SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1(1), 1 (JANUARY 1988)

How to Build an Interference Graph *

Keith D. Cooper, Timothy J. Harvey, and Linda Torczon
Rice University, 6100 S. Main – MS 132, Houston, Texas 77005

SUMMARY

The design and implementation of an interference graph is critical to the performance of a graph-coloring register allocator. The cost of constructing and manipulating the interference graph dominates the overall cost of allocation. The literature on graph-coloring register allocation suggests the use of a bit matrix coupled with lists of edges to represent the graph.[1-3] Recently, George and Appel claimed that their tests show better results using a hash table.[4] This paper examines the tradeoffs between these two approaches.

Our experiments were conducted with an optimistic, Chaitin-style register allocator.[5] We believe, however, that the lessons learned in the experiment are applicable to any program that needs to build and manipulate large graphs. For most graphs, we obtained our best results, in terms of both time and space, using a modification of the data structures suggested by both Chaitin and Briggs that we call the split bit-matrix method. On a few large graphs, we found that a closed hash-table with the universal hash function suggested by Cormen *et al.* [6] ran faster than the split bit-matrix method. We found one case where it used less space.

This suggests that the split bit-matrix technique should be the method of choice, unless the compiler regularly encounters large interference graphs. In that case, the best strategy might be to implement both data structures behind a common interface, and switch between them based on graph size.

KEY WORDS Compilers Register allocation Interference graph Hashing

Introduction

When designing a data structure for a complex application, one or both of the following constraints inevitably arises: the machine lacks sufficient memory, or the solution requires more computation than the user feels is justifiable. Often, the tradeoff becomes a simple matter of time versus space. By spending more space, the programmer can improve running time, while decreasing space increases running time. The programmer must balance these two factors, finding a scheme that makes judicious use of both time and space.

In few places is this challenge more pronounced than in a compiler. Each phase of the compiler, from parsing, through optimizing, to generating code, requires vigilance to ensure that no algorithm slips into an undesirable running time. Complicating the

* This material is based in part upon work supported by the Texas Advanced Technology Program under grant number 003604-015 and by DARPA through Army Contract DABT63-95-C-0115.
Address correspondence to Tim Harvey, harv@cs.rice.edu

CCC 0038–0644/88/010001–01

Received July 18, 1997
Revised September 5, 1997

Routine Name	*Nodes*	*Edges*	*Build Time (in seconds)*	*Total Time (in seconds)*	*Build %'age*
`fieldX.i`	5936	268353	65.54	102.75	64
`smoothX.i`	1776	74771	57.49	85.23	67
`parmvrX.i`	5614	204491	50.46	82.99	61
`parmovX.i`	5220	195009	45.27	73.42	62
`radbgX.i`	2651	156978	34.42	50.12	69
`twldrv.i`	3135	309340	31.47	46.07	68
`radf5X.i`	1967	84785	26.18	42.34	62
`radfgX.i`	2788	148129	26.21	38.34	68
`radb4X.i`	1562	65070	23.19	34.56	67
`fpppp.i`	3899	723605	19.75	32.87	60
`parmveX.i`	2697	101962	18.33	28.41	65
`radb5X.i`	1968	84833	13.51	20.60	66
AGGREGATE	91870	3847704	556.28	852.45	65

Table I. Interference-Graph Build Times

task is the complexity of a compiler's input—the compiler cannot afford to assume that the worst case never occurs.

The back end of a compiler includes several phases that approximate the solutions of NP-complete problems. This paper examines some data-structure design tradeoffs in one such phase—register allocation. The register allocator must decide, at each point in the object code, which values to keep in registers and which values to keep in memory. A well-known and widely-used technique for solving this problem works by analogy to graph coloring.[7, 1] The compiler builds a graph, called the interference graph, that encodes the fact that two values cannot occupy the same register or memory location. Nodes in this graph represent values. An edge between two nodes exists if and only if they are simultaneously live at some point in the code. Thus, an edge between n_1 and n_2 implies that they cannot occupy the same register. A coloring allocator solves the problem by discovering a k-coloring for the graph—that is, a mapping of k colors onto the nodes of the graph in such a way that no adjacent nodes have the same color. If a k-coloring cannot be found, the allocator modifies the problem by moving some values from registers into memory and tries again. When it discovers a k-coloring, it then maps the k colors onto the hardware registers of the target machine.

The costs of building and accessing the interference graph largely determine the total cost of graph-coloring register allocation. These costs are determined by two distinct factors:

1. *the representation used to implement the graph* — The algorithms and data structures that the compiler writer chooses to implement directly affect the cost of building and accessing the interference graph. This paper examines several possible implementation strategies.
2. *the properties of the individual graph being colored* — The size, connectedness, and sparsity of the interference graphs seen in a coloring allocator vary widely, in ways that might affect the choice of graph implementation. The compiler writer has little or no control over these properties.

Interference graphs come in many shapes and sizes. The graph can be a complete graph, that is, a graph that has an edge between each possible pair of nodes. A complete graph has $N(N-1)/2$ edges; this is a clear upper bound. The value of N varies significantly, as does the ratio of E/N. In our test suite of relatively small programs, N ranges from

2 to over 5,936; and E ranges from 1 to 723,605. We have seen computer-generated procedures that create graphs with an order of magnitude larger graphs.

Table I shows the interplay between the size of the interference graph, time to build the graph, and total allocation time. The graph was built as suggested by both Chaitin and Briggs, using a lower-triangular bit matrix and an edge list for each node to represent the graph. With each node, we store a tag that indicates the type of register to which it can be assigned. For most modern microprocessors, the tag can represent either an integer register or a floating-point register. The tags allow the allocator to omit edges between values of different type, reducing the total number of edges. In each case shown in Table I, building the graph required at least 60% of the total allocation time. Across the entire suite of 169 routines from parts of the Spec benchmark suite and Forsythe, Malcolm, and Moler's small library of numerical methods,[8] the average time spent building the graph was 65% of total allocation time.

The choice of data structure directly affects the running time of the rest of the allocator as well. The other phases all rely on information encoded in the graph, so access costs are a major factor in the performance of the other phases. Both Chaitin and Briggs stress that an efficient allocator requires both the bit-matrix for a quick membership test and the edge lists to speed iteration over a node's edges.[1-3] We know of several cases where implementors omitted the bit matrix in an attempt to save both time and space; unfortunately, this produces a dramatic slowdown in the rest of the allocator.

This paper examines several alternatives for building and representing the interference graph. Chaitin *et al.* first suggest the use of a hash-table for large graphs.[1] Our investigation was prompted by George and Appel's suggestion that hashing was an improvement over a bit-matrix.[4] To evaluate the issue, we built several variations of both the bit-matrix and the hash-table schemes into the register allocator for our research compiler.* Our primary motivation in this work was to examine the impact of data-structure choice on both the time and the space required for allocation. In our experiments, one variation of the bit-matrix scheme ran faster and used less space for almost all of the procedures that we compiled. On a few large procedures, one of the hash table schemes ran faster; it used less space on one procedure.

Bit Matrix

Chaitin suggests representing the interference graph using both an edge list for each node and a triangular bit matrix.[1, 2] The edge list allows for efficient examination of a node's neighbors, while the bit matrix ensures a constant-time membership test. Since the allocator uses both kinds of operation heavily, the costs of each are critical to the allocator's performance. The disadvantage of the triangular bit matrix lies in the amount of space that it requires. It always requires $N(N-1)/2$ bits, regardless of the number of edges. The hash table technique replaces the bit matrix with a hash table, on the theory that it requires space roughly proportional to E. If $E \ll N^2$, the hash table should require less space. The hash table has the same expected case asymptotic behavior as the bit matrix. However, the overhead per hash operation may be higher

* Another implementation strategy would use Cai and Paige's multiset discrimination technique to replace the hash table.[9] We did not experiment with multiset discrimination, primarily because the hash keys consist of two relatively small integers. With these keys, we felt that the discrimination technique would devolve into an unbalanced binary search.

than that required for a triangular matrix lookup, and the performance of the lookup can degrade due to bad behavior from the hash function.

