
build				.06	.05	.09			
costs				.01	.01	.01			
color				.01	.02	.04			
spill				.01					
renum				.02					
build				.05					
costs				.01					
color				.02					
total	.40	.39	.47	.97	.85	1.13	13.55	13.65	13.67

Table 3 Allocation Times in Seconds

- In most cases, the cost of the *build-coalesce* loop is higher for *Remat.* due to the additional passes of *conservative coalescing.*
- The very low cost of control-flow analysis illustrates the speed and practicality of the algorithm for calculating dominance frontiers [13].
- 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 the 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 [3]. Of course, these speeds are not competitive with the fast, local techniques used in non-optimizing compilers [19, 20]; however, we believe that global optimizations require global register allocation.

6 IMPLEMENTATION INSIGHT

To perform the experiments described in Section 5, we implemented several versions of our allocator. Naturally, we gained some insight into the implementation of Chaitin-style graph coloring allocators. This section attempts to convey those ideas in a concise form. We hope that it proves useful to other implementors. When possible, we label an insight with the particular phase of the allocator that it affects.

Constant values The rematerialization scheme suggests that the compiler should represent constants in a way that makes them visible to the allocator. In our current compiler, integer constants are obvious in the code; they are loaded using an integer load immediate instruction. Non-integer constants are stored in a static constant pool and loaded using a fixed offset from the static data area pointer. This effectively hides them from rematerialization. Changing our intermediate representation to make these values visible would expose them to rematerialization, improving the quality of the final code.

Build Like Chaitin, we advocate building two representations of the interference graph: both a bit matrix and a set of adjacency vectors. Many implementations use a single representation and claim a space savings. However, both forms of the graph are needed for efficient execution of the allocator.

In practice, we have not had problems with an explosion in the size of the interference graph. Our allocator runs in a reasonable amount of space. For example, the interference graph of `tomcatv` requires 146,404 bytes while the graph for `twldrv` requires 3,357,640 bytes [3]. Usually, the bit vectors for an iterative solution of the live variables problem are as large as our interference graphs. (Actually, this fact led us to the sparse evaluation graph techniques of Choi, Cytron, and Ferrante [9].)

Coalescing Coalescing is not transitive. Even though l_i and l_j do not interfere and l_i and l_k do not interfere, l_{ij} will interfere with l_k if l_j interferes with l_k. Thus, the order of coalescing is important.

Coalescing should proceed from innermost loops outward. Inside-out coalescing ensures that copies in more deeply nested loops get removed before copies in outer loops. In practice, this makes a noticeable difference on many routines.

Conservative coalescing and biased coloring There is a direct trade-off between the strength of conservative coalescing and the effectiveness of biased coloring. In practice, we experimented with several different restrictions on coalescing. More precise (*i.e.,* less conservative) restrictions on coalescing increase the set of live ranges that it can combine; this decreases the number of live ranges combined by biased coloring. It is unclear whether this occurs because the more precise coalescing condition restricts freedom in the graph or because it combines live ranges that biased coloring also will catch. In any case, biased coloring cannot completely eliminate the need for conservative coalescing.

Computing spill costs In our allocator, *spill costs* is always executed. Chaitin's allocator waits until it must choose the first value to spill [7]. Each approach has its merits.

Some live ranges have negative spill costs. This suggests that it is less expensive to store and reload them than to keep them in registers. Our allocator computes spill costs early and aggressively removes live ranges with negative spill cost.

Chaitin defers the computation of spill costs until the allocator recognizes that it must spill. This speeds allocation on procedures that color without spilling. Additionally, the allocator can skip the spill-cost computation for live ranges that it has already removed from the graph.

Simplify Rematerialization fundamentally changes the allocator. In other Chaitin-style allocators, live ranges are unrelated. The order of removal for trivially colored nodes has little noticeable effect on the results of allocation. The introduction of splits to isolate never-killed values and the use of biased coloring change that fact. This gives a new importance to the relative removal order of trivially colored nodes. When faced with multiple nodes that can be removed, *select* should first remove nodes that have no partners. This forces the nodes with partners to be colored first. They are colored in a less constrained graph. In practice, this increases the effectiveness of biased coloring.

