Coloring Heuristics for Register Allocation

Preston Briggs
Keith D. Cooper
Ken Kennedy
Linda Torczon

Department of Computer ScienceRice University
Houston, Tx 77251-1892

ABSTRACT

We describe an improvement to a heuristic introduced by Chaitin for use in graph coloring register allocation. Our modified heuristic produces better colorings, with less spill code. It has similar compile-time and implementation requirements. We present experimental data to compare the two methods.

1. Introduction

As a part of the programming environment project at Rice, we are building an optimizing compiler for FORTRAN [CCHK 87, CoKT 86]. Currently, the compiler produces code for the IBM RT/PC, a RISC-style processor with sixteen general purpose registers and eight floating-point registers.[1] We expect that the compiler will be easy to retarget to other architectures. Many aspects of our design were influenced by the PL.8 compiler [AuHo 82]; in particular, our front-end and optimizer rely on the code generator doing a good job of global register allocation.

We decided to use a Chaitin-style, graph coloring register allocator for several reasons [CACC 81, Chai 82]. The high quality code produced by the PL.8 compiler was a convincing argument by itself. Even more appealing was the conceptual elegance of the algorithm. After some experience with other, comparatively *ad hoc* methods of register allocation, graph coloring appeared wonderfully simple and clean.

1.1. An Implementation

A graph coloring register allocator is an interesting implementation project. Our first effort, an exploratory version, was finished in about two weeks. The quality of the generated code was good, but the register allocation time was high. Even worse, the allocation time was growing nonlinearly in the size of the routine. Nevertheless, it worked, so we turned our attention to other parts of the compiler.

1.2. A Motivating Problem

As the compiler matured, we began compiling a variety of programs to test the quality of our optimizer and code generator. While most routines needed some spill code, one example (a singular value decomposition routine, SVD) needed an exceptional amount.

At first, we believed the extra spill code was related to the size of the routine. Our assumption was that large routines need more registers. However, when we examined the object code (while looking for an unrelated bug), we found that the register allocator had done an exceptionally poor job in some portions of the routine. In particular, the loop indices and limits of a small doubly-nested loop had been spilled despite the fact that there were several free registers available. After some study, we were able to understand why the register allocator over-spilled so badly and what situations provoked this behavior.

This work has been supported by the National Science Foundation through grants CCR 86-19893 and CCR 87-06229 and by IBM Corporation.

The floating-point registers are actually provided by a co-processor; however, this is transparent to the code generator.

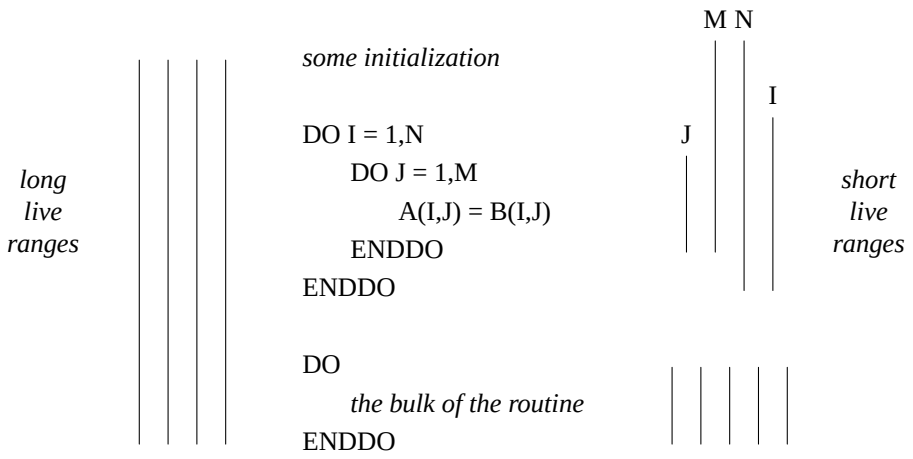

Figure 1 — the structure of SVD

As illustrated in Figure 1, the SVD routine consists of several loop nests. It begins with some initialization code, followed by a small, doubly-nested loop that implements a simple array copy. The rest of the routine consists of three large, complex loop nests that do the bulk of the work. After optimization, there are about a dozen long live ranges extending from the initialization portion, through the array copy, and into the large loop nests. During coloring, these dozen live ranges restrict the graph so much that some registers must be spilled. Looking at the computed spill costs, it appears cheaper to spill the loop indices and limits for the array copy loop (variables I, J, M and N) than the longer live ranges. Unfortunately, spilling them does not lower the register pressure in the large loop nests, so more live ranges must be spilled. After this has been done, most of the longer live ranges have been spilled and coloring proceeds. The final result: the code has almost no register utilization in the array copy loops.

