Journal of Computing and Information Technology - CIT 2, 1994, 2, 113–121

An Approach to Register Number Determination Based on Simulation of Register Allocation via Graph Colouring

Bojana Dalbelo Bašić

Faculty of Forestry, University of Zagreb, Zagreb, Croatia

An important task in most optimising compilers is register allocation i.e. deciding which program variables will use physical registers of processors. Register allocation may be formulated as a graph colouring problem. The nodes in the interference graph represent variables, and two nodes are connected by an edge if variables they represent are simultaneously live. The allocation process is successful if the graph can be coloured so that adjacent nodes are assigned different colours. The number of colours is determined by the number of available registers. Based on simulation of the register allocation procedure via graph colouring this paper discusses the necessary number of registers for storing variables within a single procedure.

Introduction

Deciding which program variables and compiler produced temporaries will be stored in a limited number of physical registers of the processor is one of the most important optimisation tasks of compilers. Efficient register allocation results in faster program execution because:

* the instructions involving register operands are faster and shorter;
* traffic between memory and processor is reduced if operands of an instruction are kept in a register (a previous instructions had either defined them or used them).

The register allocation problem may be interpreted as the problem of mapping of unlimited number of variables onto a limited, fixed number of physical registers of a processor under certain conditions. One of the elegant ways of solving this problem is to formulate it as a graph colouring problem.

The graph colouring method has become one of the prevailing techniques in solving the global register allocation problem since its first application in 1981 (Chaitin et al., 1981) although other methods are being applied in practice as well, e.g. the oldest technique is register allocation via usage counts, and another, more recent one, is the integer linear programming (Luque et al., 1992). The reasons for the prevailing of graph colouring are: simplicity of the method, efficiency of programs on which the register allocation via graph colouring has been applied, and the possibility to honour the characteristics of the processor to which the allocation procedure is adapted. Examples of application of the graph colouring method to solve the register allocation problem for different compilers and different architecture are numerous, some of them being shown in works: (Chaitin, 1982), (Chow and Hennessy, 1984), (Larus and Hilfinger, 1986), (Briggs et al., 1989), (Callahan and Koblenz, 1991).

This paper discusses the number of physical registers in a processor based on the simulation of the register allocation procedure using graph colouring. The first part of the paper explains the concept of interference. The register allocation procedure using graph colouring method based on Chaitin's heuristic (Chaitin et al., 1981) is described next. Application is shown on a simple exemplar program for Fibonacci numbers. Differences in applying the graph colouring method in compilers to RISC and CISC processor architectures are explained.

The main part of the paper describes the simulations of register allocation using Chaitin's heuristic for graph colouring. The initial goal in the simulations was to allocate the largest possible number of variables to registers. Simulation parameters were determined from the results given in (Briggs et al., 1989) and (Bernstein et al., 1989) and the heuristic used was first described in (Chaitin et al., 1981). The upper limit of the number of registers is discussed.

1. Concept of interference

Compiler front-end produces an intermediate code, i.e. a low-level language defined for an abstract computer (thus machine independent) which is easily transformed into the machine or assembly code. The intermediate code is very convenient for transformations and therefore it is the interproduct of optimising compilers. The primary difference between the intermediate code and the assembly code is that the intermediate code does not specify the registers. The intermediate code preferred by many compilers is the *three-address code* (Aho et al., 1986). Three-address instruction has the form: , where , can be constants, names of variables defined by the programmer, or names of compiler generated temporaries, and being an arithmetical or logical operation. As register allocation procedure is performed on an intermediate code, intermediate code names are candidates for residing in registers and they are often called symbolic registers, names, temporaries, or variables. In this paper we shall be using the term *variable*.

The instruction *defines* and uses , (Aho et al., 1986). A sequence of consecutive instructions of the three-address code entered only at its beginning and whose control flow is sequential, without halt or possibility of branching, (except at the end of such a sequence), represents a *basic block* (Aho et al., 1986). A directed graph with nodes representing basic blocks and with directed edges representing control flow between blocks, makes a *flow graph*.