Picking spill candidates Introducing splits to support rematerialization increases the number of nodes in the interference graph. This can adversely effect the running time of *simplify* by increasing the time spent searching for spill candidates. As described in Section 2, the search for a spill candidate must examine each node remaining in the current graph. A naive implementation repeats this process for each spill because the spill metric includes the node's current degree. Thus, choosing spill candidates takes $\mathbf{O}(k \cdot N)$ time, where k is the number of spills and N is the number of live ranges in the original graph.

Undoubtedly, some careful algorithmic work can improve this situation. In practice, we refactored the comparison by noting that

$$\frac{cost(l_i)}{l_i^\circ} \geq \frac{cost(l_j)}{l_j^\circ} \quad \textit{if and only if} \quad cost(l_i) \times l_j^\circ \geq cost(l_j) \times l_i^\circ .$$

Replacing the floating-point divisions with integer multiplies improved the constant factor.

Limited backtracking *Select* assigns colors to nodes in a somewhat arbitrary manner. Biased coloring capitalizes on this fact by carefully choosing the order in some cases where it matters. Nonetheless, improved color choice can avoid some spills. In the optimistic allocator, we can add a limited form of backtracking. When no color is available for node p, *select* can consider recoloring one of p's neighbors to open up a color for p. Such backtracking must be carefully constrained to avoid a combinatorial explosion.

While examining p's neighbors to determine available colors, we can accumulate the number of uses of each color and note which neighbor uses each color. If no color remains for p and one of the colors is used by only one of p's neighbors, *select* can try to recolor that neighbor. If it succeeds, a color is available for p.

Limited backtracking is easy to implement. It takes very little compile time. It rarely leads to degradation.[4] On the other hand, it rarely leads to a significant improvement [3]. It also fits into biased coloring; however, experience suggests that *limited lookahead* (see Section 4.2) produces more consistent improvement than *limited backtracking* in the biased scheme.

NP-noise In testing, we have often stumbled on a phenomenon that we label NP-noise. When measuring the output of a process that involves heuristic solution of an NP-complete problem, the answers often contain behavior that seems anomalous. Sometimes the heuristic must pick from a set of choices that have the same local cost. Different choices can lead to different answers.

In the allocator, we use a linear-time heuristic to find the coloring. From a decision tree perspective, it picks a path through a tree in time proportional to the height of the tree. Of course, the tree has an exponential number of nodes. At each step, it picks based on local information – the history of previous choices and the remaining nodes. Changing a local choice can lead to a different global result. Thus, simple things like the ordering of nodes, how ties are broken, and actual color assignments produce variations in the output.

These variations make it difficult to compare two competing allocators. For example, in Table 2, our rematerialization scheme produced worse code than the optimistic allocator with Chaitin's scheme for two of the routines, `colbur` and `ddeflu` in `doduc`. A heuristic that, in general, has better global behavior, can produce anomalous results in specific cases.

This effect may, in part, account for the effects seen by the Haifa group with their "best of three" spilling scheme [2]. They smooth out some of the NP-noise by using three different spill metrics and taking the best result.

[4] We have never seen it produce a worse allocation, though such situations are conceivable. It is often better to save an expensive spill now instead of possibly saving less-expensive spills in the future.

7 OTHER WORK

The relationship between economical use of memory and graph coloring has been discussed for a long time. The first paper on this subject appears to be Lavrov's 1961 paper on minimizing memory use [26]. He suggests building an *inconsistency* graph and coloring it; he does not propose a practical method for finding a coloring. Ershov and his colleagues built on this work in the ALPHA project. They solved storage allocation problems by building an interference graph and using a packing algorithm on it [14, 15]. By the late sixties, Cocke was clearly talking about applying these insights directly to register allocation; both Kennedy and Schwartz credit him with this insight [24, 29].

This early work on graph coloring register allocation emphasizes the coloring problem with little consideration for the questions of spill choice and placement. Algorithms by Cocke and Ershov (as reported by Schwartz [29]) are concerned exclusively with minimizing the number of colors required. There is no discussion of spill code and the flow graph is ignored entirely.

The first complete register allocator based upon graph coloring is described by Chaitin and his colleagues [8]. Spilling is handled by a variety of heuristics, some based upon an interval analysis of the flow graph. Unfortunately, these *ad hoc* techniques are expensive and not always effective. In a subsequent paper, Chaitin introduces a simpler technique that attempts to solve the spilling problem based on the interference graph and spill-cost estimates for each live range [6]. Section 2 describes this allocator.