1.3. A New Implementation

As a result of the poor code discovered in SVD, we again focused our attention on the problem of register allocation. While we could imagine methods that might produce better code, we were concerned about the allocator's efficiency. These concerns prompted a new implementation of the register allocator (actually, of the entire code generator). With a new, cleaner implementation, we hoped to gain a better understanding of the performance of a graph coloring register allocator.

In our new implementation, we were much more careful, focusing both on the choice of efficient data structures and algorithms and on the low-level details of coding. In particular, we followed the description of IBM's version much more closely [CACC 81]. The results were gratifying. Our new allocator was slightly smaller (though more complex) and more than twice as fast on small test cases. On larger test cases, the improvement was even more dramatic. In addition, the new version produced better spill code because of corrections and improvements made during the rewrite.

So, we had a shiny new register allocator, producing better code and running faster. Nevertheless, SVD still had the same problem. This was not unexpected; SVD exposed a problem with the coloring heuristic, not the implementation. However, we had a good basis for experimentation.

2. Improving the Coloring

For a time, we considered exploring techniques for live range splitting [Chow 83, ChHe 84, LaHi 86]. We were saved from yet another implementation effort by the discovery of a simple improvement to Chaitin's heuristic [Chai 82].

2.1. Chaitin's Heuristic

The fundamental idea behind Chaitin's register allocator is to use the routine's interference graph as a guide to allocating registers. Nodes in the interference graph represent live ranges. Each edge in the graph represents an interference between two live ranges. Thus, if two live ranges exist at a single point in the program, there is an edge between their nodes.[2] If a node has n neighbors, we say the node is of *degree n*. Chaitin's allocator attempts to color the graph in k colors, where k is the number of registers available. If it cannot find a k-coloring, it spills one or more live ranges in order to transform the graph into one that has a k-coloring. This process is repeated until it has a k-colorable graph.

There are many different ways to find a coloring of a graph. Chaitin's heuristic has two particularly nice properties. First, it is fast, taking time linear in the size of the graph. Second, in the event that it cannot color the graph, it provides useful guidance about which live ranges in the routine should be spilled. Since our method derives from Chaitin's, it is appropriate to begin with a high-level overview of how his method works.

There are three phases to Chaitin's method. The first phase builds an interference graph for the routine. The second phase simplifies the graph and produces a stack of nodes to drive the third phase. The third phase assigns colors to the nodes of the simplified graph. Because the differences between Chaitin's technique and ours lie in the simplification and coloring phases, they deserve more explanation.

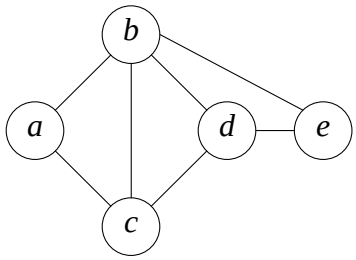

Figure 2 — a graph requiring three colors

The simplification phase repeats the following steps until the graph is empty.

1) If there exists a node n with $degree(n) < k$, remove n and all of its edges from the graph. Place n on the stack for coloring.
2) Otherwise, choose a node n (live range) to spill. Remove n and all of its edges from the graph. Mark n to be spilled.

If phase two marks any node for spilling, instructions are inserted into the intermediate text to spill the live range; that is, the value is stored to memory after each definition and restored before each use.[3] Because inserting spill code involves some register use, it may induce additional spilling. The register allocator repeats the entire process of building the interference graph, simplifying it, and inserting spill code until no additional live ranges must be spilled.

In (2), we must decide which of the remaining nodes to spill. Chaitin recommends taking the node with the lowest ratio of *spill cost* to *degree*. For a live range, the spill cost is the number additional cycles that would be required to save and restore the live range. We estimate the spill cost as the number of loads and stores that would have to be inserted, weighted by the loop nesting depth of each insertion point. These costs are precomputed. When the simplification phase reaches a point where it must choose a node to spill, it divides the precomputed spill cost by the node's current degree.

Once all the spill code has been inserted, the actual coloring is handled by the third phase. It is driven by the stack produced in phase two. While the stack is non-empty, it performs the following steps:

[2] The precise definition of interference is subtle. See the original paper for the "ultimate" notion of interference [CACC 81].

[3] See Chaitin's paper for refinements of this simple scheme [Chai 82].