A simple code generator takes a sequence of three-address instructions which form a basic block and generates the target code assuming that all register values must be stored in memory when moving across basic blocks boundaries. Register allocation inside the basic block or a smaller sequential part of the code is *local register allocation* (Aho et al., 1986) and it can be solved efficiently but we are then forced to store register values at the end of each basic block. *Global register allocation* helps reduce the number of LOAD and STORE instructions by defining which variables will stay in the register across block boundaries, so registers are actually allocated for the entire procedure.

The fundamental terms related to register allocation are *variable life* and *interference of variables*. A variable is *live* at a given point in a program if it is previously defined and if there is a path in the flow graph from this point to a certain usage of this variable, otherwise variable is dead (Aho et al., 1986). If two variables

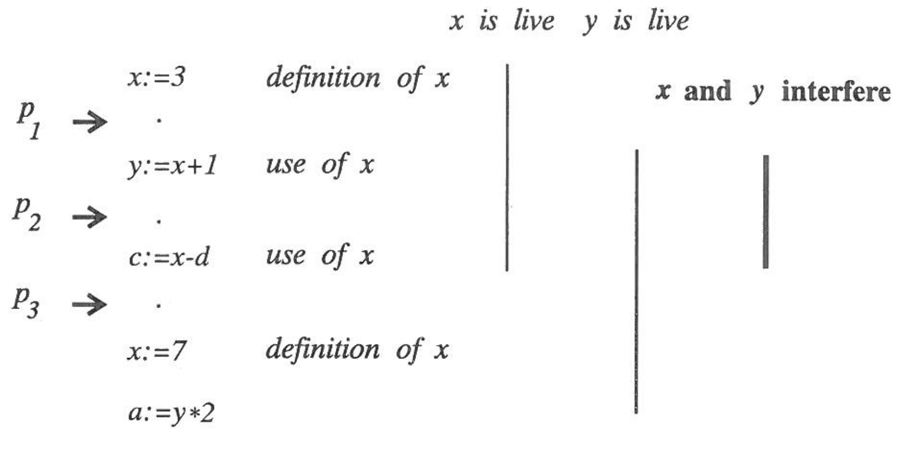

Fig. 1. Example of variable life and interference. Variable is live at points $_1$ and $_2$, dead in the point $_3$

are simultaneously *live* (in data flow sense) they *interfere*. An example of life of variables x and y and their interference is given in Fig. 1.

The main constraint in a register allocation procedure is that the variables which interfere cannot be allocated to the same physical register. In the register allocation procedure using graph colouring, the variables which interfere will represent nodes in the graph connected with an edge.

A more restrictive definition of interference is given in the reference (Chaitin et al., 1981): *two variables interfere if one of them is live at the moment of definition of the other and if both of them do not have the same value*. This definition enables a simpler and cheaper (in the terms of space and time) procedure for the construction of the interference graph. Live variable analysis is performed in the compiler optimisation phase called data flow analysis. By applying various data flow algorithms, compiler gathers information about program in whole and then used them in optimisation actions, code generation and register allocation. A flow graph is the basic structure for global data-flow algorithms. (Aho et al., 1986).

We assume that global register allocation has been carried out after a flow graph had been constructed and live variable analysis accomplished by marking all live variables entering or leaving each basic block.

2. Formulation of the Allocation Problem as a Graph Colouring Problem

The colouring of the graph $G = (V, E)$ is the function ρ from a set of nodes V to a set of colours $\{1, 2, \ldots, k\}$, so that each node is assigned one of k colours. Colouring is regular if adjacent nodes v_i, $v_j \in V$, $\{v_i, v_j\} \in E$, where E is set of edges, have been assigned different colours, i.e. $\rho(v_i) \neq \rho(v_j)$. If such a function exists it is said that the graph G is k-colourable.

