Global Register Allocation Based on Live Range Growth and Register Coalesce

Luciana L. Ambrosio[1] **, Mariza A. da S. Bigonha**[1] **,**
Roberto da S. Bigonha[1]

[1]Departamento de Ciência da Computação - Universidade Federal de Minas Gerais, Brazil

leal@dcc.ufmg.br, mariza@dcc.ufmg.br, bigonha@dcc.ufmg.br

Abstract. *Register Allocation is the compiler pass that determines which program values should be assigned to machine registers. Frequently, there are less machine registers than necessary, and consequently, some values should be spilled to memory. An efficient register allocation reduces the number of memory access instructions in the code. However, this is a NP-problem, making it solved through heuristics. The heuristic commonly used is graph coloring. The work presented in this paper implements a new heuristics for register allocation, based in Live Range Growth.*

1. Introduction

Register allocation is one of the most important problems in code optimization. Registers are small, expensive and fast memories, that exist inside the CPU, and keep the most frequently used values of the program being executed. Register allocation is the compiler pass that decides which values should be kept in the registers. Usually, there is few registers in the machine, less than the necessary. Because of this, some values kept in registers must be spilled to memory. To spill a value to memory means that at least one access to memory will be necessary to access this value, which increases the runtime of the program. The problem of finding an optmal register allocation is NP-hard, making it solved through heuristics.

The most widely used solution to the problem is by graph colorint [3]. This approach represents the conflicts between live values[1] with a given number of colors, which represent the avaible number of registers in the machine. The values that have not a color assigned to it, are spilled to memory. In spite of being the most widely used solution, the graph coloring has some problems. For example, to calculate the spill cost of a value, the graph coloring method considers only one basic basic, it does not consider the control flow graph analysis and the data analysis of the program (only the liveness variable analysis). Because of this, the method does not produce an efficient code in some situations. That is the reason why more efficient register allocation technics are being researched and developed [1, 7, 2, 4, 13, 12].

This article presents the research and implementation results of a new register allocation heuristic developed by the algorithm proposed by Ottoni and Araújo [12], which presents a new formulation to solve the global register allocation problem that uses a technic called Live Range Growth. This technics tries to solve the graph coloring problems, implementing: (a) control flow graph analysis, (b) liveness variable analysis, and (c) data

[1]values that will be referenced by the program

analysis called Register Consistency Reachability. A load/store RISC architecture, where all the computations are via registers, except the load and store instructions, are the basis of the implementation.

2. Register Allocation

Several register allocation technics [3, 1, 2, 4, 13, 7, 12, 10, 8] were developed. Despite of this, the widely adopted method in the last decades is the Graph Coloring Method, developed by Chaitin [3]. This approach offers a simplified abstraction of the register allocation problem. In the method, a graph called interference graph is built, in which the nodes represents the register allocation candidates and the edges represent interferences between the candidates. It is said that two candidates interfere if the are both live at the same point in a procedure, and because of the interference, they can not be allocated to the same register. Suppose k is the number of avaiable registers in the machine. If the graph nodes could be colored with k or fewer colors, in such a way that a pair of nodes connected by an edge receives different colors, then the coloring corresponds to the allocation. If a k coloring could not be effected, the code will be modified to try a new coloring. This work was extended by several contributions, for example [1, 7].

Briggs [1] work adds many improvements and extensions to Chaitin [3] work, and the most important one can be described as Optimistic Coloring. Chaitinŝ approach pessimistically assumes that every node with a level higher than the number of avaiable colors will not be colored, and will be spilled. Briggs optimistically assumes that the higher level nodes will also receive colors, reaching smaller spill costs.

The algorithm proposed by George and Appel [7], in spite of being an improvement of Chaitinŝ algorithm, also improves the Briggs performance. The coalesce heuristic developed by Chaitin might turn the graph not colorable. Briggs proposed a conservative coalesce heuristic, a colorable graph before the heuristic application, continued to be colorable after the application of the conservative coalesce heuristic. But the Briggs algorithm is too conservative, leaving many copy instructions in the code of the program. George and Appel method intercalates the simplification pass with the Briggs coalesce heuristic, making the algorithm more aggressive. This method is called Iterated Register Coalescing, and has five phases:

1. Interference graph construction: separates each node like being move related (target or source of a move instruction) or not.
2. Simplification: removes the not move related nodes, whose level are smaller than the number of avaiable registers.
3. Coalesce: realizes the conservative coalesce heuristic in the graph obtained by the prior phase. After the nodes junction, if the resulting node is no longer move related, it becames avaiable for the next execution of the simplification phase. The simplification and coalesce phases are repeated until only significant-degree or move-related nodes remain in the graph.
4. Freeze: if neither simplification nor coalesce apply, the move related nodes are looked. The move in which this node is involved are freezed: this node is not considered a move related node. The simplification and coalesce phases are resumed.
5. Selection: selects colors to the graph nodes.

This algorithm has a better performance than Briggs, eliminates more moves instructions and garantees that no adicional spill will be introduced in the code.

