Register Allocation in the SPUR Lisp Compiler[†]

James R. Larus
Paul N. Hilfinger

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

ABSTRACT

Register allocation is an important component of most compilers, particularly those for RISC machines. The SPUR Lisp compiler uses a sophisticated, graph-coloring algorithm developed by Fredrick Chow [Chow84]. This paper describes the algorithm and the techniques used to implement it efficiently and evaluates its performance on several large programs. The allocator successfully assigned most temporaries and local variables to registers in a wide variety of functions. Its execution cost is moderate.

1. Introduction

Assigning local variables and compiler-produced temporaries to a limited set of hardware registers is an important task in most compilers.[1] The importance of register allocation is magnified when compiling code that will run on a Reduced Instruction Set Computer (RISC) [Patterson85]. Many RISCs (including MIPS [Hennessy84], RISC II [Patterson82], SOAR [Ungar84], and SPUR [Hill85]) have load-store architectures in which instructions operate only on values that are in registers. The latter three machines also have overlapping register windows, which require many temporaries to evaluate nested function calls properly. For example, to evaluate: f(g(x), h(y)), a compiler must save the result from g(x) in a temporary, so that the evaluation of h(y) can use the single set of outgoing argument registers.

[†] SPUR is sponsored by DARPA under contract N00039-85-C-0269. The first author was supported by a California MICRO Fellowship. Much of this work was done on equipment supplied by a Digital Equipment Corporation CAD/CAM grant.

[1] For convenience, we will refer to both user-defined variables and compiler-produced temporaries as "temporaries" throughout this paper. This simplification is justified since the SPUR Lisp compiler makes no distinction between the two.

Register allocation can be viewed as the process of mapping an unlimited number of temporaries into the finite set of registers provided by a computer. An elegant formalization of this problem is the graph-coloring approach first used by Chaitin [Chaitin81; Chaitin82]. In this method, the nodes in an *interference graph* represent temporaries that must be assigned to registers. Two nodes are connected by an edge if their respective values are live, in the usual dataflow sense, at a statement in the program. The coloring problem is to assign a fixed set of colors to the nodes, subject to the constraint that two nodes connected by an edge cannot be given the same color. When the number of colors is equal to the number of machine registers, a successful coloring will produce an assignment of the temporaries to registers.

Graph coloring is an NP-complete problem, so that a fast and general solution is unlikely to be found. In addition, many graphs cannot be colored because the minimum number of colors required (the *chromatic number*) is greater than the number of available registers. A practical register allocator must produce an assignment in spite of this limitation. Hence, graph coloring register allocators do not look for an optimal coloring, but rather for a correct and feasible one. In particular, they produce an assignment even when a graph's chromatic number exceeds the number of machine registers by causing some temporaries to reside in memory, rather than a register, and introducing *spill code* to load and store these temporaries' values as needed. Thus, the graph coloring allocation problem is really two problems: finding a correct coloring of a given graph and, failing that, deciding which temporaries to spill to enable a coloring.

1.1. Graph Coloring Allocators

Chaitin's technique for finding a coloring is to remove nodes from the interference graph that have fewer than **R** (the number of registers) neighbors since these nodes can be trivially colored. Removing these nodes reduces the number of neighbors of some of the remaining nodes. The process continues until no nodes are left, in which case the graph can be colored by examining the nodes in the reverse of the order in which they were removed, or until all remaining nodes have more than **R** neighbors. In such a situation, Chaitin chooses to spill the temporary with the minimum estimated cost for the added spill code. The spilled node is removed from the graph and the coloring process continues. This technique is successful, in part, because of the large number of registers available to Chaitin: 16 in [Chaitin81] and 32 in [Chaitin82].

Another technique, called *priority-based coloring*, was developed by Chow [Chow83; Chow84]. In this scheme, each temporary is assigned a priority that is the estimated additional cost if the temporary resides in memory rather than in a register. Temporaries with more than **R** neighbors are assigned to registers in decreasing order of priority. A temporary that cannot be assigned to a register because **R** or more of its neighbors have been colored must be split and spill code introduced. A temporary is split by dividing the set of blocks in which the temporary is live into two sets and allocating separately in each. This process is described in detail in Section 2.