1) Remove n from the stack and reinsert it in the graph, along with all its edges.
2) Assign n a color that differs from all its neighbors.

This coloring process must succeed; the work done in simplification ensures that fact.

Suppose we are trying to find a 3-coloring for the graph shown in Figure 2 (that is, we have a machine with three registers). Applying the simplification phase will remove all the nodes, marking none for spilling. One possible sequence of removals is *a, c, b, d, e*.

Next, the coloring phase reinserts nodes into the graph, assigning colors. In our example, the node *e* is inserted first and can be given any color, say *red*. Next, the node *d* is added; *d* can get any color except *red*. Running through the entire stack might result in the assignment: *e:red*, *d:blue*, *b:green*, *c:red*, and *a:blue*.

Chaitin's heuristic is not guaranteed to find the minimal coloring. For example, suppose we want to find a 2-coloring of the simple graph shown in Figure 3. Clearly, one exists; for example *w:red*, *x:blue*, *y:blue*, and *z:red*. However, the simplification phase immediately runs into a problem; it finds that there are no nodes with degree less than two. Thus, it must select a node to spill. Assume for the example that all spill costs are equal, so we can choose any of the nodes to spill; for example, *x*. We remove *x* from the graph and mark it for spilling. Phase two will now remove the remaining three nodes without further spilling. Since we have a node marked to spill, we have to insert spill code, rebuild the interference graph, and try again.

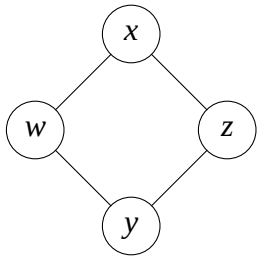

Figure 3 — a simple graph requiring two colors

This example shows that Chaitin's heuristic does not always find the minimal coloring, nor can it be guaranteed to find a *k*-coloring, if one exists. Despite this, Chaitin's method produces excellent results. It does this by generating good allocations and using spill costs to guide the generation of spill code.

2.2. An Improvement

Studying the simple example in Figure 3 suggests an improvement to Chaitin's approach that is based on an algorithm due to Matula and Beck [MaBe 81]. Instead of removing any node that has a degree less than or equal to the number of colors, it is possible to remove, at each step, the node with the lowest degree. This can be accomplished using the following method.

1) Let N be an array, such that $N[i]$ is the first element of a linked list of nodes that have i neighbors. If there are no such nodes, $N[i]$ is null.
2) Let the interference graph be represented by nodes, each of which is on some list in N, and edges, represented by an adjacency list.
3) At each stage, search N from the beginning until the first non-null cell $N[i]$ is found. Remove from the graph the element at the head of the list beginning at $N[i]$, and move each of its neighbors down one cell in N, reflecting the fact that each of them has one fewer neighbor in the graph.

The critical observation, due to Matula and Beck, is that each node is originally in a box indexed by the number of edges incident upon it. Thus, the search in (3) that removes a node will take no more steps than the degree of the node removed. The total cost of searching is, therefore, bounded by the sum of the degrees of every node, which is exactly equal to twice the number of edges in the graph. Since every other step in (3) can be done in constant time, the total cost is linear in the size of the interference graph. This is the same as the

asymptotic bound achieved by Chaitin's algorithm.

In fact, the search in (3) can be shortened by the following observation. If we have just removed node n from $N[i]$, $i>0$, the search can start at $N[i-1]$ rather than at $N[0]$. Because we removed a node with degree i, we know that there were no nodes of lower degree available. In removing n, we lowered the degree of each of its neighbors by exactly one. This might have created a node of degree $i-1$, but it cannot have created a node of lower degree. Thus, $N[j]$ must still be empty, for $0 \le j < i-1$.

Once all the nodes have been removed, we begin the coloring phase, in which we rebuild the graph by inserting each node and its associated edges in the reverse of the order of deletion. As each node is inserted, it is colored with the first color that does not appear in any of its neighbors. If this procedure is applied to Figure 3, we are able to color the graph with two colors, even though Chaitin's procedure failed.

Why does this work? When we insert a node that has k or more neighbors, we may discover that some of its neighbors have been assigned the same color. In that case, we still may have an available color to assign it. Our method will color such a node and go on. In Chaitin's method, the node would have already been spilled.

However, one problem remains: what to do when the coloring phase encounters a node that already has neighbors with each of the k colors. At that point, we must choose a live range to spill. After all, not every graph can be colored in k colors, no matter how powerful our coloring heuristic.