Our bit-matrix implementation uses the `VectorSet` implementation described by Briggs and Torczon.[10] Because the address computation for a diagonal array generates a multiplication and a division, $(low + (high \times (high - 1))/2)$, we precompute a lookaside array that stores all of the possible values of $(high \times (high - 1))/2$. This ensures that each access requires only an array lookup and an addition.

Improvements to Chaitin & Briggs

Our current implementation, which, for clarity of exposition, we will refer to as the split bit-matrix method, departs from Chaitin and Briggs in two ways. First, it incorporates an insight that underlies both Gupta, Soffa, and Steele's work on clique separators for graph-coloring allocation[11] and Koblenz and Callahan's hierarchical coloring work.[12] Both papers reduce the size of the interference graph by breaking it into disjoint subgraphs. If we can determine, at design time, that two classes of nodes cannot possibly interfere, we can construct separate and smaller graphs for each class. Since the bit matrix requires $\mathbf{O}(N^2)$ space, the savings can be substantial.

Modern microprocessors feature multiple functional units and often split their registers into two or more classes. Typical processors have both integer, or general purpose, registers and floating-point registers. This creates a natural split for register allocation. Since floating-point values and integer values cannot occupy the same hardware registers, they cannot interfere. Thus, we can build separate interference graphs for the integer values and the floating-point values. Initially, this idea seemed flawed: surely, we reasoned, the ratio of integer values to floating-point values is so one-sided that there would be little benefit from this method. (For example, all address computations require integer registers.) As Table II shows,* however, there is enough balance

between the two types of values in our examples that splitting the interference graph into two disjoint graphs provides a substantial memory savings.†

Our second departure from prior practice relates to the construction and storage of edge lists for each node. Both Chaitin and Briggs recommend using two passes to build the edge lists. The first pass determines the degree of each node; the second pass builds a precisely-sized vector to hold each edge list. To simplify the creation of the edge vectors, entries in the bit matrix are used as flags to record the fact that each edge has been processed. Thus, both the first pass and the second pass fill in the bit matrix. This necessitates both clearing and setting the bit matrix twice.

The extra pass over the graph seems wasteful. While it does not change worst case asymptotic complexity, it does increase the running time of the allocator. One alternative is to expend extra space in the vectors that hold the edge lists. Instead of sizing each array precisely, the allocator can choose a standard size for an edge list vector and chain together multiple vectors if more slots are needed. Unless all nodes have a degree equal to an integer multiple of the number of pointers in the standard edge-list node, some pointer slots will be empty, wasting space. Each edge-list node also needs a pointer to the next edge-list node. However, by using this extra memory, the allocator

* The row marked "AVERAGE" is the average memory savings over all 169 routines in our test suite.

† Indeed, we theorize that we could use an aggressive live-range splitting scheme like those described in Briggs' thesis[3] to provide benefits similar to Callahan and Koblenz' tiling method, as the splits would provide natural places to further subdivide the nodes in the interference graph.

Routine Name	Number of Integers	Number of Floats	Memory Savings	Routine Name	Number of Integers	Number of Floats	Memory Savings
advbndX.i	763	13060	10%	pdiagX.i	721	1807	41%
bcndX.i	95	1803	10	prophy.i	211	1455	22
bcndbX.i	102	1010	17	putbX.i	495	1264	40
bcndlX.i	112	1023	18	radb2X.i	696	685	50
bcndrX.i	114	1052	18	radb3X.i	801	1013	49
bcndtX.i	104	1037	17	radb4X.i	1079	1389	49
celbndX.i	658	861	49	radb5X.i	1136	1849	47
ddeflu.i	409	3415	19	radbgX.i	2311	2419	50
debflu.i	435	3679	19	radf2X.i	702	677	50
deseco.i	891	5504	24	radf3X.i	803	1012	49
energyX.i	383	2476	23	radf4X.i	1083	1410	49
erhs.i	843	1334	47	radf5X.i	1140	1842	47
fieldX.i	3564	11669	36	radfgX.i	2436	2516	50
fpppp.i	386	4194	15	rhs.i	1096	1239	50
getbX.i	608	1639	40	rinjX.i	63	1739	7
inibndX.i	185	1460	20	slv2xyX.i	534	927	46
iniset.i	759	1308	47	smoothX.i	1299	1604	49
initX.i	936	13668	12	ssor.i	212	799	33
injbatX.i	178	3837	9	supp.i	13	988	3
injchkX.i	142	1489	16	svd.i	594	864	48
jacld.i	681	1030	48	transX.i	227	4799	9
jacu.i	531	749	48	twldrv.i	1067	16755	11
linjX.i	62	1709	7	vslv1pX.i	1206	1577	49
parmovX.i	2663	21781	19	vslv1xX.i	1099	1438	49
parmveX.i	1756	11310	23	waveX.i	88	2033	8
parmvrX.i	2818	23453	19	yeh.i	86	960	15
				AVERAGE:			36%

Table II. Space Impact of a Split Interference Graph

can avoid the initial pass over the code and its overhead, including clearing and setting the bit matrix an extra time.

Our implementation already allowed the edge lists to be chained together for another reason. After the interference graph has been built, the allocator attempts to coalesce (or combine) values connected by a COPY instruction. If two values are connected by a COPY and they do not otherwise interfere, they can be combined and the COPY can be deleted. In the interference graph, this replaces two nodes with distinct edge lists by a new node with their combined edge list. To save an allocation followed by two frees, the allocator simply links the edge lists together. All subsequent phases of the allocator recognize these linked lists and handle them correctly. Thus, the first edge-list pointer was already present, as was the code to handle them correctly throughout the allocator.

Implementation

For our experiments, we implemented both the original Chaitin-Briggs approach and the split bit-matrix method. Splitting the interference graph into a floating-point graph and an integer graph saves both memory and time. Changing the representation of the edge lists trades running time of the allocator against the use of additional memory. We present measurements of both techniques in the Experimental Results and Observations section. The memory savings from using separate graphs more than compensates for the memory wasted in the fixed-size edge lists. Eliminating the extra pass required for precise-sized edge lists is a clear speed improvement. Thus, the split bit-matrix method results in an allocator that is both faster and smaller than the original Chaitin and Briggs scheme.

Hash Table

George and Appel, in their paper on iterated register coalescing, recommend using a hash table in place of the bit matrix.[4] Using a hash table to replace the bit matrix is attractive because the hash table requires E table entries, where the bit-matrix needs $N(N-1)/2$ entries. If the bit-matrix is sparse, implying that $E \ll N^2$, the hash table will have many fewer entries than the bit matrix.

Because of our long-standing interest in the design and implementation of graph coloring register allocators, we were interested in their suggestion. However, we were concerned that the tradeoff between the two approaches was not as clear cut as the previous argument suggests.

- The characteristics of interference graphs that determine sparsity—like size, average degree, maximum degree, and minimum degree—vary over a wide range. These properties depend on the source language, the compiler, run-time conventions, and other translation issues. Since the benefits of hashing rely on sparsity, the variation in characteristics might complicate the choice between the hash table and the bit matrix.
- Hashing relies on expected-case linear behavior. From the speed perspective, the hash-table lookup must compete against a bit-matrix lookup. The latter operation is quite simple, and most compilers do a good job of optimizing it. For hashing to win, the hash function must retain its linear behavior and the lookup must require just a few instructions.

To investigate these issues, we implemented several hash table schemes in our allocator and made a series of measurements to assess the tradeoffs.

The differences between the techniques are clear. The hash table requires space proportional to E; the bit matrix requires space proportional to $N(N-1)/2$. Each hashed access requires at least one evaluation of the hash function; each bit-matrix access requires evaluating the address polynomial $(low + (high \times (high-1))/2)$ and a mask operation to reach the appropriate bit.* Poor behavior from the hash function may lead to more work; the bit matrix avoids this by spending extra space.