1.2. SPUR and SPUR Lisp

SPUR Lisp is an implementation of Common Lisp [Steele84] for the SPUR multiprocessor workstation [Hill85]. Further details of the architecture and Lisp system can be found in the paper by Zorn [Zorn86]. Each SPUR processor is a RISC with a simple load-store architecture.

SPUR's overlapping register windows provide 4 sets of registers for each function. The 10 global registers are dedicated to specific usages and are not allocated on a per-function basis. However, 2 of the global registers are used to load and store values for the spill code.

The register allocator uses 9 of the 10 local registers. The tenth register holds a pointer to a vector containing a function's constants. The allocator also uses any empty incoming argument registers. 4 of the 6 incoming argument registers hold arguments (the others hold a return address and an argument count) and so can be used for allocation if they do not contain actual arguments.

The allocator does not use the outgoing argument registers for two reasons. First, these registers overlap the incoming argument registers of the next call frame and are used to store temporaries in that frame. Second, the allocator currently assumes that a temporary uses a register for an entire basic block in which the temporary is live. If the allocator used the outgoing registers between functions calls to store short-lived temporaries, it would have to allocate over a range of statements that did not coincide with basic blocks, which would complicate the allocation. This change would probably not improve the allocation because short-lived temporaries, having few neighbors, are the easiest type of variables to allocate.

1.3. Overview of the Paper

The rest of this paper describes an implementation of Chow's algorithm and presents measurements that confirm the practicality and effectiveness of this approach. Section 2 briefly describes Chow's algorithm for readers who are unfamiliar with it. Section 3 discusses the efficient implementation of the technique. The final section presents measurements of the allocator's effectiveness and speed.

2. Priority-Based Coloring

Chow's register allocator operates on an interference graph built out of *live-ranges* rather than temporaries. A live range is a pair: *(T, BB)* where *T* is a temporary and *BB* is the *set* of basic blocks in which *T* is live. The interference graph is a graph $G = (LR, E)$, where the vertices, *LR*, are the live ranges corresponding to the temporaries in the program and the arcs are defined:

$$E = \{((T_1, BB_1), (T_2, BB_2)) \,/\, BB_1 \cap BB_2 \neq \{\}\}$$

Figure 1 below contains the algorithm that is the heart of the allocator. It divides the set of live ranges in an interference graph into two sets. *Constrained* nodes are live ranges that have at least **R** neighbors. *Unconstrained* live ranges have fewer than **R** neighbors and so can be trivially colored.

The algorithm orders the constrained live ranges by the projected savings if a live range's temporary resided in a register instead of memory throughout the basic blocks in the live range. The highest priority live range is either colored, if it has fewer than **R** distinctly colored neighbors, or is split.

Splitting a live range requires creating a new live range for the same temporary and moving a subset of the basic blocks from the old to the new live range. This process should reduce the number of colored neighbors of the old live range, thereby facilitating coloring the graph but also requiring the introduction of spill code (see Figure 2). However, creation of a new live range can constrain previously unconstrained nodes and so the algorithm may have to move live ranges between the unconstrained and constrained sets. When all constrained live ranges have been either colored or split into unconstrained pieces, the algorithm trivially colors the unconstrained live ranges.

3. Implementation

This section discusses an implementation of Chow's algorithm. As is usual with this type of algorithm, the data structures used to represent the live ranges and interference graph must be efficient in both time and space so that programs that produce large and complex graphs can be compiled. The structures described below have proven effective in the SPUR Lisp compiler.

3.1. Structure of the Allocator

The SPUR Lisp compiler's register allocator assigns compiler-produced temporaries and user-visible local (lexically-scoped) variables in functions to the available machine registers on a function-by-function basis. The allocator operates directly on assembly code rather than on an intermediate form. The allocator has three distinct sections (see Figure 3).