Eventually, we arrived at the following approach. Consider the third phase of the process, where nodes are added back into the graph and colors are selected. If, at some point, a node is added and no color is available, we leave it uncolored. Then, we continue reinserting the rest of the nodes and giving them colors where possible. After the entire graph has been rebuilt, we insert spill code for the uncolored nodes, rebuild the interference graph, and try again. Deferring the spill decision opens up a second opportunity for improvement. When we insert a node that has k or more neighbors, some of them may be uncolored (spilled), possibly leaving a color available.

This is exactly what happened in SVD. Chaitin's simplification phase chose the indices and limits of the array copy loop as the first candidates for spilling. Unfortunately, spilling them doesn't help the situation in the later loops; some of the long live ranges must be spilled. By moving the spill decision to the third phase, we allow the allocator to reconsider each spill decision, in inverse order. Thus, it can avoid pointless spills, like those we saw in SVD.

The overall flow of our register allocator is similar to Chaitin's (see Figure 4). The major difference is that the choice of live ranges to be spilled is deferred until the coloring phase.

As a graph coloring technique, this method is stronger than Chaitin's method. If Chaitin's method colors the graph without inserting spill code, our method will, too. If Chaitin's method introduces spill code, our method may need to introduce spill code.

2.3. A Final Refinement

The problem with the method described in section 2.2 is that it does not take any notion of spill cost into account. Since it only looks at the degree of a node, it would be forced to make spill decisions without any estimate of their run-time impact. Such an allocator would produce arbitrary allocations — possibly terrible allocations.

One of the strength's of Chaitin's method is that the simplification phase encodes information about the spill costs associated into its ordering. This ordering information, in turn, forces the coloring phase of the algorithm to assign more expensive nodes first. We wanted to capture this property of Chaitin's work and incorporate it into our allocator. To accomplish this, we modified the simplification phase so that it carefully orders the nodes based on cost in those areas where the coloring phase may need to generate spill code.

To add this ordering information, we went back to the notion underlying Chaitin's ordering. We remove the nodes with fewer than k neighbors in arbitrary order. Whenever we discover that all the remaining nodes have k or more neighbors, we fall back on Chaitin's estimator and find the node with the minimum ratio of *spill cost* to *degree*. We remove that node from the graph and push it on the stack. This has the effect of ordering the nodes by Chaitin's metric in the vicinity of any node that his heuristic would have marked for

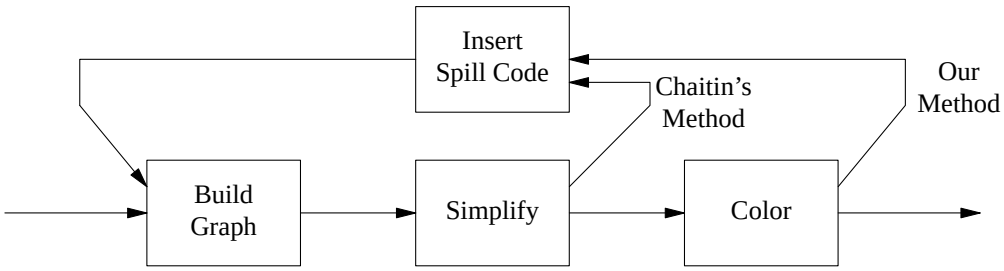

Figure 4 — register allocator structure

spilling.

This ordering information helps us in two ways. First, it determines the order in which the coloring phase assigns colors in the neighborhood of a possible spill. Second, if we are forced to spill, it ensures that we spill the same live range that Chaitin's method would spill.[4] Thus, either we spill a subset of the live ranges that Chaitin would spill or the same set.

This second observation is important. Our method always does as well as Chaitin's method. In some cases, it does better.

3. Results

We applied our new coloring heuristic to the SVD routine. The improvement was stunning. The number of registers spilled was reduced by 51%; the estimated spill costs were reduced by 22%. What happened? As nodes were removed from the graph, the nodes for those live ranges in the smaller loop nests were removed relatively early. However, this time all of the nodes were placed on the stack. The simplification phase continued removing nodes from the graph until it was empty. Then, when inserting nodes back into the graph, the coloring phase was forced to spill some of the longer live ranges. When it reached the live ranges in the smaller loop nests, it discovered that it still had available registers, so it was able to assign them to registers. In essence, our new heuristic allowed the coloring phase of the allocator to clean up bad decisions made in the simplification phase.