This creates an interesting implementation dilemma: there are a number of different hashing methods with important tradeoffs in expected time and space requirements, not to mention a plethora of possible hashing functions. Thus, simply comparing one hash table implementation against the bit-matrix implementation would not supply enough information to draw reasonable conclusions. For our experiments, we implemented two different styles of hash tables, using a variety of different hash functions. We report results for three specific hash functions: the one used by George and Appel, along with the two best-performing functions that we discovered.

* Recall that we obtain $high \times (high-1)/2$ from a precomputed table.

Routine Name	Node Count	Edge Count	Edges Per Node	Routine Name	Node Count	Edge Count	Edges Per Node
fieldX.i	5936	268353	45.21	putbX.i	533	13685	25.68
parmvrX.i	5614	204491	36.43	efill.i	529	22140	41.85
parmovX.i	5220	195009	37.36	pastem.i	510	7657	15.01
fpppp.i	3899	723605	185.59	tomcatv.i	497	20907	42.07
twldrv.i	3135	309340	98.67	pintgr.i	485	5819	12.00
radfgX.i	2788	148129	53.13	inisla.i	481	7720	16.05
parmveX.i	2697	101962	37.81	cosqb1X.i	468	9888	21.13
radbgX.i	2651	156978	59.21	cosqf1X.i	467	11193	23.97
deseco.i	2271	60569	26.67	dyeh.i	458	10356	22.61
radb5X.i	1968	84833	43.11	prophy.i	432	14371	33.27
radf5X.i	1967	84785	43.10	drepvi.i	427	7399	17.33
rhs.i	1826	75234	41.20	debico.i	427	4269	10.00
smoothX.i	1776	74771	42.10	orgpar.i	406	4974	12.25
erhs.i	1769	57707	32.62	yeh.i	397	4937	12.44
radf4X.i	1589	65860	41.45	heat.i	390	4201	10.77
radb4X.i	1562	65070	41.66	rkfs.i	386	6333	16.41
vslv1pX.i	1559	50308	32.27	bilan.i	345	5982	17.34
advbndX.i	1557	44711	28.72	denptX.i	307	3947	12.86
vslv1xX.i	1511	46909	31.05	rfftf1X.i	295	5987	20.29
initX.i	1422	51962	36.54	decomp.i	294	5168	17.58
ddeflu.i	1378	33509	24.32	fftbX.i	284	7022	24.73
jacld.i	1271	134765	106.03	repvid.i	281	4733	16.84
radf3X.i	1164	39162	33.64	injbatX.i	280	3288	11.74
radb3X.i	1161	39409	33.94	injallX.i	273	2385	8.74
pdiagX.i	1122	17574	15.66	fftfX.i	269	6741	25.06
svd.i	947	20418	21.56	rffti1X.i	259	5306	20.49
jacu.i	933	74914	80.29	blts.i	253	9104	35.98
paroi.i	916	15712	17.15	transX.i	251	4927	19.63
iniset.i	915	8028	8.77	rfftb1X.i	249	4378	17.58
energyX.i	915	14061	15.37	buts.i	247	8010	32.43
debflu.i	912	18111	19.86	ssor.i	238	7983	33.54
radf2X.i	869	26829	30.87	dcoera.i	238	1894	7.96
radb2X.i	869	27206	31.31	putdtX.i	237	2909	12.27
				AGGREGATE	91870	3847704	41.88

Table III. Edge and Node Counts for the Largest of the Routines Studied

Hash-table structure

The main consideration when choosing the hash-table data structure is to decide how collisions will be handled. Collisions occur when two values hash to the same slot in the hash table. Under *open hashing* or *chaining*, each entry in the table is a linked list that contains all of the values that hash to that entry.[14, 15] Under *open addressing*, also called *closed hashing*, each entry in the table holds a single value.[16, 14] When collisions occur, the new value is put at some offset away from its hash entry. If another collision occurs at that new point, the value moves to an offset from that position, and so forth. The offset is either a constant (*linear probing*), or can be computed using a secondary hashing function (*double hashing*).[14-16, 6]

Initial size

Regardless of the data structure employed, the final performance of the hash table will depend heavily on the initial size of the table. If the table is too large, its space advantage over the bit matrix method disappears. If the table is too small, the lookup time increases and the allocator's performance suffers. At the time that the allocator must decide the table size, it only knows the number of nodes in the graph. The size of the table and its resultant efficiency, however, rely on the number of edges, a number which is unknown until the graph is built. Thus, in our experiments, the table size selected is based on the average number of edges per node, as measured by an empirical analysis of a sample set of codes.

George and Appel found that the ratio of E/N in their environment averaged sixteen.[4] Table III shows some of this data for the routines with the largest graphs in our test suite. Considering just the initial graphs, we found a ratio of E/N of about forty-two;[*] the "AGGREGATE" number includes the initial graph for each of the 169 Fortran routines in the test suite. To make matters worse, the ratio varied significantly, from less than one[†] to one hundred and eighty five. Additionally, we were unable to find a simple function that would predict a good initial table size—that is, provide a value for E given N.[‡]

A second problem complicates this issue. The allocator may require more than one attempt to construct a k-coloring. If it cannot construct a k-coloring, it selects some values to spill (*i.e.*, keep in memory rather than registers), modifies the code to reflect these decisions, and begins the allocation cycle again on this modified code. The subsequent interference graphs can have different E/N ratios. For example, including all 1,250 graphs built by the allocator for the 169 routines in the test suite drops E/N to twenty-two, a decrease of almost one-half. While, in general, E/N decreases on subsequent buildings of the interference graphs, the ratio can increase from one graph to its successor. The complex interactions between the allocator's copy-coalescing phase and its spill-code insertion phase have subtle effects on the register pressure in different regions of the code. These, in turn, change the shape of the interference graph. Thus, if we could establish a good metric for initial table size, it would almost certainly need to change across the different interference graphs built in a single allocation.

Since the compiler is unlikely to choose the perfect table size, our hash-table implementations allow for resizing. Performance degrades sharply for hash tables that become too full, so if our initial guess at the size of the hash table is wrong, we allocate a new table that is roughly twice the size of the current hash table[§] and then rehash all of the items into it, theoretically improving our performance at a small amortized cost.

Hash functions

Because performance also depends on the behavior of the hash function, we experimented with many different functions before selecting the three presented here. The input to the function presents its own problems. Edges in the graph are represented as an ordered pair of integers, not arbitrarily long strings. The integers are relatively small, typically under 10,000. Thus, the hashing function does not have many signif-

[*] Remember that the number of edges is the same, whether we build the single hash table or the split hash table, since our implementation does not add interference edges between values of differing types.

[†] This is a very short routine that passes two parameters and simply returns one of them; there are two nodes, but only one interference edge. All the allocators processed this routine quickly.

[‡] Of course, the function $E = N^2/2$ is an upper bound, but it ensures that the hash table has as many entries as the bit matrix. Since hash table entries must be larger than the corresponding bit-matrix entries, $E = N^2/2$ forfeits any space savings.

[§] We precompute a table of prime numbers, each of which is close to a power of two. When the table is expanded, the new table has the next larger size found in the table. Using appropriately chosen prime numbers ensures that all of our closed hashing methods work correctly.