The allocator first creates a control-flow graph of the function and calculates USE, DEF, and LIVE dataflow information for each basic block. It then creates an interference graph describing the constraints on the allocation of registers and colors it using Chow's algorithm. Finally, it inserts the

```
Unconstrained ← {g ∈ G | #neighbors (g) < R};
Constrained ← G - Unconstrained;

while Constrained ≠ {} do
   foreach c ∈ Constrained do
      if #colored_neighbors (c) ≥ R then
         split live range c into live ranges c, n;
         if #neighbors (n) < R then
            Unconstrained ← Unconstrained ∪ {n};
         else
            Constrained ← Constrained ∪ {n};
      fi

      /* Splitting a node can constrain previously unconstrained nodes */
      foreach u ∈ Unconstrained do
         if #neighbors (u) > R then
            Unconstrained ← Unconstrained - {u};
            Constrained ← Constrained ∪ {u};
         fi
      od

   c ← high_savings (Constrained);
   Constrained ← Constrained - {c};
   color c if possible, otherwise c will have spill code added later;
od

foreach u ∈ Unconstrained do
   color u;
```

#neighbors (lr)	Number of neighbors of lr in G
#colored_neighbors (lr)	Number of colored neighbors of lr in G
high_savings (s)	Returns the uncolored lr with the largest savings if it is not spilled

Figure 1. Chow's graph coloring algorithm. The algorithm splits constrained nodes (those with more than R colored neighbors) and assigns a color to the live range that would benefit most from being in a register. The process terminates when all live ranges have been colored or split into uncolorable and unconstrained pieces.

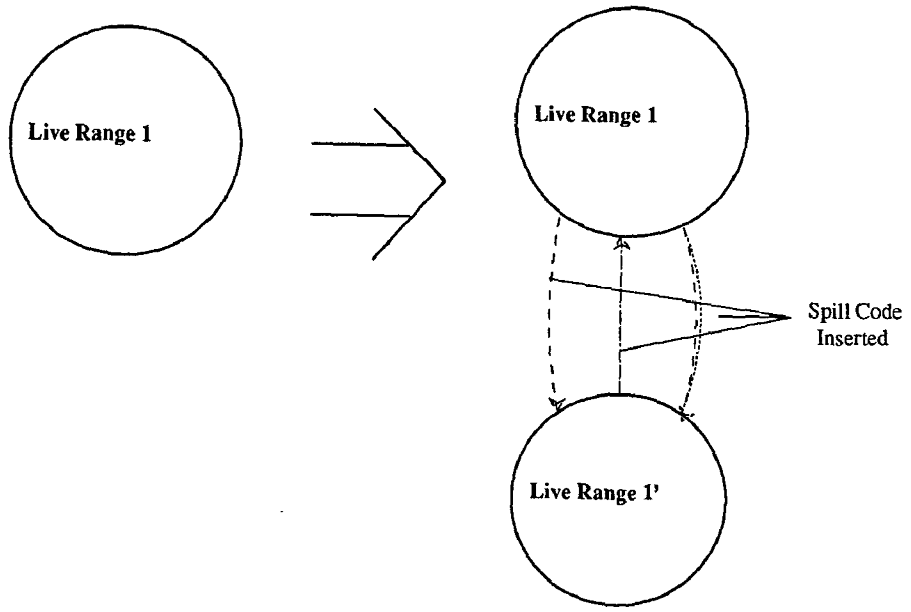

Figure 2. A live range before and after splitting. A live range is split into two live ranges to reduce the number of arcs incident on it. However, when control flow jumps between the two parts of the live range, the temporary must be saved in memory.

load and store instructions needed to spill temporaries.

3.2. Live Ranges

Each node in the interference graph corresponds to a live range and has the structure shown in Figure 4.