Overall, registers were used much more effectively because the long live ranges were spilled. This lowered the register pressure throughout the SVD routine. The short live ranges, like those for the loop indices and limits in the array copy loop (see Figure 1), could now be allocated registers. This also explains the difference in improvement between the number of registers spilled and the estimated cost of spilling them. The values that get assigned registers under the new heuristic tend to have shorter live ranges. The improvement reflects the fact that the algorithm is cleaning up small cases that can really be assigned to registers, but get missed in Chaitin's method.

3.1. Numerical Programs

In general, the results from our new register allocator have been encouraging. Figure 5 summarizes the results we obtained from applying it to five floating-point intensive programs.[5] Some explanation of these numbers is in order.

The table is organized by programs. Within

[4] To the extent that either algorithm is consistent. If they encounter two live ranges with the same spill cost, the choice between them is made on some basis — often something as trivial as a symbol table index.

[5] LINPACK is Dongarra's famous double precision benchmark [DBMS 82, Dong 83]. SVD is a version of the singular value decomposition from Forsythe, Malcom, and Moler [FoMM 77]. SIMPLEX is a parallel optimization code that executes a multi-directional search along simplex edges [Torc 89]. EULER is a 1D simulation of shock wave propagation. The routines under CEDETA are part of the Celis-Dennis-Tapia code for equality constrained minimization [CeDT 84]. Only a few of the routines from CEDETA are shown.

Program	Routine	Object Size	Live Ranges	Registers Spilled			Spill Cost			Dynamic
				Old	New	Pct.	Old	New	Pct.	Pct.
SVD	SVD	4,456	442	101	49	51	596,713	465,627	22	1.35
LINPACK	EPSLON	136	37	0	0	0	0	0	0	1.05
	DSCAL	184	42	0	0	0	0	0	0	
	IDAMAX	224	46	0	0	0	0	0	0	
	DDOT	272	50	2	2	0	22	22	0	
	DAXPY	296	51	3	3	0	33	33	0	
	MATGEN	360	57	3	3	0	34	34	0	
	DGEFA	592	71	18	16	11	3,611	3,331	8	
	DGESL	704	94	21	16	24	635	499	21	
	DMXPY	2,000	230	57	55	4	22,151	22,020	1	
SIMPLEX	VALUE	280	55	2	2	0	22	22	0	1.37
	CONVERGE	304	58	1	1	0	11	11	0	
	CONSTRUCT	304	65	1	1	0	11	11	0	
	SIMPLEX	2,792	157	24	13	46	25,860	16,827	35	
EULER	SHOCK	184	43	0	0	0	0	0	0	0.19
	DERIV	592	80	2	2	0	22	22	0	
	CODE	928	142	20	19	5	775	775	0	
	CHEB	976	137	7	7	0	310	310	0	
	FINDIF	1,064	131	18	13	28	10,878	10,658	2	
	FFTB	1,064	133	17	17	0	14,036	14,036	0	
	BNDRY	1,424	167	3	3	0	7	7	0	
	INPUT	2,056	178	12	6	50	432	342	21	
	DIFFR	2,528	237	38	28	26	7,141	6,546	8	
	DISSIP	2,936	366	52	16	69	33,664	32,766	3	
	INIT	4,328	479	41	38	7	3,792	3,780	0	
CEDETA	DQRDC	1,840	152	45	24	47	14,479	12,223	16	n/a
	GRADNT	14,672	1,274	73	59	19	1,893	1,413	25	
	HSSIAN	16,376	1,552	135	91	33	2,870	2,503	13	

Figure 5 — register allocation improvements

a program, individual subroutines are listed.[6] The object size column gives the number of bytes of object code generated using our technique. The numbers for spilling are reported by the allocator. Spill costs are calculated as in Chaitin's work (described in section 2.1). For each of the static measurements, the table shows numbers for Chaitin's method (Old), our method (New), and the percentage difference (Pct). The final column gives the measured run-time performance improvement for each program.

In general, it appears that large routines show a greater percentage improvement in the number of registers spilled than do small routines. This is not surprising. Large routines tend to have large and complex interference graphs. They also

[6] As of March, 1989, the compiler does not handle I/O constructs. Routines containing I/O were compiled with the vendor supplied *f77* compiler. Thus, for example, the driver routines for each program are not listed.

tend to have both more live ranges and more long live ranges. This produces more nodes with degree greater than *k*, provoking more spilling. The difference in the number of registers spilled by the two methods highlights the frequency with which a node of high degree has either neighbors that were assigned the same color, or neighbors that were spilled.