From this point in the paper, graph colouring will be understood as regular colouring.

The register allocation problem can be formulated as a graph colouring problem in the following way:

The interference graph is constructed when variables in the program are represented as graph nodes, with an edge existing between two nodes representing interfering variables. The number of colours equals the number of available registers. If the graph can be coloured by k colours, it implies that we have assigned different colours to adjacent nodes, i.e., we have allocated variables which interfere to k physical registers.

Since the problem of k-colourability of graphs is NP-complete (Garey and Johnson, 1979), a heuristic method is introduced to yield satisfactory results in polynomial time.

The introduced heuristic (Chaitin, 1981) is the following: If there is a node v in the graph the degree ow which is strictly less than k, then: G *is* k*-colourable when and only when* $[G \setminus v]$ *is* k*-colourable*.

Based on this simple heuristic, an unlimited number of variables from the intermediate code is tried to be mapped onto a limited number of physical registers.

The register allocation process comprises four steps: construction of the interference graph, graph simplification, spill code insertion, and graph colouring (Fig. 2).

The construction of the interference graph is the phase of stepped pass over intermediate code instructions, and at each definition of a variable a new node to the interference graph is added. Edges which are added to the graph connect the newly introduced node with all currently live variables at the moment of new node's definition. During the graph construction phase the compiler uses results of live variable analysis earliery performed as part of the global data flow analysis (Aho et al., 1986).

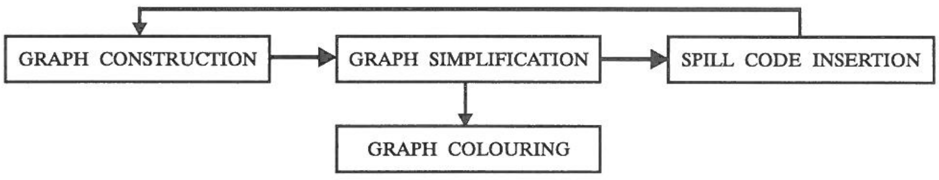

Fig. 2. Phases in register allocation via graph colouring

During the simplification phase, nodes of a degree less than k are consecutively removed from the graph (until such nodes remain in the graph) and are put on the stack for colouring. This process is blocked if only nodes of a degree higher or equal to k remain in the graph. At this point, a node (variable) is selected for which an additional LOAD and STORE instructions will be inserted (LOAD at each use and STORE at each definition of this variable). Such additional code is called *spill code*. The selected variable is also removed from the interference graph and the simplification procedure continues until an empty graph is obtained. After the simplification phase, graph nodes are either on the stack for colouring or are marked for spill code insertion (if there are any).

If after simplification there are no nodes for spill code insertion, this means that k-colouring of the graph is achieved. Nodes are taken from the stack and are assigned a colour which was not already assigned to the adjacent nodes. The way the nodes were eliminated from the graph ensures that each time we take a node from the top of the stack there will be at least one colour not assigned to the adjacent nodes. Namely, if we are able to colour graph G with k colours, then graph G in which we introduce a node v (with a degree less then k) is also k-colourable.

If there are nodes selected for spilling after the simplification phase, it means that the graph construction procedure will be repeated and then the simplification as well. Namely, the life of variables selected for spilling will be split into several short segments. Therefore, instead of a node with numerous edges, we shall introduce into the graph several nodes with a smaller number of edges. Considering the applied heuristic, such a graph will be more suitable for simplification in the next iteration.

Example

The following Pascal module finds the n-th element in the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21...) in which every element is defined as the sum of the preceeding two (except the first two).

```
function FIBON (N:integer): integer;
    var
        FN, FN1, FN2, I : integer
    begin
        if N = 1 then
            FIBON:=1
        else if N = 2 then
            FIBON := 1
        else if N > 2 then
            begin
            FN1 := 1;
            FN2 := 1;
            I := 3;
            repeat
                FN := FN1 + FN2;
                FN2 := FN1;
                FN1 := FN;
                I := I + 1;
            until I > N;
            FIBON := FN;
        end;
    end;
```

In accordance with the definition of the life of variables and the interference of variables, the interference graph for the *Fibonacci numbers* program is shown on Fig. 3.