The interference graph nodes do not contain a list of arcs. Rather, each live range holds a list of the basic blocks in the live range. Each of these blocks must contain dataflow information about the live variables within the block. This information is conveniently represented as a set of live ranges. Hence, the neighbors of a live range *LR* are simply the union of the live variables in the blocks from *LR* minus *LR* itself. Live variable information is stored as bit-vectors so that this union can be efficiently calculated.

The **number_neighbors**, **number_colored_neighbors**, and **cost_savings** fields in the structure save frequently referenced and infrequently modified information. This information can be calculated in a single pass over the neighbors of a live range or the statements in its basic blocks. However, the numbers change only when a live range is split and are referenced frequently in the allocation process, so that it saves time to cache the values.

3.3. Limiting Basic Block Size

Although Chow's algorithm allocates registers on a per-basic block basis, its performance can be improved by allocating over a smaller piece of the program. Smaller blocks allow a more precise delimitation of the lifetime of a variable and hence produce a less dense and more easily colored interference graph.

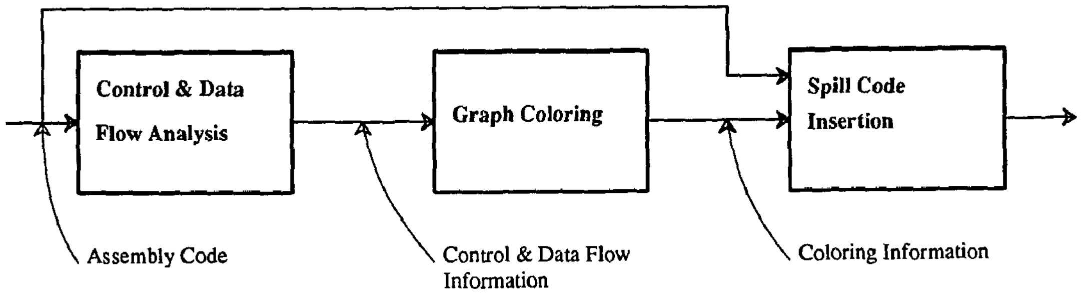

Figure 3. Components of the register allocator. The first stage creates a control-flow graph and calculates dataflow information. The second stage creates a register interference graph and colors it. The final stage inserts the spill code.

```
type live_range =
  record
    temporary: temporary;
    blocks: set_of_basic_blocks;
    number_neighbors: integer;
    number_colored_neighbors: integer;
    cost_savings: integer;
  end;
```

Figure 4. Structure of a live range node in the interference graph. A node's neighbors can be calculated from the live variable information kept in the basic blocks by the algorithm described in Section 3.2.

Rather than change the algorithm to operate at the statement level, we can achieve a similar result by limiting the maximum number of statements in a basic block. A basic block that would naturally contain more statements is prematurely ended and a new block is given the overflow. Measurements in Section 4 show that this approach is very effective in improving the allocator's performance and does not have a large effect on the time required for allocation.

3.4. Splitting a Live Range

Splitting a live range is among the most important actions of the allocator. When a live range is split, the allocator will later have to add spill code to save the temporary when leaving the old live range and restore it when reentering the new live range. The goal of splitting is to produce two large live ranges, each with fewer colored neighbors than the original live range, and a minimal number of jumps between the blocks in one live range and the other. Unfortunately, the problem of splitting a live range to minimize the number of jumps between blocks in the two live ranges is NP-complete.

However, the heuristic shown in Figure 5 works well in practice. It does a breadth-first traversal of the control-flow graph of the basic blocks in the live range being split. The one unusual aspect is that blocks that were artificially split to limit their size are treated as a single block. These blocks have a single successor, as do blocks that end in an unconditional branch. However, distinguishing the two does not improve the allocator's performance. Blocks are moved between the old and new live ranges as long as a block remains in the old live range and the number of colored neighbors of the new live range is less than **R**. The algorithm can fail to reduce the number of colored neighbors of a node, in which case the node will not be colored and will have to have spill code introduced.

3.5. Constrained and Unconstrained Sets

The algorithm in Figure 1 could be simplified considerably if the live ranges were not divided into **Constrained** and **Unconstrained** sets but were just examined and colored or split in order of priority. Unfortunately, this approach produces a significantly worse allocation, with no appreciable savings in execution time.