In more than half of these routines, we show no static improvement. These routines tend to be small. In these cases, both methods produce the same code. As we noted in section 2.3, when our method cannot improve on Chaitin's, it produces the same results.

The amount of dynamic improvement is small. Recall that the data listed for registers spilled and for spill costs are relative — the percentages show the reductions in the amount of spilling. The static numbers say nothing about how total spill-related cycles relate to total execution cycles. For the programs in Figure 5, floating point instructions dominate the execution time, as shown by the dynamic improvement numbers. Thus, spill-related cycles are a small part of total execution time. This is consistent with the results reported by Larus and Hilfinger [LaHi 86]. For example, in LINPACK, around 90% of the execution time is spent inside DAXPY, a routine that spills only three live ranges, all integer values.

Finally, there are two large routines that defy all the trends, INIT and DMXPY. The explanation for INIT is simple. This routine initializes all the data for EULER. It consists of a long series of assignment statements and simply nested loops. It generates a relatively simple interference graph with low spill costs.

For DMXPY, the explanation is more complex. DMXPY is a routine that multiples a matrix by a vector and adds the result to another vector — a fairly simple computation that can be implemented with a loop that looks like:

```
DO J = 1,N2
  DO I = 1,N1
    Y(I) = Y(I) + X(J) * M(I,J)
  ENDDO
ENDDO
```

It appears that this routine was actually implemented by unrolling the J loop sixteen times and combining the assignments to Y. Thus, the routine consists of the computation in the unrolled J loop along with several auxiliary I loops to handle cases where the number of columns in the matrix is not evenly divisible by sixteen. The unrolled J loop looks like:

```
DO J = JMIN, N2, 16
  DO I = 1, N1
    Y(I) = ((((((((((((((( Y(I))
       + X(J-15)*M(I,J-15)) + X(J-14)*M(I,J-14))
       + X(J-13)*M(I,J-13)) + X(J-12)*M(I,J-12))
       + X(J-11)*M(I,J-11)) + X(J-10)*M(I,J-10))
       + X(J- 9)*M(I,J- 9)) + X(J- 8)*M(I,J- 8))
       + X(J- 7)*M(I,J- 7)) + X(J- 6)*M(I,J- 6))
       + X(J- 5)*M(I,J- 5)) + X(J- 4)*M(I,J- 4))
       + X(J- 3)*M(I,J- 3)) + X(J- 2)*M(I,J- 2))
       + X(J- 1)*M(I,J- 1)) + X(J)   *M(I,J)
  ENDDO
ENDDO
```

Notice the complexity of the assignment statement. It is not surprising that our coloring algorithm failed to significantly improve the amount of spilling in this routine. DMXPY is simply an example of how the decision to perform one reasonable optimization can significantly reduce the effectiveness of later optimizations.

3.2. An Integer Program

The emphasis of the project is to provide support for scientific programs. Thus, most of our experimentation has involved programs where floating point operations are the dominant cost. To get a better feeling for the impact of spill code, we looked at a program with no floating point arithmetic, a non-recursive quicksort routine.[7] Using sixteen registers, we found no difference in the spill code generated by Chaitin's method and ours. In both cases, the code produced was quite good (sorting an array of 200,000 integers in eight seconds). To look at the effect of smaller register sets, we modified both register allocators to use a subset of the machine's sixteen general purpose registers. Figure 6 summarizes the results of compiling quicksort with each allocation method and a variety of available registers.[8]

While a single routine comprises a very small test set, the experimental results suggest

[7] Quicksort is an implementation of the non-recursive algorithm given by Wirth [Wirt 76]. It was implemented by R.M. Lewis at Rice.

[8] The RT/PC's register usage conventions prevent meaningful experimentation with less than 8 registers.

Number	Registers Spilled			Spill Cost			Object Size			Running Time		
Registers	Old	New	Pct.	Old	New	Pct.	Old	New	Pct.	Old	New	Pct.
16	3	3	0	1,303	1,303	0	360	360	0	8.2	8.2	0
14	5	4	20	5,105	2,568	50	384	368	4	8.3	8.2	1
12	8	6	25	11,809	7,750	34	400	392	2	8.7	8.4	3
10	13	8	38	37,000	15,875	57	440	416	5	10.0	8.9	11
8	17	11	35	125,000	71,675	43	464	432	7	13.2	11.2	15

Figure 6 — quicksort study results

several conclusions.

- The quality of spill code is important, both dynamically and statically.
- Our method shows greater improvement over Chaitin's method in highly constrained situations.
- An adequate register set is important; with "only" eight registers, quicksort ran 27% slower and required 17% more code space.