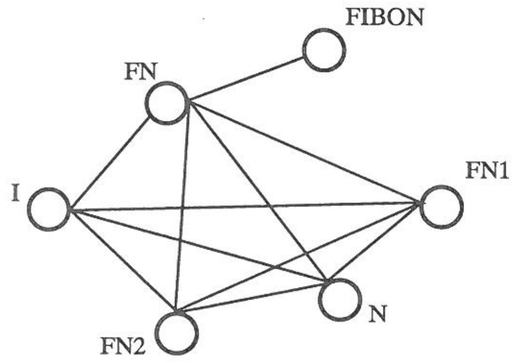

Fig. 3. Graph with chromatic number 5

Obviously, this graph cannot be coloured with 4 colours (it contains a complete subgraph of 5 nodes, so the chromatic number of the graph is 5). Applying the heuristic (Chaitin et al., 1981) for register allocation via graph colouring, an attempt of simplification would have had placed a node FIBON on the stack for colouring, and then found that there were no nodes of a degree less than 4, and therefore would select a node — for instance, FN2 — for spilling.

In the second iteration, in the interference graph constructed based on the intermediate code with a spill code for node FN2, the number of interferences was not decreased. Interferences of node FN2 are determined with currently live variables at the moment of its definition (LOAD instruction), and these are still nodes N, I, FN1, and FN, therefore one more node for spilling, for instance N, should be selected in the second iteration.

Fig. 4 shows the interference graph in the third iteration. There are no more interferences between FN2 and N, since neither of them is live at the moment of definition of the other, so these variables may use the same physical register.

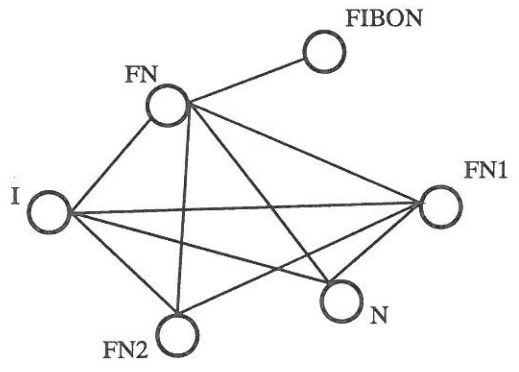

Fig. 4. Graph with chromatic number 4

Simplification of such graph (i.e. successive removal of nodes of a degree less than 4) will place all nodes on the stack for colouring. One of the possible sequences of graph simplification gives the following state on the stack for colouring:

top of the stack ⟹	I	→	R4
	FN	→	R3
	FN1	→	R2
	N	→	R1
	FN2	→	R1
	FIBON	→	R1

Thus, 4-colouring of the interference graph is determined, i.e. all variables from the intermediate code, with inserted spill code for FN2 and N, are allocated to 4 registers. Finally, the series of three-address instructions for the critical part of the *Fibonacci numbers* program, with inserted information on the need to reference FN2 and N from the memory, is as follows:

```
          R2 := 1
          STORE 1, FN2
          R4 := 3
loop:
          LOAD R1, FN2
          R3 := R2 + R1
          STORE R2, FN2
          R2 := R3
          R4 := R4 + 1
          LOAD R1, N
          if R4 > R1 goto end
          goto loop
end: ...
```