The reason for this result is clear. Coloring an unconstrained live range can unnecessarily increase the difficulty of coloring a constrained node, perhaps even causing it to be split. In addition, there is no reason to color an unconstrained node before a constrained one, since a color can always be found for an unconstrained node.

3.6. Inserting Spill Code

Spill code must be added to a function in two situations. The first, and simplest, occurs when a live range is split. At each entry point to a split live range in which the temporary is not defined before it is used, the temporary's value must be loaded from memory. At each exit point from the live range in which the temporary is still live, its value must be saved in memory.

```
block ← An entry point to the live range lr;
queue ← {block};

while (queue ≠ {} and #colored_neighbors (new_lr) ≤ R - 1 and #blocks (lr) > 1) do
  block ← head (queue);
  if block is in lr then
      move block from lr to new_lr;
      if block has 1 successor then      /* Block was split or ends with unconditional branch */
        add successor (block) to head of queue;
      else                               /* Block ends with conditional branch */
        add successor1 (block) to tail of queue;
        add successor2 (block) to tail of queue;
      fi
od
```

Figure 5. Heuristic for splitting live range. The algorithm tries to form a new, compact live range by doing a breadth-first traversal of the control-flow graph. It treats blocks that were split to limit their size as a single unit.

	LISP	SLC	RSIM
Number of functions	1,993	1,288	153
Functions needing coloring	780 (39%)	541 (42%)	71 (46%)
Functions needing spill code	104 (5%)	38 (3%)	8 (5%)
Number of spill insts.	1,627 (1%)	862 (0.7%)	400 (3%)
Avg. increase in function size	16 (5%)	21 (5%)	45 (10%)
Avg. number of regs. for allocation	11.3	11.4	11.6

Table 2. Performance of register allocator. The allocator had 9 local registers and any unused incoming argument registers to allocate in each function. Basic blocks were limited to a maximum of 5 instructions.

The other type of spill code is used in live ranges that cannot be allocated a register because they have too many colored neighbors and cannot be split further. On conventional architectures, the instructions in these live ranges could operate directly on the value of the temporary that is stored in memory. However, on SPUR and other RISC machines, this solution is not possible since instructions operate only on registers. In this case, we must insert a load instruction before each use of the temporary and a store instruction after each definition. To ensure that it is always possible to load both operands to an instruction, the allocator needs to reserve two registers from the allocation process.

3.7. Differences from Chow's Allocator

SPUR's allocator differs from Chow's allocator in four aspects described below. These differences are not major and do not affect the coloring process. However, they simplify the allocator.

First, SPUR's allocator assumes that all temporaries reside in registers and adds spill code where necessary to save or restore temporaries. Chow's allocator, on the other hand, assumes that temporaries reside in memory and assigns them to registers. Chow's approach permits his compiler to produce functioning code without running the allocator, which is impossible for the SPUR compiler and also for SPUR's load and store architecture.

Second, Chow's allocator operates on a low-level intermediate representation (instructions to a hypothetical machine) while the SPUR allocator processes SPUR assembly code. Chow's allocator is more portable than SPUR's since its representation is independent of a given machine. However, both representations are machine instructions, their differences are of little importance to the allocation process.

Chow has the advantage of following his allocation by a code selection phase, while the SPUR allocator has to modify the assembly code to reflect the location of temporaries. These modifications are simple for SPUR since operands must be in registers. The allocator can only add load and store instructions and does not have to change addressing modes or handle the special cases of non-orthogonal instructions.

Finally, SPUR's allocator, unlike Chow's, does not do any local (within a basic block) allocation before attempting global allocation. Chow claimed that local allocation would reduce the work required of the slow global allocator. However, he appears to have found that most basic blocks in real programs are short and so most of the allocation was done by the global allocator. Leaving out the local allocator simplified the program.