Routine Name	Time to Build Graph (secs)	Total Compilation Time (secs)	% of Time Spent Building Graph
fieldX.i	44.63	85.87	52
smoothX.i	9.60	17.80	54
parmvrX.i	28.20	54.43	52
parmovX.i	25.58	48.86	52
radbgX.i	20.14	36.00	56
twldrv.i	19.92	32.74	61
radf5X.i	17.48	30.39	58
radfgX.i	15.21	27.84	55
fpppp.i	13.58	26.19	52
radb4X.i	15.77	26.09	60
parmveX.i	10.84	19.82	55
radb5X.i	8.92	15.26	58
AGGREGATE	329.74	592.36	56

Table IV. Interference-Graph Build Times for the Split Bit-Matrix Method

icant bits with which to work. Indeed, our first simple hashing functions exhibited undesirable behavior for precisely this reason. After much testing, we selected the following three functions for this paper:

1. $((A \times low) + (B \times high))\ \texttt{mod}\ table\_size$
2. $\lfloor table\_size \times \{[(\{low \ll 16\} \mid high) \times 0.6180339887]\ \texttt{mod}\ 1\}]\rfloor$
3. $(low << 16 + high)\ \texttt{mod}\ table\_size$

The first function is the universal hashing function described by Cormen *et al.*[6] The second function is the multiplicative function described by Knuth.[15] The final function comes from George and Appel's work; their experiments used this function. The first two were the best performing functions that we found. We included data on the third function to allow direct comparison with George and Appel's work.

EXPERIMENTAL RESULTS AND OBSERVATIONS

To evaluate the tradeoffs between the Chaitin-Briggs method, the split bit-matrix method, and the hashing techniques, we implemented the techniques in the register allocator for our laboratory compiler. We compiled each of the 169 Fortran procedures in the test set that we use for regression testing the compiler. These routines are drawn from a number of sources, including the Spec benchmark suite and the library originally written to accompany Forsythe, Malcolm, and Moler's textbook on numerical methods.[8] The tables in the Appendix present detailed results for the largest routines; this section summarizes and interprets those results.

This section presents a series of direct comparisons. First, we derive some simple analytical results to roughly predict the space consumption of the various techniques. Next, we compare the space requirements of Chaitin-Briggs against hashing, of the split bit-matrix method against hashing, and of Chaitin-Briggs against the split bit-matrix method. We examine the speed differences between these three methods, and, finally, look at the impact of the different methods on total allocation time.

Simple Analytical Results

Using simple analysis, we can establish some expectations for the space required by the various methods. The Chaitin-Briggs method constructs a single lower-triangular bit-matrix, without the diagonal. It uses a single bit to represent each possible edge in the graph, which gives it a significant, albeit constant, advantage over the hashing methods. For a graph with N nodes, this always requires $\frac{N(N-1)}{2}$ useful bits or $\frac{N(N-1)}{16}$ bytes.

The hashing methods use at least one word per edge, but they need only represent edges that exist, plus enough overhead space to allow the specific hash function to achieve a reasonable distribution. Thus, they require at least E entries, each occupying a word of memory. Closed hashing uses one word per entry, for a lower bound of E words. Open hashing represents each entry with a word holding the edge's identity, plus a pointer to another entry in the same bucket. This requires $2 \cdot E$ words. Additionally, open hashing has an array of pointers to the linked lists; the size of this array is independent of both N and E.

To compare these bounds, we need to understand the relationship between N and E. Unfortunately, E can only be discovered by building the graph. We can try to estimate E as a function of N, but it varies widely, as shown previously (see Table III). Taken over all 1,250 graphs that the allocator builds for the test suite, E/N averaged twenty-two edges per node. By assuming that $E = 22 \cdot N$, we can estimate the value for N where the memory requirements of the hash table techniques become smaller than those of the bit-matrix method. Ignoring any table expansion done to accommodate excess collisions, the closed table will use $22 \cdot 4 \cdot N$ bytes. Including the array of pointers, the open table will require $22 \cdot 12 \cdot N$ bytes.*

If we graph these lower bounds on space requirements, we see that the closed hash table's equation intersects the bit-matrix's equation when $N = 1409$. Thus, for graphs with fewer than 1409 nodes, with E/N of twenty-two, the Chaitin-Briggs method should use less space. The open hash table's function crosses the bit-matrix function at $N = 4224$. We expect the split bit-matrix method to require less space than Chaitin-Briggs. These numbers are crude approximations; minor changes in E/N can move these cross-over points dramatically, and the lower bounds for the hash table memory usage are likely to be exceeded. Thus, actual behavior will almost certainly vary from the predicted behavior. However, these numbers provide a starting point for our analysis.

Space Comparisons

To compare the actual space requirements, we measured the memory usage of all the implementations. The results are summarized in Tables V through VII in the Appendix.

Chaitin-Briggs versus Hashing Table V compares the Chaitin-Briggs method, which builds a single table, against the various hash-table implementations. It shows that Chaitin-Briggs requires less memory than hashing for all but the largest graphs in

* Our implementation expands the hash table if the length of a collision-induced chain gets too long. In the closed table, this expands the table and rehashes into it. In the open table, this expands the pointer array and rehashes into it. On our test routines, the open table never rehashed; the average list length was almost always less than two.

our study. For the largest examples, closed hashing with either the universal or the multiplicative hash function demonstrates consistent improvement over Chaitin-Briggs. The analytical results suggest that open hashing will eventually show improvement as well, but that closed hashing will always use less space than open hashing.

The Split Bit-Matrix Method versus Hashing Table VI compares the split bit-matrix method, with thirty-element edge lists, against the various hash-table implementations. The split bit-matrix method beats hashing on 168 out of the 169 routines; we would expect it to outperform Chaitin-Briggs in this test. (Closed hashing beats the split bit-matrix method on routine `parmvrX`.) Splitting the interference graph changes the constants in the equation, but it does not change the asymptote. For large enough graphs, closed hashing should become the most space efficient technique.

Chaitin-Briggs versus the Split Bit-Matrix Method Table VII compares the memory requirements of the Chaitin-Briggs method against those of the split bit-matrix method. Results are shown for edge-list array sizes of 10, 20, and 30 elements. Longer vectors improve memory efficiency; however, the impact of the space wasted by fixed-size arrays and their links is larger on the smaller graphs. (Splitting the graph should *not* exacerbate this problem since it eliminates no edges.) Also, as we said in the section describing the split bit-matrix method, the memory saved from having two much smaller bit matrices will often more than make up for the small amount of memory wasted in the edge lists. Note that the AGGREGATE numbers give greater weight to large routines since they are simple arithmetic sums.

Speed Comparisons

The previous section examined the implementations from the perspective of memory consumed. The second important difference between these methods is the speed of allocation. To assess the impact of different graph implementations on allocation speed, we measured the time required to build the interference graph under each method. Tables VIII through X compare the time required by the various implementations to build interference graphs. Finally, Table XI shows the impact of the differing implementation techniques on total allocation time.

As many other authors have noted, the overhead costs of heap allocation can be significant in allocation-intensive tasks. For open hashing and the single-size edge lists used in the split matrix method, the cost of allocation might be important. In our implementations, we use an allocator based on Hanson's *Arena* data structure,[17] rather than the standard system calls. This drastically reduces the time spent in allocation and deallocation.

Chaitin-Briggs versus Hashing Table VIII compares the time required to build the interference graph with Chaitin-Briggs against the various hash-table implementations. In addition to requiring more memory, hashing can also be slower. Open hashing is noticeably slower on our example programs; closed hashing is sometimes faster than the Chaitin-Briggs method. The degradation for `fpppp.i` and `twldrv.i` under closed hashing is particularly striking. Of the hash implementations, closed hashing with the universal hash function is the clear winner.

The Split Bit-Matrix Method versus Hashing Table IX compares the time required to build the interference graph with the split bit-matrix method against the various hash-table implementations. An array size of thirty was used for the edge lists. The split bit-matrix method does better against closed hashing on the examples where closed hashing beat Chaitin-Briggs (see Table VIII). Closed hashing still wins in three individual cases.

Chaitin-Briggs versus the Split Bit-Matrix Method Table X compares the interference graph build times for Chaitin-Briggs against the split bit-matrix method. Times for Chaitin-Briggs are given in seconds. In most cases, the split bit-matrix method is faster. Longer edge-list arrays often improve the split bit-matrix method's performance. Likewise, as we explain in the section describing the split bit-matrix method, eliminating the second pass over the code usually results in a speed improvement. A few routines are slower with the modified technique; these are small routines with relatively short allocation times.