We intend to collect more data on the effectiveness of our allocator for smaller register sets. Additionally, we would like to experiment with a more diverse set of non-floating point programs.

3.3. Costs

In earlier sections, we argued that our coloring procedure runs as quickly as Chaitin's. The whole truth is somewhat more complex. It is important to distinguish between the time spent in each of the phases shown in Figure 4, and the overall time spent performing allocation.

The time spent in a single execution of the graph building phase will be the same in either method. We assume that the amount of time spent generating spill code depends on the number of live ranges spilled. It will vary somewhat, depending on the number of individual loads and stores that must be inserted for a live range. That minor difference is factored out by the fact that our method spills either the same set or a subset of the live ranges spilled by Chaitin.

Consider the remaining parts of the allocator, simplification and coloring. In the event that no spill code is needed, Chaitin's method, Matula and Beck's method, and our method will all run in time linear in the number of live ranges plus interferences. If the simplification phase reaches a point where each remaining node has k or more neighbors, this situation changes. Here, Matula and Beck will retain their linear behavior — their method simply removes the nodes of higher degree, in ascending order. Both Chaitin's method and our method, however, must examine spill costs at this point. In selecting the next node to remove, they must find the node with minimum estimated cost. A simple implementation would search all the remaining nodes; a more clever algorithm might use a heap to decrease the cost of finding subsequent values. In either case, the costs involved in making that selection are the same in both Chaitin's method and ours.

Given an understanding of the costs of the individual parts of the allocator, we can now discuss the total allocation time. For any graph that Chaitin's procedure can color without inserting spill code, our procedure has the same behavior. Both will run in time linear in the size of the interference graph. In cases where Chaitin's procedure needs to insert spill code, two differences arise:

1) Both methods will spend the same amount of time in the simplification phase, but our method will run through the coloring phase, where Chaitin's will not. Fortunately, the time for the coloring phase is linear in the number of live ranges, so the asymptotic complexity of both methods remains the same.

2) In those cases where we spill fewer live ranges, we will spend slightly less time generating that spill code. The experimental evidence suggests that this happens about half of the time; the relevant number in the table is the number of registers spilled. We never spill more live ranges than Chaitin.

Thus, if no spill code is necessary, both allocators run in time linear in the size of the interference graph. If spill code must be generated, the results will vary from case to case, but the asymptotic complexity of the two methods will be the same.

The dominant factor in allocation time is not the execution speed of the individual phases, but rather the number of times that the Build-Simplify-Color cycle must be repeated. When a coloring cannot be found, spill code must be inserted and the entire cycle must be re-executed (recall Figure 4). In some cases, the allocator makes several passes through the cycle. An intuitive explanation is that spilling a live range does not entirely remove it; it simply divides that live range into several shorter live ranges, one for each definition or use. These shorter live ranges must also be colored; this may provoke further spilling. Fortunately, the process seems to converge very rapidly; a typical large routine might spill fifty live ranges during the first pass, but only two live ranges during the second pass (see Figure 7).

For any given routine, the two algorithms may require a different number of passes. For example, on DMXPY from Figure 5, our method requires three passes where Chaitin's requires only two. On the other hand, there are examples (like Figure 3) where our method avoids spilling completely, but Chaitin's technique must insert spill code; that forces his method to take at least one additional pass. In our experience, the two algorithms typically require the same number of passes. We have never observed either method needing more than three passes.

We have collected some measurements of the CPU-time required by various phases of the register allocator; these result are presented in Figure 7. The times shown are in CPU-seconds, measured with a 60 hertz clock. The parenthesized numbers in the *spill* row indicate the number of registers being spilled. The *build* row entries are unfortunately somewhat distorted for the larger routines. The build phase actually has three major parts: finding and renumbering distinct live ranges, repeatedly building the graph and coalescing registers, and calculating spill costs. In the current implementation, the build phase is dominated by the time required to calculate spill costs — a situation that we hope to remedy.

It is immediately apparent how inexpensive the simplification and coloring phases are; in the largest example, they take less than three seconds total. The extra passes through the coloring phase required by our method increase the total allocation time only slightly. Note that the spill times and later build times slightly favor our method — a consequence of processing fewer spill instructions. Overall, the time required for the two methods appears to be quite similar.

Also of interest is the behavior of the simplification phase as allocation progresses. The speedup between first and second invocations is due to the amount of time spent handling nodes of high degree. Recall that whenever the simplifier has only nodes of degree $\geq k$, it must search through the remaining nodes, comparing the ratios of spill cost to degree. During the second (and third) invocations, the number of such searches is much smaller.