Chow and Hennessy describe a priority-based coloring scheme [10, 11]. In their work, values initially reside in memory. They divide the register set between local and global allocation. They build an imprecise interference graph and color constrained live ranges in priority order. Their priority function uses spill cost normalized by live range length. When a constrained live range cannot be colored, they split it into smaller pieces and try to color them independently.

Fabri, in the context of her work on minimizing memory storage requirements, introduces a renaming transformation that is analogous to Chow's live range splitting [16]. She notes that, in some cases, splitting improved the resulting coloring. She mentions its applicability to register allocation problems.

Several groups have refined Chow's approach. Larus and Hilfinger make several modifications [25]. In their work, values initially reside in registers. They eliminate local allocation, limit basic block sizes, and use a more sophisticated live range splitting technique. Gupta, Soffa, and Steele describe an approach that partitions the interference graph into subgraphs that are colored individually and later merged [21].

The Hewlett-Packard compiler group describe another priority-based allocator [23]. They assign colors in order of decreasing degree in the interference graph. When they must spill, they examine both the current live range and all those that have been assigned colors and spill the value with minimal cost under their spill metric. Their metric combines Chaitin's notion of spill cost with a term that includes the impact of keeping a value in a register across loops where it is live but not used. While we have not implemented a version of their algorithm, we believe that their scheme will avoid problems like the one we observed with the SVD routine. On the other hand, it will spill some live ranges that Chaitin proves are trivially colored.

Bernstein and his colleagues describe a collection of techniques [2]. They show that a *best-of-three* coloring scheme produces better results than Chaitin's original scheme.[5] They impose a "largest degree first" ordering when removing unconstrained nodes from the graph in *simplify* (Step (5a) in Section 2). They introduce a *cleaning* heuristic to decrease the amount of spill code generated within a single basic block. In our experience, these improvements are orthogonal to those presented in this paper.

Callahan and Koblenz construct a fine-grained hierarchical decomposition of the flow graph, a *tiling* [5]. Coloring is performed for individual tiles and the results are merged in two passes over the tree. Their

[5] See the discussion of NP-noise in Section 6 for one possible explanation of this behavior.

algorithm is explicitly parallel.

Of course, not all global allocators are built on the graph coloring paradigm. Other approaches include the bin-packing allocators built by Digital Equipment Corporation and the probabilistic allocator of Fischer and Proebsting [1, 17]. It is difficult to compare these techniques because the implementations work on different intermediate representations, they follow different optimizers, and the compilers target different machines.

8 SUMMARY

Optimistic coloring is a simple improvement over Chaitin's original allocator. It produces the same allocation as Chaitin's method, except when it improves on the result. The results in Table 1 show that this happens regularly. The costs are nearly identical; occasionally one method will require an extra trip around the main loop for spill code insertion. In another paper, we have shown that *optimistic* allocators generate better allocations on machines that use register pairs for some values [4]. Any implementation of a Chaitin-style allocator should incorporate this improvement; it makes a significant difference in the allocation quality at little or no cost.

Our method of rematerializing *never-killed* values finds the maximum extents for each *never-killed* value. It ensures that the allocator will spill these values in the least expensive way. In procedures where *never-killed* values must be spilled, the result is better code. The results in Table 2 show that this actually happens. When no opportunities for rematerialization exist, the features added for rematerialization do not hurt. Additionally, the algorithms that we describe in Section 4 are both fast and practical. Our experience suggests that asymptotically fast algorithms produce good compile-time performance.

ACKNOWLEDGEMENTS

Ken Kennedy participated in our early discussions on coloring and encouraged this work from the start. Jorge Moré pointed out Matula's algorithm. Mark Krentel suggested that our coloring heuristic was giving up too easily and encouraged us to try more aggressive coloring heuristics. 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] Anklam, P., Cutler, D., Heinen, Jr., R., and MacLaren, M. D. *Engineering a Compiler: VAX-11 Code Generation and Optimization.* Digital Press, Bedford, Massachusetts, 1982.