3. Differences in Graph Colouring Method Application to RISC and CISC Processor Architectures

The main difference between the application of the graph colouring method in register allocation to compilers for RISC and CISC processor architectures is that for RISC each operand has to be brought to a register before the execution of an operation. It means that the register allocation procedure (Chaitin at al., 1981) is either iterative and only terminates when all operands are in the registers (which is achieved at times by inserting a spill code) or it does not have to be iterative, but then two registers have to be reserved for those values for which a free physical register could not be found (Larus and Hilfinger, 1986). CISC architecture compilers can solve the register allocation problem in this way too, but this is neither essential nor optimal, because of the possibility of using instructions referencing the memory operand. While spilling for RISC means inserting LOAD and STORE instructions as described in (Chaitin et al., 1981) and in this paper, spilling for CISC could be implemented using instructions referencing a memory operand. The application of the graph colouring method for CISC type processors is given in (Chow and Hennessy, 1984). In Chow's priority based procedure each variable is initially placed in the memory and is assigned a priority — an estimated additional cost if a variable resides in the memory rather than in a register. Variables with more than n neighbours are assigned to registers in decreasing order of priority.

It should be pointed out that procedure calls are points in the program at which register pressure is the highest, especially for CISC architecture compilers. For RISC architecture compilers a separate set of registers may be dedicated for the passing of parameters at procedure calls either by hardware (e.g. register windows) or software conventions.

4. Description of Simulations

Simulations were carried out with the aim to test the upper bound of the number of physical registers with the intention of placing, using Chaitin's heuristic, as many variables as possible in registers without spilling.

The efficiency of the allocation was tested by changing the number of available colours (i.e. number of registers), but other parameters as well, such as: number of nodes in the graph (N), degree of nodes in the graph and graph density, i.e. number of edges divided by $(N-1)*N/2$.

Adjacency matrices of interference graphs were generated randomly, the order of these matrices ranging between 10 and 200. As the graph colouring method determines the global register allocation within a single procedure, the selected range agrees with the number of live variables in a predominant majority of medium and smaller procedures (Briggs et al., 1989).

The first test aimes to compare the efficiency of allocation of variables to 8, 16 and 32 registers for the same randomly generated adjacency matrix of the interference graph. The percentage of variable allocation to registers is observed when the order of the matrix grows.

Since the adjacency matrices of the interference graph are sparse, the probability of interference between any two variables (i.e. nodes) equalling 0.3 was chosen. The matrix order N was increased from 10 to 200, with an increment of 10. Adjacency matrices of interference graphs are attempted to be coloured with 8, 16, and 32 colours (representing number of registers). The average percentage of variable register allocation, using Chaitin's heuristic, for sets of four randomly generated adjacency matrices is shown on Fig. 5.