Total Allocation Speed Table XI shows the total allocation time for the largest routines. The initial number is for Chaitin-Briggs; the other methods are shown relative to Chaitin-Briggs. We might have expected to see a change in the relative advantage of one method over another if the cost of other operations, like membership testing, varied in a different way than the cost of building the graph. The table shows that this is not the case—the split bit-matrix method is usually faster than the other methods. Closed hashing, with the universal function, beat it on `radb4X.i`. Consulting the other tables shows that closed hashing is faster than the split bit-matrix method for this routine, but uses more space for the graph. Similarly, Chaitin-Briggs beat the split bit-matrix method on `radf3X.i`; the earlier tables show that Chaitin-Briggs wins on time but not space for edge-list arrays of thirty and on space but not time for edge-list arrays of ten.

Conclusions

Building the interference graph has always consumed a large part of the time and space required by a graph-coloring allocator. The experiments described in this paper explore the tradeoffs between several implementation approaches. The results are quite clear. Below some threshold size, the split bit-matrix method is generally smaller and faster than either hashing or the original Chaitin-Briggs method. Above the threshold, the compiler should use closed hashing with the universal hash function. In practice, the threshold will be a function of the specific implementations used and the properties of a "typical" input program. The compiler writer should determine the threshold experimentally and allow the compiler to select between the split bit-matrix technique and the closed hash table when it sees the number of live ranges to be graphed. Because the two implementations provide essentially the same functionality, a common interface should be easy to construct. Thus, retrofitting an existing allocator to select the appropriate data structure should be simple.

Acknowledgments

The authors would like to thank Mark Krentel for his assistance in finding interesting hash functions to test. We also thank Preston Briggs for his tireless patience as a sounding board for ideas and advice. Without help from Lal George, this work would not have moved forward; he was quick with a reply to every question that we asked.

Finally, we thank the members of the Massively Scalar Compiler Group, who wrote and support the large amount of code necessary to enable these experiments: Preston Briggs, John Lu, Rob Shillner, Phil Shielke, Taylor Simpson, Lisa Thomas, and Edmar Wienskowski.

REFERENCES

1. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein, 'Register allocation via coloring', *Computer Languages*, **6**(1), 47–57 (1981).
2. Gregory J. Chaitin, 'Register allocation and spilling via graph coloring', *SIGPLAN Notices*, **17**(6), 98–105 (1982). *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*
3. Preston Briggs, 'Register allocation via graph coloring', *Ph.D. Thesis*, Rice University, April 1992.
4. Lal George and Andrew W. Appel, 'Iterated register coalescing', *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996, pp. 208–218.
5. Preston Briggs, Keith D. Cooper, and Linda Torczon, 'Improvements to graph coloring register allocation', *ACM Transactions on Programming Languages and Systems*, **16**(3), 428–455 (1994).
6. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, 1992.
7. S. S. Lavrov, 'Store economy in closed operator schemes', *Journal of Computational Mathematics and Mathematical Physics*, **1**(4), 687–701 (1961). English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3:810-828, 1962.
8. George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.
9. Jiazhen Cai and Robert Paige, '"Look Ma, no hashing, and no arrays neither"', *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 1991, pp. 143–154.
10. Preston Briggs and Linda Torczon, 'An efficient representation for sparse sets', *ACM Letters on Programming Languages and Systems*, **2**(1–4), 59–69 (1993).
11. Rajiv Gupta, Mary Lou Soffa, and Tim Steele, 'Register allocation via clique separators', *SIG-*

PLAN Notices, **24**(7), 264–274 (1989). *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

12. David Callahan and Brian Koblenz, 'Register allocation via hierarchical graph coloring', *SIGPLAN Notices*, **26**(6), 192–203 (1991). *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*
13. Preston Briggs, Keith D. Cooper, and Linda Torczon, 'Aggressive live range splitting', *Technical Report 90-126*, Rice University, November 1990.
14. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1992.
15. Donald E. Knuth, *The Art of Computer Programming*, volume 3, Addison-Wesley, 1973.
16. Robert Sedgewick, *Algorithms in C++*, Addison-Wesley, 1992.
17. David R. Hanson, 'Fast allocation and deallocation of memory based on object lifetimes', *Software – Practice and Experience*, **20**(1), 5–12 (1990).

APPENDIX: Tables of Data

Space constraints restrict the amount of data shown. However, the AGGREGATE field on each table shows the accumulated totals for all 169 routines studied. The first column of numbers in each table presents the base case; subsequent columns compare other methods against the base case. The comparisons are expressed as ratios. For example, Table V compares the memory requirements of the hashing implementations to the Chaitin-Briggs method. The hashing results report the ratio $\frac{(\textit{bytes for hashing})}{(\textit{bytes for Chaitin-Briggs})}$.

Routine Name	E/N	Chaitin-Briggs (bytes)	Open Hashing: Universal	Open Hashing: Multiplicative	Open Hashing: George & Appel	Closed Hashing: Universal	Closed Hashing: Multiplicative	Closed Hashing: George & Appel
AGGREGATE		12580123	3.37	3.37	3.37	1.99	1.99	6.55
fieldX.i	45.21	2202256	1.21	1.21	1.21	0.95	0.95	3.81
parmvrX.i	36.42	1969812	1.10	1.10	1.10	0.53	0.53	2.13
parmovX.i	37.36	1703025	1.22	1.22	1.22	0.62	0.62	2.46
fpppp.i	185.59	950137	6.64	6.64	6.64	4.41	4.41	17.66
twldrv.i	103.61	614264	4.88	4.88	4.88	3.41	3.41	13.66
radfgX.i	52.65	485809	2.98	2.98	2.98	2.16	2.16	8.63
parmveX.i	37.80	454613	2.37	2.37	2.37	1.15	1.15	4.61
radbgX.i	59.20	439237	3.46	3.46	3.46	2.39	2.39	9.55
deseco.i	26.57	322340	2.32	2.32	2.32	0.81	0.81	3.25
radb5X.i	43.08	242064	3.89	3.89	3.89	2.17	2.17	8.66
radf5X.i	43.08	241818	3.89	3.89	3.89	2.17	2.17	8.67
rhs.i	41.19	208392	4.15	4.15	4.15	2.52	2.52	10.06
smoothX.i	42.09	197136	4.36	4.36	4.36	2.66	2.66	10.64
erhs.i	32.61	195585	3.70	3.70	3.70	1.34	1.34	5.36
radf4X.i	41.44	157807	5.00	5.00	5.00	3.32	3.32	6.64
radb4X.i	41.65	152490	5.13	5.13	5.13	1.72	1.72	6.88
vslv1pX.i	32.27	151905	4.37	4.37	4.37	1.73	1.73	6.90
advbndX.i	29.43	151515	4.09	4.09	4.09	1.73	1.73	6.92
vslv1xX.i	31.04	142695	4.47	4.47	4.47	1.84	1.84	7.35
initX.i	36.51	126380	4.33	4.33	4.33	2.07	2.07	8.30
ddeflu.i	23.28	118680	3.36	3.36	3.36	2.21	2.21	4.42
jacld.i	105.98	100965	11.98	11.98	11.98	10.39	10.39	20.77
radf3X.i	33.62	84681	5.25	5.25	5.25	3.09	3.09	6.19
radb3X.i	33.92	84245	5.30	5.30	5.30	3.11	3.11	6.22
pdiagX.i	15.60	78680	3.45	3.45	3.45	1.67	1.67	3.33
svd.i	21.54	56050	5.25	5.25	5.25	2.34	2.34	2.34
jacu.i	80.25	54405	13.43	13.43	13.43	9.64	9.64	38.55
paroi.i	17.09	52441	4.90	4.90	4.90	2.50	2.50	2.50
iniset.i	8.80	52326	3.73	3.73	3.73	2.51	2.51	2.51
energyX.i	15.37	52326	4.65	4.65	4.65	2.51	2.51	5.01
debflu.i	19.54	51984	5.31	5.31	5.31	2.52	2.52	5.04
radf2X.i	30.87	47197	7.32	7.32	7.32	2.78	2.78	11.11
radb2X.i	31.30	47197	7.39	7.39	7.39	2.78	2.78	5.55
slv2xyX.i	20.66	44838	6.05	6.05	6.05	2.92	2.92	2.92
celbndX.i	26.42	43995	7.01	7.01	7.01	2.98	2.98	11.92
saturr.i	37.42	34782	10.19	10.19	10.19	3.77	3.77	3.77

Table V. Memory Requirements: Chaitin-Briggs Versus Hashing

Table VI compares the split bit-matrix method against the hash-table methods. An array size of thirty was used for the edge lists.