4. Conclusions

For appropriate target machines, a register allocator based on graph coloring is a good choice. The compile-time performance is good, and the resulting code is excellent. Further, the luxury of good global register allocation simplifies the design of the rest of the compiler.

Since finding a minimal coloring is NP-Complete, work in this area has turned to heuristic methods. These techniques sacrifice precision for speed. The difficulty is in finding a heuristic that:

Phase	DQRDC		SVD		GRADNT		HSSIAN	
	Old	New	Old	New	Old	New	Old	New
Build	1.75	1.75	6.40	6.40	16.40	16.40	21.50	21.50
Simplify	.10	.10	.74	.74	.72	.72	1.83	1.83
Color		.03		.10		.18		.23
Spill	(45) .14	(24) .13	(101) .67	(49) .62	(73) 4.02	(59) 4.00	(135) 6.33	(91) 6.29
Build	1.70	1.51	8.49	6.85	20.35	20.10	27.58	27.36
Simplify	.02	.02	.06	.04	.13	.12	.15	.17
Color	.03	.03	.06	.06		.17		.18
Spill					(4) 4.15	(4) 4.12	(8) 6.87	(6) 6.83
Build					17.78	17.75	24.30	24.38
Simplify					.12	.12	.13	.13
Color					.18	.17	.18	.18
Total	3.74	3.57	16.42	14.81	63.85	63.85	88.87	89.08

Figure 7 - CPU time for allocator phases

- is computationally inexpensive, both asymptotically and practically,
- performs well on the graphs arising from real programs, and
- provides useful direction when spill code must be introduced.

Both Chaitin's method and our own meet these criteria.

Our results, to date, are encouraging. Our heuristic either produces the same allocation as Chaitin's heuristic, or it produces a better allocation. The costs, both in implementation effort and compile time, are comparable.

We intend to continue studying the performance of our allocator, including analyses of larger programs. The CPU-time measurements in Figure 7 suggest that we should investigate improved techniques for calculating spill costs. We may also explore live range splitting as a means for improving the overall allocation.

5. Acknowledgements

Jorge Moré pointed out Matula's algorithm, which influenced this work. Mark Krentel suggested that our coloring heuristic was giving up too easily and encouraged us to try a more aggressive coloring scheme. John Cocke, Peter Markstein, and Vicky Markstein provided us with invaluable encouragement and advice.

6. References

[AuHo 82] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices* 17(6), June, 1982.

[CCHK 87] A. Carle, K.D. Cooper, R.T. Hood, K. Kennedy, L. Torczon, and S.K. Warren. A practical environment for scientific programming. *IEEE Computer* 20(11), November, 1987.

[CeDT 84] M.R. Celis, J.E. Dennis, and R.A. Tapia. A trust region strategy for equality constrained optimization. In *Numerical Optimization 1984* (P.T. Boggs, R.H. Byrd, and R.B. Schnabel, editors), SIAM, 1984.

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

[CACC 81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages* 6, January, 1981.

[Chow 83] F. Chow. A portable machine-independent global optimizer—design and measurements. PhD. Thesis. Technical Note No. 83-254, Computer Systems Laboratory, Stanford University, Dec. 1983.

[ChHe 84] F. Chow and J. Hennessy. Register allocation by priority based coloring. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6), June, 1984.

[CoKT 86] K.D. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the programming environment. *ACM Transactions on Programming Languages and Systems* 8(4), October, 1986.

[DBMS 82] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users' Guide*, SIAM, Philadelphia, PA., 1982.

[Dong 83] J.J. Dongarra. Performance of various computers using standard linear equations software in a Fortran environment. Argonne National Laboratory Technical Memorandum 23, August 1983 and subsequent revisions.

[FoMM 77] G.E. Forsythe, M.A. Malcolm, and C.B. Moler. *Computer Methods for Mathematical Computations*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[LaHi 86] J.R. Larus and P.N. Hilfinger. Register allocation in the SPUR Lisp compiler. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices* 21(7), June, 1986.

[MaBe 81] D. Matula and L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. TR CSE 8104, Dept. of Computer Science and Engineering, Southern Methodist University, Dallas, TX, July, 1981.

[Torc 89] V.J. Torczon. Nonlinear optimization by parallel searches on simplex edges. PhD. Thesis, Department of Mathematical Sciences, Rice University, Houston, TX, expected May, 1989.

[Wirt 76] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.