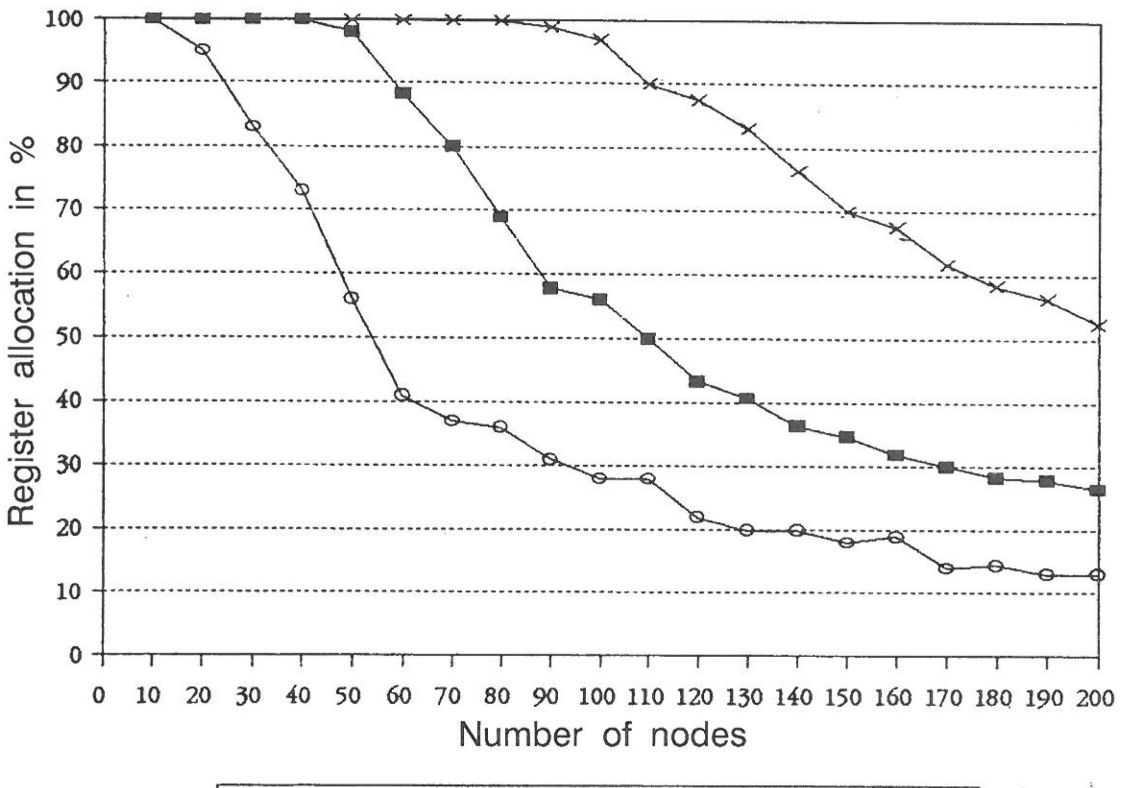

Fig. 5. Results of variable allocation in 8, 16 and 32 registers for graph with 30% density.

The results show that the percentage of allocation to registers for randomly generated graphs with the same probability of interference between nodes is inversely proportional to the order of the adjacency matrix of the interference graph, the results being better the larger is the number of available registers.

It could be stated that the results of variable allocation to registers are relatively poor, and that this is expected and could be explained if the graph node degree is defined as a random variable of binomial distribution with parameters $B(N-1, 0.3)$. The expectation of the node degree is equal for all nodes, which means that all degree values will be about $0.3(N-1)$, since the probability of interference between any two nodes is exactly 0.3.

As Chaitin's algorithm effectively allocates variables to registers only if there are nodes of a degree strictly less than the given number of colours, in the majority of cases the algorithm will be blocked very early. This implies that some elimination from the graph of nodes chosen for spilling will be performed as to reach the node which meets the criterion for register allocation.

If we consider a graph of equal density, but with significant variety in nodes degrees within the graph (just contrary to the test, where the expectation of the degrees was equal for all nodes), then it might be possible to immediately determine the nodes which meet the condition for register allocation and, by their elimination, the degrees of the remaining nodes would dynamically decrease, so that *there is the possibility* of consecutive elimination from the graph of nodes which meet the condition for register allocation.

We would like to point out here the high dependence of the efficiency of the algorithm upon the manner the edges within the graph are distributed for the same N and same graph density. A graph formed in a way that the expectations of the degrees are equal within the graph seems to be the most unsuitable for Chaitin's heuristic algorithm.

The definition of parameters for adjacency matrices which form the input for the algorithm in the next test is based on the quotation from (Berstein et al., 1989):

"Although theoretically any arbitrary graph can arise in this manner, in practice, the interference graphs that are obtained from actual programs are quite sparse. A good (experimentally sound) estimate is that the number of edges will be approximately 20 times the number n of vertices."

In this test, the probability of interference between two variables is not the same for any graph size as it was in the previous test. Since the number of edges in this test is 20 times the number of nodes, the following is valid for the density:

$$\text{density} = \frac{20 * N}{\frac{N * (N-1)}{2}} = \frac{40}{N-1}.$$

Thus, the probability of interference between two variables in this test is set to $40/(N-1)$, which means that the probability of an edge existing between two nodes will be inversely proportional to the number of nodes in the interference graph.