4. Evaluation of the Allocator

The performance of the allocator was measured on three large programs, which are summarized in Table 1. LISP is the Spice Lisp system from CMU [Wholey84] that has been modified for SPUR. This program is a complete runtime system and interpreter for Common Lisp. SLC is the SPUR Lisp compiler, which is the Spice Lisp compiler from CMU with an additional code generator and register allocator for SPUR. RSIM is a circuit simulator [Terman83]. These three programs may not be typical of all Lisp programs, but they are large, well-written programs (also, they were written by a number of independent authors) whose performance is important.

	LISP	SLC	RSIM
Lines of code	39,912	20,526	2,721
Number of Functions	1,993	1,288	153
Compiled Instructions	168,728	125,872	15,599

Table 1. Programs used to evaluate register allocator. LISP is the SPUR Lisp runtime system. SLC is the SPUR Lisp compiler. RSIM is a circuit simulator.

Table 2 summarizes the allocator's performance on these programs. The allocator used the 9 local registers and any unused incoming argument registers for the temporaries in each function. Basic blocks were limited to a maximum of 5 SPUR instructions. These parameters produced an excellent allocation at a reasonable cost.

In the Lisp system (and similarly in the other two programs), 39% of the functions had more temporaries than registers. The other 61% of the functions did not require allocation. The allocator failed to color 13% of these functions (5% of all functions) and needed to add an average of 17.8 instructions to these functions to save and restore values. The average function that required spill code increased 5% in size.

The number of registers available for allocation in a function varied since empty input argument registers are used by the allocator. An average function had 1.7 incoming arguments, so that the allocator had 11.3 registers to allocate. The results for SLC and RSIM are similar.

4.1. Effect of the Number of Registers

To test the allocator's performance with different numbers of registers, we compiled SLC with the number of local registers artificially limited or increased. Programs compiled with a limited number of registers would execute properly, albeit slowly. However, code compiled with an increased number of registers would not run on the proposed SPUR hardware. Table 3 summarizes the results.

	Number of Local Registers				
	3	6	9	12	15
Funct. needing coloring	831 (64%)	664 (52%)	541 (42%)	435 (34 %)	381 (30%)
Funct. needing spill code	388 (30%)	109 (8%)	38 (3%)	12 (0.9%)	0 (0%)
Number of spill insts.	7503 (6%)	2587 (2%)	862 (0.7%)	237 (0.2%)	0 (0%)
Avg. increase in funct. size	7.7%	6.7%	4.7%	3.4%	0%
Allocation time (CPU min)	10.0 (39%)	8.1 (33%)	6.7 (31%)	5.6 (27%)	5.3 (26%)

Table 3. Measurements of compiling SLC with different numbers of local registers. (The allocator also used empty incoming argument registers.) The first two rows report the number of functions that had more temporaries than registers and the number that required spill code (the percentage is the percentage of all functions.) The third row reports the number of spill instructions (and percentage of all instructions). The fourth row reports the average increase in function size of the functions that required spill code. The last row reports the time (CPU seconds on a VAX 8650 and percentage of total time to compile and assemble the program) to calculate control and dataflow information, create the interference graph, and allocate the registers.

As expected, the allocator performs better and runs faster when given more registers to allocate. However, its performance is acceptable even when given only a few registers.

4.2. Effect of Basic Block Size

The allocator's effectiveness is strongly affected by the size of the basic blocks in the assembly code. A smaller piece of code generally has fewer live variables, and hence fewer neighbors in the interference graph, and is easier to color. Table 4 shows that when basic blocks are limited to smaller numbers of statements, less spill code is needed. The amount of spill code increases dramatically with larger basic blocks, however the absolute amount is still a small percentage of the instructions in the program.

Because of two opposite effects, the running time of the allocator is not reduced by smaller basic blocks. On one hand, smaller blocks mean fewer live ranges must be split and fewer spill instructions inserted, both of which reduce the work that the allocator must do. On the other hand, small blocks also mean that there are more blocks, which increases the time required for flow analysis and to do the calculations mentioned in Section 3.2. However, for a broad range of block sizes, the execution time is roughly constant, so that smaller sized blocks can be used to increase the allocator's performance.