Routine Name	Split Bit-Matrix (bytes)	Open Hashing Universal	Open Hashing Multi-plicative	Open Hashing George & Appel	Closed Hashing Universal	Closed Hashing Multi-plicative	Closed Hashing George & Appel
AGGREGATE	9549088	4.44	4.44	4.44	2.62	2.62	8.63
fieldX.i	1313217	2.03	2.03	2.03	1.60	1.60	6.39
fpppp.i	1173269	5.38	5.38	5.38	3.57	3.57	14.30
parmvrX.i	1112017	1.94	1.94	1.94	0.94	0.94	3.77
parmovX.i	972020	2.14	2.14	2.14	1.08	1.08	4.32
twldrv.i	503571	5.96	5.96	5.96	4.16	4.16	16.66
radfgX.i	453014	3.19	3.19	3.19	2.31	2.31	9.26
radbgX.i	418153	3.63	3.63	3.63	2.51	2.51	10.03
parmveX.i	311149	3.46	3.46	3.46	1.68	1.68	6.74
deseco.i	202970	3.68	3.68	3.68	1.29	1.29	5.17
radf5X.i	174669	5.38	5.38	5.38	3.00	3.00	12.01
radb5X.i	174590	5.39	5.39	5.39	3.00	3.00	12.01
smoothX.i	172173	5.00	5.00	5.00	3.05	3.05	12.18
rhs.i	154417	5.59	5.59	5.59	3.40	3.40	13.58
vslv1pX.i	136871	4.86	4.86	4.86	1.91	1.91	7.66
erhs.i	132801	5.45	5.45	5.45	1.97	1.97	7.90
radf4X.i	129604	6.09	6.09	6.09	4.05	4.05	8.09
radb4X.i	127521	6.14	6.14	6.14	2.06	2.06	8.22
jacld.i	125450	9.64	9.64	9.64	8.36	8.36	16.72
vslv1xX.i	122746	5.19	5.19	5.19	2.14	2.14	8.54
initX.i	104333	5.24	5.24	5.24	2.51	2.51	10.05
advbndX.i	103108	6.01	6.01	6.01	2.54	2.54	10.17
ddeflu.i	89737	4.45	4.45	4.45	2.92	2.92	5.84
radf3X.i	73794	6.02	6.02	6.02	3.55	3.55	7.10
radb3X.i	73693	6.06	6.06	6.06	3.56	3.56	7.11
jacu.i	70492	10.36	10.36	10.36	7.44	7.44	29.75
supp.i	63363	7.75	7.75	7.75	4.14	4.14	16.55
iniset.i	60074	3.25	3.25	3.25	2.18	2.18	2.18
pdiagX.i	54256	5.01	5.01	5.01	2.42	2.42	4.83
radb2X.i	50387	6.92	6.92	6.92	2.60	2.60	5.20
radf2X.i	50331	6.87	6.87	6.87	2.60	2.60	10.42
saturr.i	49245	7.20	7.20	7.20	2.66	2.66	2.66
celbndX.i	46457	6.64	6.64	6.64	2.82	2.82	11.29
svd.i	44166	6.67	6.67	6.67	2.97	2.97	2.97
subb.i	39055	9.36	9.36	9.36	6.71	6.71	6.71
paroi.i	38479	6.67	6.67	6.67	3.41	3.41	3.41

Table VI. Memory Requirements: Split Bit-Matrix Versus Hashing

Table VII compares memory requirements of the Chaitin-Briggs method against the split bit-matrix method. The memory shown includes the total memory allocated to hold the matrix and the bit vectors.

Routine Name	Chaitin-Briggs (bytes)	Split Bit-Matrix Array Size: 10	20	30	Routine Name	Chaitin-Briggs (bytes)	Split Bit-Matrix Array Size: 10	20	30
AGGREGATE	12580123	1.08	0.84	0.76					
fieldX.i	2202256	0.72	0.63	0.60	subb.i	18428	4.24	2.64	2.12
parmvrX.i	1969812	0.67	0.59	0.56	putbX.i	17755	2.18	1.58	1.39
parmovX.i	1703025	0.69	0.60	0.57	efill.i	17490	2.59	1.57	1.25
fpppp.i	950137	2.05	1.44	1.23	pastem.i	16256	1.28	0.94	0.86
twldrv.i	614264	1.36	0.96	0.82	tomcatv.i	15438	2.77	1.69	1.36
radfgX.i	485809	1.25	1.01	0.93	pintgr.i	14701	1.19	0.90	0.82
parmveX.i	454613	0.91	0.74	0.68	inisla.i	14460	1.44	1.02	0.93
radbgX.i	439237	1.32	1.04	0.95	cosqb1X.i	13689	1.87	1.33	1.16
deseco.i	322340	0.82	0.68	0.63	cosqf1X.i	13630	2.05	1.43	1.22
radb5X.i	242064	1.09	0.81	0.72	dyeh.i	13110	2.08	1.47	1.25
radf5X.i	241818	1.09	0.81	0.72	prophy.i	11664	2.52	1.57	1.21
rhs.i	208392	1.12	0.83	0.74	drepvi.i	11395	1.70	1.21	1.02
smoothX.i	197136	1.27	0.97	0.87	debico.i	11395	1.17	0.90	0.82
erhs.i	195585	0.98	0.75	0.68	orgpar.i	10302	1.32	0.93	0.85
radf4X.i	157807	1.26	0.93	0.82	yeh.i	9850	1.50	1.12	1.01
radb4X.i	152490	1.28	0.94	0.84	heat.i	9506	1.29	0.96	0.91
vslv1pX.i	151905	1.23	0.98	0.90	rkfs.i	9312	1.64	1.12	0.97
advbndX.i	151515	0.99	0.75	0.68	bilan.i	7439	1.86	1.27	1.08
vslv1xX.i	142695	1.19	0.94	0.86	denptX.i	5890	1.76	1.23	1.11
initX.i	126380	1.25	0.93	0.83	rfftf1X.i	5439	2.75	1.97	1.56
ddeflu.i	118680	1.03	0.81	0.76	decomp.i	5402	2.18	1.49	1.25
jacld.i	100965	2.65	1.60	1.24	fftbX.i	5041	3.17	2.08	1.76
radf3X.i	84681	1.35	0.99	0.87	repvid.i	4935	2.14	1.33	1.21
radb3X.i	84245	1.36	0.99	0.87	injbatX.i	4900	1.72	1.26	1.06
pdiagX.i	78680	0.93	0.76	0.69	injallX.i	4658	1.50	1.08	1.05
svd.i	56050	1.13	0.85	0.79	fftfX.i	4522	3.38	2.22	1.87
jacu.i	54405	2.73	1.65	1.30	rffti1X.i	4192	2.83	1.86	1.57
paroi.i	52441	1.01	0.80	0.73	blts.i	4000	4.43	2.65	2.09
iniset.i	52326	1.27	1.15	1.15	transX.i	3937	2.94	2.06	1.59
energyX.i	52326	0.96	0.76	0.71	rfftb1X.i	3875	2.86	2.05	1.56
debflu.i	51984	1.07	0.81	0.71	buts.i	3813	4.11	2.47	1.93
radf2X.i	47197	1.65	1.21	1.07	ssor.i	3540	4.57	2.83	2.18
radb2X.i	47197	1.66	1.21	1.07	dcoera.i	3540	1.58	1.22	1.20
slv2xyX.i	44838	1.19	0.91	0.82	putdtX.i	3510	2.33	1.68	1.51
celbndX.i	43995	1.56	1.16	1.06	inibndX.i	3422	2.33	1.60	1.39

Table VII. Memory Requirements: Chaitin-Briggs Versus Split Bit-Matrix

Table VIII compares the time required to build the interference graph with Chaitin-Briggs against the various hash-table implementations.