For random generated matrices (N ranging from 50 to 200), with the probability of interferencc between two variables equaling $40/(N-1)$, Table 1 shows the results of variable allocation to registers (in percentages) for a selected number of 16 and 32 registers. Clearly, all deviations of 40%, or 80%, in these tables are random.

The obtained results are explained if we assume that the rows (or columns) in the adjacency matrix are Bernoulli's trials in which the occurrence of one for each graph is given depending on N (precisely $40/(N-1)$) and is constant within the matrix. In other words, the expected number of the degree of nodes is constant regardless to the size of the graph, and equals 40. The nodes in a large graph ($N = 200$) as well as in a small one ($N = 50$) will in average have the same degree which equals 40.

If we choose 64 registers, we shall clearly be able to assign for allocation to registers all nodes, regardless to the size of the graph. If we select a 500 node graph, the graph's density is 8% and the probability of a node occurring with a degree higher than 64 is practically zero. This means that if we choose 64 registers we shall be able to place into registers all variables. The boundary number of registers above which we shall be able to assign practically all variables

Table 1. Results of allocation in 16 and 32 registers

Number of nodes N	Density= $40/(N-1)$	% in 16 registers	% in 32 registers
200	0.20	41.5	82.0
190	0.21	38.4	78.4
180	0.22	41.1	80.6
170	0.24	37.1	79.4
160	0.25	38.8	82.5
150	0.27	41.3	80.6
140	0.29	40.0	76.4
130	0.31	36.9	81.5
120	0.34	41.6	81.6
110	0.37	40.0	80.9
100	0.40	37.0	80.0
90	0.45	38.9	78.9
80	0.51	39.8	78.7
70	0.58	35.7	81.4
60	0.68	35.0	80.0
50	0.82	40.0	78.0

from the program using algorithm (Chaitin et al., 1981) for graph colouring is found to be 40.

This conclusion is drawn from the results of simulations carried out on randomly generated graphs with the equal probability of interference between any two nodes, i.e. with a homogeneous structure, which certainly cannot be expected to be the case with graphs obtained from actual programs. In reference (Bernstein et al., 1989) it is stated that *graphs that are obtained from actual programs do not seem to be structurally similar to random sparse graphs.* This observation has lead to the idea to test whether, for graphs with a higher nonhomogeneity of degrees of nodes (structurally more alike graphs from actual programs), Chaitin's heuristic would yield better results of register allocation than in the previous test.

For the comparison of results, graph densities equal those in the previous test, namely $40/(N-1)$, but the probability of interference between nodes is not the same within the graph. Generated graphs had 10% of pairs of nodes with a higher probability of interference than $40/(N-1)$ and 10% of pair of nodes with a lower probability of interference than $40/(N-1)$. These two probabilities were chosen so that the graph density for the given N remained the same as in the previous test. Thus, graphs were generated consisting of nodes with significantly lower or significantly higher degrees then 40.

Table 2. presents results of register allocation simulation for variables from such interference graphs. As expected, these results are better than the results of the previous test.

Table 2. Percentages of variables allocated to registers for graphs of $40/(N-1)$ density and introduced nonhomogeneity in graph structure.

Number of nodes N	Density = $40/(N-1)$	% in 8 registers	% in 16 registers	% in 32 registers
60	0.68	26.2	57.5	95.0
80	0.51	27.0	56.9	99.4
120	0.34	31.4	63.4	100
150	0.27	34.1	67.5	100
170	0.24	31.7	59.4	100

Results show that changing the graph structure has resulted in significant increase in the percentage of assignment to registers, e.g. *for 32 registers an average of 98.9% of variables* (from graphs generated as described) could be assigned to registers without spilling. Holding equal graph density as in previous test, i.e. $40/(N-1)$, but introducing nonhomogeneity implies that some nodes have lower degree than 40 and some higher than 40. This enables the consecutive elimination nodes with the degree less then a number of registers (32) and by removing them dynamically decreasing degree of other nodes, thus making available for colouring nodes which initially had a degree higher than 32.