4.3. Number of Split Live Ranges

Table 5 displays the increase in the number of live ranges after the splitting required to color the test programs. As can be seen, the average increase is small. However, it is the maximum that determines the size of the allocator's fixed-length data structures and the allocator's worst-case running time. In the experiments described in Tables 2 and 3, the maximum increase in the number of live ranges never exceeded 70 or 80%. Although pathological cases that require more splitting can be constructed, they do not appear in practice.

	Maximum Instructions Allowed in a Basic Block				
	2	5	10	20	100
Functs. needing coloring	541 (42%)	541 (42%)	541 (42%)	541 (42%)	541 (42%)
Functs. needing spill code	33 (3%)	38 (3%)	49 (4%)	107 (8%)	166 (13%)
Number of spill insts.	775 (0.6%)	862 (0.7%)	1174 (0.9%)	1875 (1.5%)	4141 (3.2%)
Avg. increase in funct. size	5.1%	4.7%	5.4%	5.0%	9.0%
Allocation time (CPU min)	8.3 (35%)	6.8 (31%)	6.3 (29%)	6.3 (29%)	6.7 (31%)
I-graph time (CPU min)	3.9	3.1	2.8	2.6	2.7
Coloring time (CPU min)	3.3	2.9	2.7	2.9	3.2

Table 4. Measurements of compiling SLC with different limits on basic block size. Small blocks reduce the number of live variables in a block and increase the allocator's effectiveness. The first two rows report the number of functions that had more temporaries than registers and the number that required spill code (the percentage is the percentage of all functions.) The third row reports the number of spill instructions (and percentage of all instructions). The fourth row reports the average increase in function size of the functions that required spill code. The fifth row reports the CPU time (CPU seconds on a VAX 8650 and percentage of total time to compile and assemble the program) to calculate control and dataflow information, create the interference graph, and allocate the registers. The last two rows split the allocator's time between creating the interference graph (including flow analysis) and coloring it.

4.4. Execution Time Performance of Allocation

The SPUR Lisp compiler currently produces code that runs on an instruction-level simulator of SPUR. By adding two new instructions to the simulator, we were able to count the number of load and store instructions that moved spilled values between memory and the processor.

Unfortunately, SLC does not yet run on the simulator, so we had to use simpler benchmarks. Three of the well-known Gabriel Lisp benchmarks [Gabriel85] had spill code inserted. The static and dynamic counts of the spill instructions in them and in RSIM is shown in Table 6. Note that the dynamic figures measure the spill code in the benchmarks and any Lisp runtime routines that they invoke. No firm conclusions can be drawn from this small sample. However, it appears that the allocator does a reasonable job of inserting spill instructions to reduce the execution cost of this additional code.

Program	Number of Live Ranges After Coloring Avg. Increase	Max. Increase
LISP	11%	84%
SLC	5%	22%
RSIM	3%	60%

Table 5. Percentage of live ranges that were split. The first column is the average increase in the number of live ranges after coloring. The second column is the maximum increase. Note that these numbers do not measure the percentage of live ranges that were split, since in most programs a few heavily constrained live ranges are repeatedly split.

4.5. Cost of Allocation

Table 7 breaks down the execution cost of the allocator. As can be seen, the allocator requires between 20-30% of the time to compile and assemble a program. However, 40% of this time (8-12% of the total time) is spent calculating control and dataflow information, which can be used for other optimizations within the compiler. The true cost of the allocator appears to be between 10-20% of the execution time of the compiler and assembler. To put this figure in perspective, we should note that the SPUR Lisp compiler is not a highly optimizing compiler.

Program	Static Spill Inst. / Total Inst.	Dynamic Spill Inst. / Total Inst.
BROWSE	1.0%	< 0.01%
FFT	10.0%	3%
PUZZLE	1.2%	< 0.01%
RSIM	2.6%	3%

Table 6. Percentage of instruction cycles consumed by the spill code. The benchmarks are the programs from the Gabriel Lisp benchmarks that required register allocation and from the RSIM circuit simulator.