Routine Name	Chaitin-Briggs (seconds)	Open Hashing: Universal	Open Hashing: Multi-plicative	Open Hashing: George & Appel	Closed Hashing: Universal	Closed Hashing: Multi-plicative	Closed Hashing: George & Appel
AGGREGATE	852.45	1.83	2.20	2.39	1.26	1.37	46.26
fieldX.i	102.75	1.51	1.75	1.61	1.54	1.62	87.35
smoothX.i	85.23	1.68	2.02	1.61	0.28	0.32	3.80
parmvrX.i	82.99	1.38	1.59	1.35	0.63	0.70	36.26
parmovX.i	73.42	1.41	1.63	1.38	0.70	0.75	29.32
radbgX.i	50.12	1.65	1.90	2.62	1.24	1.33	26.66
twldrv.i	46.07	2.34	2.71	4.54	3.06	3.21	106.05
radf5X.i	42.34	1.97	2.43	1.85	0.53	0.57	9.13
radfgX.i	38.34	1.68	1.93	1.97	1.47	1.65	54.69
radb4X.i	34.56	2.04	2.48	3.16	0.51	0.57	7.00
fpppp.i	32.87	2.69	3.04	9.98	3.21	3.41	341.00
parmveX.i	28.41	1.54	1.85	1.55	0.72	0.86	17.92
radb5X.i	20.60	2.10	2.59	2.08	1.09	1.18	19.29
deseco.i	14.43	1.89	2.27	1.92	1.15	1.30	9.86
jacld.i	12.71	2.89	3.47	4.28	2.67	3.50	70.72
vslv1xX.i	12.24	1.86	2.21	1.99	1.27	1.37	15.33
vslv1pX.i	11.70	1.87	2.20	1.96	1.35	1.43	31.82
rhs.i	11.47	1.99	2.38	2.06	1.49	2.07	29.15
initX.i	11.38	1.86	2.22	1.98	1.35	1.61	15.32
radf4X.i	11.27	2.23	2.82	2.25	1.51	1.69	20.08
erhs.i	11.18	1.96	2.24	1.92	1.36	1.62	9.97
advbndX.i	10.50	2.01	2.54	2.13	1.40	1.56	21.40
radb3X.i	7.80	2.17	4.83	2.12	1.12	1.29	12.10
ddeflu.i	7.51	2.28	2.82	2.35	2.99	1.69	8.28
jacu.i	7.17	2.75	3.34	3.68	2.47	2.93	67.93
radf3X.i	6.06	2.35	2.79	2.23	1.47	1.63	18.66
svd.i	5.99	1.78	2.09	1.73	0.92	1.03	2.67
pdiagX.i	4.61	1.80	2.14	1.75	0.94	1.06	6.83
radb2X.i	4.00	2.12	2.46	2.09	1.49	1.69	6.64
radf2X.i	3.90	2.18	3.42	2.15	1.49	1.89	8.02
energyX.i	3.65	2.09	2.67	1.97	0.74	0.88	5.90
debflu.i	3.01	2.24	2.68	2.08	1.28	1.42	7.52
supp.i	2.77	2.94	3.55	3.88	2.50	3.04	22.26
paroi.i	2.45	2.04	2.41	2.22	1.17	1.38	4.71

Table VIII. Interference-Graph Build Times, Chaitin-Briggs Versus Hashing

Table IX compares the time required to build the interference graph with the split bit-matrix method against the various hash-table implementations. An array size of thirty was used for the edge lists.

Routine Name	Split Bit-Matrix (seconds)	Open Hashing			Closed Hashing		
		Universal	Multi-plicative	George & Appel	Universal	Multi-plicative	George & Appel
AGGREGATE	592.36	2.64	3.16	3.44	1.81	1.98	66.57
fieldX.i	85.87	1.81	2.09	1.93	1.85	1.93	104.53
parmvrX.i	54.43	2.10	2.43	2.06	0.97	1.07	55.28
parmovX.i	48.86	2.11	2.44	2.07	1.05	1.13	44.06
radbgX.i	36.00	2.30	2.64	3.65	1.73	1.85	37.12
twldrv.i	32.74	3.29	3.81	6.39	4.31	4.51	149.22
radf5X.i	30.39	2.74	3.39	2.58	0.73	0.80	12.73
radfgX.i	27.84	2.31	2.66	2.71	2.03	2.28	75.32
fpppp.i	26.19	3.37	3.81	12.52	4.02	4.28	427.97
radb4X.i	26.09	2.70	3.29	4.18	0.67	0.76	9.28
parmveX.i	19.82	2.21	2.65	2.23	1.03	1.24	25.69
smoothX.i	17.80	8.07	9.65	7.70	1.35	1.51	18.19
radb5X.i	15.26	2.83	3.50	2.80	1.47	1.60	26.05
jacld.i	10.66	3.44	4.14	5.10	3.18	4.17	84.32
deseco.i	10.08	2.70	3.24	2.74	1.65	1.86	14.12
erhs.i	9.43	2.32	2.66	2.28	1.61	1.92	11.82
vslv1xX.i	9.28	2.45	2.91	2.62	1.67	1.81	20.22
vslv1pX.i	9.02	2.42	2.85	2.54	1.75	1.85	41.27
rhs.i	8.84	2.58	3.09	2.67	1.94	2.69	37.82
radf4X.i	8.64	2.91	3.68	2.93	1.97	2.20	26.19
initX.i	8.57	2.47	2.95	2.63	1.79	2.13	20.34
advbndX.i	7.62	2.77	3.50	2.94	1.93	2.15	29.49
radf3X.i	7.42	1.92	2.28	1.82	1.20	1.33	15.24
jacu.i	6.31	3.13	3.80	4.19	2.81	3.33	77.19
ddeflu.i	5.48	3.13	3.86	3.22	4.10	2.32	11.35
radb3X.i	4.71	3.59	8.01	3.51	1.85	2.13	20.04
svd.i	4.44	2.41	2.82	2.33	1.24	1.39	3.60
radb2X.i	4.14	2.04	2.38	2.02	1.43	1.63	6.42
radf2X.i	3.93	2.16	3.39	2.13	1.48	1.87	7.96
pdiagX.i	3.30	2.52	2.99	2.44	1.32	1.48	9.54
energyX.i	2.66	2.87	3.67	2.71	1.01	1.21	8.09
supp.i	2.39	3.41	4.11	4.50	2.90	3.53	25.80
debflu.i	2.30	2.93	3.51	2.72	1.68	1.86	9.84
subb.i	1.97	2.85	3.56	3.45	1.83	2.13	4.65
iniset.i	1.97	2.22	2.50	2.83	3.30	3.67	43.50
prophy.i	1.96	3.14	3.83	3.47	1.22	1.41	2.39
celbndX.i	1.94	2.63	3.28	2.82	2.33	2.51	10.25

Table IX. Interference-Graph Build Times, Split Bit-Matrix Method Versus Hashing

Table X compares the interference graph build times for Chaitin-Briggs against the split bit-matrix method.