[2] Bernstein, D., Goldin, D. Q., Golumbic, M. C., Krawczyk, H., Mansour, Y., Nahshon, I., and Pinter, R. Y. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices 24*, 7 (July 1989), 258–263. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[3] Briggs, P. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Apr. 1992.

[4] Briggs, P., Cooper, K. D., and Torczon, L. Coloring register pairs. *ACM Letters on Programming Languages and Systems 1*, 1 (Mar. 1992), 3–13.

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

[6] Chaitin, G. J. Register allocation and spilling via graph coloring. *SIGPLAN Notices 17*, 6 (June 1982), 98–105. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[7] Chaitin, G. J. Register allocation and spilling via graph coloring. United States Patent 4,571,678, Feb. 1986.

[8] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. Register allocation via coloring. *Computer Languages 6*, 1 (Jan. 1981), 47–57.

[9] Choi, J.-D., Cytron, R., and Ferrante, J. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages* (Jan. 1991), pp. 55–66.

[10] Chow, F. C., and Hennessy, J. L. Register allocation by priority-based coloring. *SIGPLAN Notices 19*, 6 (June 1984), 222–232. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[11] Chow, F. C., and Hennessy, J. L. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems 12*, 4 (Oct. 1990), 501–536.

[12] Cooper, K. D., Hall, M. W., Hood, R. T., Kennedy, K., McKinley, K., Mellor-Crummey, J., Torczon, L., and Warren, S. K. The ParaScope parallel programming environment. *Proceedings of the IEEE* (Feb. 1993). (*to appear*).

[13] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct. 1991), 451–490.

[14] Ershov., A. P. Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs. *Doklady Akademii Nauk S.S.S.R. 142*, 4 (1962). English translation in *Soviet Mathematics* 3:163–165, 1962.

[15] Ershov[6], A. P. Alpha – an automatic programming system of high efficiency. *Journal of the ACM 13*, 1 (Jan. 1966), 17–24.

[16] Fabri, J. Automatic storage optimization. *SIGPLAN Notices 14*, 8 (Aug. 1979), 83–91. In *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*

[17] Fischer, C. N., and Proebsting, T. A. Probabilistic register allocation. *SIGPLAN Notices 27*, 7 (July 1992), 300–311. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[18] Forsythe, G. E., Malcolm, M. A., and Moler, C. B. *Computer Methods for Mathematical Computations.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[19] Fraser, C. W., and Hanson, D. R. A retargetable compiler for ANSI C. *SIGPLAN Notices 26*, 10 (Oct. 1991), 29–43.

[20] Fraser, C. W., and Hanson, D. R. Simple register spilling in a retargetable compiler. *Software – Practice and Experience 22*, 1 (Jan. 1992), 85–99.

[21] Gupta, R., Soffa, M. L., and Steele, T. Register allocation via clique separators. *SIGPLAN Notices 25*, 7 (July 1989), 264–274. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[22] Hopkins, M. E. Private communication. Conversation during visit to Rice, Feb. 1991.

[6] The modern spelling is *Ershov*. Older variations include *Yershov* and *Eršov*.

[23] Johnson, M. S., and Miller, T. C. Effectiveness of a machine-level global optimizer. *SIGPLAN Notices 21*, 7 (July 1986), 99–108. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*.

[24] Kennedy, K. *Global Flow Analysis and Register Allocation for Simple Code Structures*. PhD thesis, Courant Institute, New York University, Oct. 1971.

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

[26] Lavrov, S. S. Store economy in closed operator schemes. *Journal of Computational Mathematics and Mathematical Physics 1*, 4 (1961), 687–701. English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3:810-828, 1962.

[27] Matula, D. W., and Beck, L. L. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM 30*, 3 (July 1983), 417–427.

[28] Nickerson, B. R. Graph coloring register allocation for processors with multi-register operands. *SIGPLAN Notices 25*, 6 (June 1990), 40–52. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[29] Schwartz, J. T. On programming: An interim report on the SETL project, Installment II: The SETL language and examples of its use. Tech. rep., Courant Institute, New York University, Oct. 1973.

[30] SPEC release 1.2, Sept. 1990. Standards Performance Evaluation Corporation.

[31] Wegman, M. N., and Zadeck, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*, 2 (Apr. 1991), 181–210.