4.6. Other Measurements

Chow measured the performance of his allocator on six small benchmark programs. His results are difficult to compare to these results because of the differences in the architectures of the target machines. He found that a small number of registers (about 6, depending on the benchmark) reduced the running time of the benchmarks to a minimum. Because the target machines (PDP10, M68000) were not load/store architectures, his optimal allocation did not put all temporaries into registers. On SPUR, execution time is minimized when all temporaries reside in registers, which appears to require more than 6 registers for "real" programs.

5. Conclusion

The SPUR Lisp register allocator is very successful in placing temporaries in registers. The allocation algorithm is simple to understand and performs well on real programs. Its execution cost is moderate for most programs, though particular, complex functions may require considerable time.

	Flow Analysis			Coloring			Total
Program	FG	DF	Total	IG	Color	Total	
LISP	143 (6%)	150 (6%)	12%	198 (8%)	297 (11%)	19%	31%
SLC	43 (3%)	81 (6%)	9%	63 (5%)	173 (13%)	18%	27%
RSIM	4 (2%)	7 (3%)	5%	7 (2%)	31 (11%)	13%	18%

Table 7. Execution cost of the allocator. FG is the time to create a control-flow graph (CPU time in seconds on a VAX 8650 and percentage of total compiler and assembler time). DF is the time to calculate and propagate dataflow information. IG is the time to create an interference graph. Color is the time to color the graph and insert spill code.

Bibliography

[Chaitin81]
Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter Markstein, "Register Allocation Via Coloring," *Computer Languages*, vol. 6, 1981, pp. 47-57.

[Chaitin82]
Gregory J. Chaitin, "Register Allocation & Spilling Via Graph Coloring," *Proceedings of the SIGPLAN '82 Symposium on Computer Construction,* SIGPLAN Notices, vol. 17, no. 6, June 1982, pp. 98-105.

[Chow83]
Fredrick C. Chow, "A Portable Machine-Independent Global Optimizer--Design and Measurements," Technical Note no. 83-254, Computer Systems Laboratory, Stanford University, December 1983.

[Chow84]
Frederick Chow and John Hennessy, "Register Allocation by Priority-based Coloring," in Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices vol. 19, no. 6, June 1984, pp. 222-232.

[Gabriel85]
Richard P. Gabriel, *Performance and Evaluation of Lisp Systems,* MIT Press, 1985.

[Hennessy84]
John L. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers,* vol. C-33, no. 12, December 1984.

[Hill85]
Mark D. Hill, et. al., "SPUR: A VLSI Multiprocessor Workstation," Submitted to *IEEE Computer.*

[Patterson82]
David A. Patterson and Carlo H. Sequin, "A VLSI RISC," *Computer,* vol 15, no. 9, September 1982, pp. 8-21.

[Patterson85]
David A. Patterson, "Reduced Instruction Set Computers," *CACM,* vol. 28, no. 1, pp. 8-21.

[Steele84]
Guy Steele Jr., *Common Lisp,* Digital Press, 1984.

[Terman83]
Chris Terman, "Simulation Tools for Digital LSI Design," MIT Laboratory for Computer Science, Technical Report #304, September, 1983.

[Ungar84]
David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson, "Architecture of SOAR: Smalltalk on a RISC," *Proceedings of the Eleventh International Symposium on Computer Architecture,* June 1984, pp. 188-197.

[Wholey84]
Skef Wholey and Scott E. Fahlman, "The Design of an Instruction Set for Common Lisp," The 1984 ACM Symposium on LISP and Functional Programming, Austin Texas, August 1984, pp. 150-158.

[Zorn86]
Benjamin Zorn, James Larus, George Taylor, and Paul Hilfinger, "SPUR Lisp: Common Lisp on a RISC Multiprocessor," submitted to 1986 ACM Conference on Lisp and Functional Programming.