An even higher nonhomogeneity in the graph structure and, thus, even better results than in the above simulations, could be intuitively expected for interference graphs generated from actual programs.

5. Conclusion

Based on the evaluation of the density of the interference graphs created from actual programs given in (Bernstein et al., 1989), it can be concluded that the upper bound for the number of registers for which we shall be able to assign almost all variables to registers without spilling using heuristics for graph colouring, is 40. It should be stressed that this conclusion refers to the randomly generated graphs with 40/(N-1) density, for any number of nodes N and with the same expectations of node degrees equalling 40. Choosing the same graph density but introducing nonhomogeneity in graph structure (intuitively expected for interference graphs generated from actual programs) this upper bound of the number of registers is reduced to 32.

An example of different approach for register number determination, based on empirical evaluation of particular Instruction Set Processor (namely DECsystem10) is given in (Lunde, 1977). It is stated that there are no test programs (six CALGO algorithms coded in four different languages) using more then 15 registers simultaneously.

Taking into account that simulations described in this paper were designed to be completely machine independent and with the aim to place as many variables as possible into registers without spilling, it could be expected that the optimal number of registers for global allocation would be less than 32.

Acknowledgements

I express my gratitude to Prof. Slobodan Ribarić, ScD, for his assistance in the development of the simulations and preparation of this work.

References

AHO, A.V., SETHI, R., ULLMAN, J.D. (1986), *Compilers, Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass.

BERNSTEIN, D., GOLDIN, D., GOLUMBIC, C., KRAWCZYK, H., MANSOUR, Y., NASHON, I., PINTER, Y.R. (1989), Spill Code Minimization Techniques for Optimizing Compilers, *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989, pp. 258–263.

BRIGGS, P., COOPER, D.K., KENNEDY, K., TORCZON, L. (1989), Coloring Heuristics for Register Allocation, *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989, pp. 275–284.

CALLAHAN, D. AND KOBLENZ, B. (1991), Register Allocation via Hierarchical Graph Coloring, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26–28 June 1991, pp.192–203.

CHAITIN, G.J., AUSLANDER, M.A., CHANDRA, A.K., COCKE, J., HOPKINS, M.E., MARKSTEIN, P.W. (1981), Register Allocation via Coloring, *Computer Languages*, Vol. 6, No. 1, pp. 47–57.

CHAITIN, G. J. (1982), Register Allocation & Spilling via Graph Coloring, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982, pp. 98–105.

CHOW, F. AND HENNESSY, J. (1984), Register Allocation by Priority Based Coloring, *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 19, No. 6, June 1984, pp. 222-232.

GAREY, M.R. AND JOHNSON, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco.

LARUS, J. AND HILFINGER, P. (1986), Register Allocation in the SPUR Lisp Compiler, *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 21, No. 7, June 1986, pp. 255–263.

LUNDE, A. (1977) Empirical evaluation of some features of Instruction Set Processor Architectures, *Communications of the ACM*, Vol.20, No.3, pp. 143–153.

LUQUE, E., RIPOLL, A., DIEZ T. (1992), Heuristic Algorithms for Register Allocation, *IEE Proceedings-E*, Vol. 139, No. 1, January 1992, pp. 73–80.

Received: December, 1993
Accepted: January, 1994

Contact address:
Bojana Dalbelo Bašić,
Faculty of Forestry,
University of Zagreb,
Svetošimunska 25, Zagreb
Tel. +385-41-218-288
Fax: +385-41-218-616
E-mail: bojana.dalbelo@x400.srce.hr
Croatia

Bojana Dalbelo Bašić graduated in mathematics in 1982 at the Faculty of Natural Sciences and Mathematics, University of Zagreb. In 1993 she received her MSc degree in computer sciences from the Faculty of Electrical Engineering, University of Zagreb. She is currently assistant lecturer at the Faculty of Forestry, University of Zagreb.