Routine Name	Chaitin-Briggs (seconds)	Split Bit-Matrix Array Size: 10	20	30	Routine Name	Chaitin-Briggs (seconds)	Split Bit-Matrix Array Size: 10	20	30
AGGREGATE	852.45	0.71	0.71	0.69					
fieldX.i	102.75	0.87	0.87	0.84	efill.i	2.07	0.78	0.79	0.75
smoothX.i	85.23	0.24	0.19	0.21	slv2xyX.i	1.99	0.82	0.79	0.79
parmvrX.i	82.99	0.67	0.66	0.66	getbX.i	1.79	0.86	0.83	0.88
parmovX.i	73.42	0.68	0.66	0.67	putbX.i	1.58	0.81	0.79	0.84
radbgX.i	50.12	0.75	0.75	0.72	drepvi.i	1.45	0.94	0.77	0.74
twldrv.i	46.07	0.60	0.98	0.71	cosqf1X.i	1.45	0.77	0.67	0.68
radf5X.i	42.34	0.31	0.72	0.72	pastem.i	1.42	0.75	0.71	0.73
radfgX.i	38.34	0.76	0.73	0.73	dyeh.i	1.17	1.08	0.79	0.85
radb4X.i	34.56	0.81	0.75	0.75	cosqb1X.i	1.13	0.85	0.79	0.87
fpppp.i	32.87	0.83	0.82	0.80	inithx.i	1.07	0.79	0.78	0.76
parmveX.i	28.41	0.72	0.70	0.70	inisla.i	1.01	0.84	0.81	0.81
radb5X.i	20.60	0.75	0.73	0.74	rkfs.i	0.90	0.84	0.81	0.79
deseco.i	14.43	0.84	0.70	0.70	fftbX.i	0.74	0.93	0.93	0.78
jacld.i	12.71	0.99	0.90	0.84	decomp.i	0.73	0.86	0.92	0.93
vslv1xX.i	12.24	0.79	0.76	0.76	blts.i	0.70	0.87	0.91	0.87
vslv1pX.i	11.70	0.79	0.76	0.77	orgpar.i	0.68	0.74	0.72	0.75
rhs.i	11.47	0.84	0.79	0.77	repvid.i	0.64	0.91	0.77	0.80
initX.i	11.38	0.81	0.75	0.75	fftfX.i	0.63	0.87	0.79	0.92
radf4X.i	11.27	1.86	0.94	0.77	heat.i	0.58	0.95	0.81	0.79
erhs.i	11.18	0.73	0.72	0.84	bilan.i	0.55	0.76	0.76	0.78
advbndX.i	10.50	0.76	0.72	0.73	ssor.i	0.52	0.88	0.88	0.94
radb3X.i	7.80	0.64	0.61	0.60	pintgr.i	0.52	0.77	0.81	0.67
ddeflu.i	7.51	0.94	0.75	0.73	rfftf1X.i	0.51	0.92	0.96	0.86
jacu.i	7.17	0.98	0.91	0.88	rffti1X.i	0.48	0.98	0.85	0.85
radf3X.i	6.06	0.81	0.79	1.22	bilsla.i	0.46	0.80	0.89	0.87
svd.i	5.99	0.73	0.73	0.74	inibndX.i	0.45	0.91	0.91	0.91
pdiagX.i	4.61	0.74	0.72	0.72	transX.i	0.44	0.95	0.91	0.91
radb2X.i	4.00	0.82	1.03	1.03	denptX.i	0.43	0.95	0.93	0.86
radf2X.i	3.90	0.84	0.82	1.01	buts.i	0.43	1.09	0.93	0.93
energyX.i	3.65	0.84	0.75	0.73	colbur.i	0.39	0.69	0.74	0.74
debflu.i	3.01	0.95	0.77	0.76	yeh.i	0.37	0.89	0.73	0.76
supp.i	2.77	0.93	0.91	0.86	ihbtr.i	0.37	1.05	0.86	0.97
paroi.i	2.45	0.76	0.73	0.73	debico.i	0.36	1.03	0.83	0.81
tomcatv.i	2.35	0.79	0.82	0.81	putdtX.i	0.35	0.80	0.80	0.83
prophy.i	2.35	0.84	0.84	0.83	injbatX.i	0.31	0.97	0.97	0.71

Table X. Interference-Graph Build Times, Chaitin-Briggs Versus Split Bit-Matrix Method

Finally, Table XI shows the total time for the entire register-allocation process. An array size of thirty was used for the edge lists in the split bit-matrix method.

Routine Name	Bit Matrix: Chaitin-Briggs (seconds)	Bit Matrix: Split Bit Mat.	Open Hashing: Universal	Open Hashing: Multiplicative	Open Hashing: George & Appel	Closed Hashing: Universal	Closed Hashing: Multiplicative	Closed Hashing: George & Appel
AGGREGATE	865.62	0.68	1.81	2.16	2.35	1.24	1.35	45.55
fieldX.i	98.34	0.87	1.58	1.83	1.68	1.61	1.69	91.27
smoothX.i	86.04	0.21	1.67	2.00	1.59	0.28	0.31	3.76
parmvrX.i	78.02	0.70	1.47	1.69	1.44	0.67	0.75	38.56
parmovX.i	69.54	0.70	1.49	1.72	1.45	0.74	0.79	30.95
radbgX.i	52.31	0.69	1.58	1.82	2.51	1.19	1.27	25.55
twldrv.i	47.17	0.69	2.28	2.65	4.43	2.99	3.13	103.57
radf5X.i	41.79	0.73	2.00	2.47	1.88	0.53	0.58	9.26
radfgX.i	40.01	0.70	1.61	1.85	1.89	1.41	1.58	52.41
radb4X.i	36.22	0.72	1.94	2.37	3.01	0.49	0.54	6.68
fpppp.i	35.08	0.75	2.52	2.85	9.35	3.00	3.19	319.52
parmveX.i	28.38	0.70	1.54	1.85	1.55	0.72	0.86	17.94
radb5X.i	21.00	0.73	2.06	2.54	2.04	1.06	1.16	18.93
deseco.i	14.61	0.69	1.87	2.24	1.89	1.14	1.28	9.74
jacld.i	13.80	0.77	2.66	3.20	3.94	2.46	3.22	65.13
vslv1xX.i	13.08	0.71	1.74	2.07	1.86	1.19	1.29	14.35
vslv1pX.i	12.48	0.72	1.75	2.06	1.84	1.26	1.34	29.83
radf4X.i	12.14	0.71	2.07	2.62	2.09	1.40	1.57	18.64
initX.i	12.05	0.71	1.76	2.10	1.87	1.27	1.52	14.47
rhs.i	11.96	0.74	1.91	2.29	1.97	1.43	1.99	27.95
erhs.i	11.45	0.82	1.91	2.19	1.88	1.33	1.58	9.73
advbndX.i	10.82	0.70	1.95	2.47	2.07	1.36	1.51	20.77
radb3X.i	8.38	0.56	2.02	4.50	1.97	1.04	1.20	11.26
jacu.i	8.17	0.77	2.42	2.94	3.23	2.17	2.57	59.62
ddeflu.i	8.05	0.68	2.13	2.63	2.19	2.79	1.58	7.72
radf3X.i	6.62	1.12	2.15	2.55	2.05	1.35	1.49	17.08
svd.i	6.48	0.69	1.65	1.94	1.60	0.85	0.95	2.47
pdiagX.i	4.64	0.71	1.79	2.12	1.74	0.94	1.05	6.78
radb2X.i	4.45	0.93	1.90	2.22	1.88	1.33	1.52	5.97
radf2X.i	4.35	0.90	1.95	3.07	1.93	1.34	1.69	7.19
energyX.i	3.94	0.68	1.94	2.47	1.83	0.68	0.82	5.46
supp.i	3.24	0.74	2.51	3.03	3.32	2.14	2.60	19.03
debflu.i	3.24	0.71	2.08	2.49	1.93	1.19	1.32	6.99
prophy.i	2.80	0.70	2.20	2.68	2.43	0.85	0.99	1.68

Table XI. Total register-allocation times for each method