3. Register Allocation Based in Live Range Growth

The solution based in Live Range Growth was proposed by Ottoni and Araújo [12] for specialized embedded processors like *Digital Signal Processors*, DSPs, to solve the problem of allocating address registers to array references in loops using auto-increment addressing mode. Our proposal is to use this method for the global register allocation in general processors. To facilitte the understanding of our implementation, the algorithm for Global Register Allocation Based in Live Range Growth proposed by Ottoni and Araújo is described in details in the rest of this section.

The central problem for obtaining a solution based in Live Range Growth (LRG) for Global Register Allocation, is the calculation of the smallest cost associated with a given live range. A live range is a set of variables attributed to the same register, having the effect that all the uses and definitions of those variables are done by this register. A web is the combination of du-chains (definition-use chains) that intercept, that have a common use [11]. In the beginning of the implementation, each web corresponds to a live range. A heuristic algorithm, called Live Range Growth is then used, suquencially joins pairs of live ranges, until the total number of live ranges is equal to the number of avaiable machines registers in the given architecture. To decide which pair of live ranges will be joined at each iteration, all the combinations of pairs will be evaluated, and the one that results in the smallest cost will be the chosen one. The cost of a live range is measured by the number of loads and stores necessaries to ajust the register. This way, the central problem of this technic is the determination, for a given live range, of the number of the instructions to keep the register with the correct variable during the program runtime.

In this technic, each reference[2] must be preceded by an unique other reference to a variable that belongs to the same live range. This property is important to determine, at each program point, which variable will be attributed to the register allocated to the variables of this live range, independently of the executed path. To satisfy this property, the program is transformed in a representation based in a form called Single Reference Form (SRF). This form was proposed by Cintra e Araújo [5] and it is an adaptation of the Static Single Assignment [6] (SSA) form that has the property that each reference can only be preceded by an unique other reference. Another important concept to understand the technic is related to ϕ function. ϕ function is a special form of attribuition that receives as the argument a set of definitions $\{x_1, ..., x_n\}$ of a variable x that achieves a junction point (a node with more than one predecessor) in the Control Flow Graph (CFG) of the program and produces a new attribution x_{n+1} to the variable x [5]. This way, it is said that $x_{n+1} = \phi\ (x_1, ... , x_n)$.

The first pass of the Global Register Allocation Algorithm Based in Live Range Growth is the inserction of ϕ function in the beginning of each basic block that belongs to the iterated dominance frontier[3] of the basic blocks that use or define some variable that belongs to the live range. It is necessary because an use or definition of a variable forces it to be allocated to the register. In the model proposed by Ottoni and Araújo, two informations are crucial:

1. the determination of which variable that belongs to the live range is attributed to the register associated to this live range in each point in the program;
2. to determine if the variable attributed to the register has an equal or a different value compared to its value stored in the memory. When the values are equal, we

[2] reference is defined as the use or definition of a variable

[3] iterated dominance frontier are the blocks with more than one predecessor or the last block

say that this value is consistent, otherside it is inconsistent. This information is important because a store instruction could be saved if the values in the register and in the memory are equal, and it is necessary that this register be allocated to another variable that belongs to the same live range, in a point in the program.

The next algorithm pass is the calculation of the two information explained above. To calculate the variables attribution to the register associated with a live range in each point in the program and its consistency with the memory, it is used a data flow analysis called Register Consistency Reachability. Suppose V is the set of variables that belongs to the live range, C the consistency with the memory, I the inconsistency with the memory, and X the ignorance if the value stored in the register is consistency or not with the memory; the RCR analysis itens are pairs *(v, e)* where *v* is a variable that belongs to the live range and *e* is the consistency state, in which:

- v $\in$ V $\cup$ V_ϕ $\cup$ $\{\epsilon\}$, where ϵ means that no variable is allocated to the register;
- e $\in$ $\{$C,I,X$\}$ $\cup$ E_ϕ.

The ϕ function result must also be formed by a pair, constituted by a variable V and a consistency state C or I. However, to knowing the solutions of ϕ functions, the RCR results are very important to know which registers states reach a given ϕ function and this way, to avaliate its best solution. So, suppose the sets V_ϕ = $\{\omega_i\}$ and E_ϕ =$\{\sigma_i\}$, the solution of a ϕ function is the pair (ω_i, σ_i). The ω_is and σ_is are the variables of the optimization problem that decrease the number of load and store instructions. This way, we want to choose the solutions of ϕ function that minimizes the cost of the live range.

A set of itens is calculated for each point between two references in the program. For example, in a instruction like x := y + z, if x and y are variables that belong to the live range, we consider a calculation point before the reading of y, another one between the reading of y and the writing of x, and another one after the writing in x. As z does not belong to the same live range, its use is ignored in the caculation of the item related to this live range. For each reference r, the set of RCR elements that reach its beginning is given by all the elements that reach the ending of some reference p that can precede r in the program control flow, or in[r] = $\bigcup_{p\ pred\ r}$ out[p]. We define that the first program reference is empty and its consistency state is X, in[r_o] = $\{(\epsilon$,X$)\}$. The fact that, in all the points that preced a reference, the set of RCR itens contains an unique element of the same live range is very important. It is the result of the insertion of ϕ function, and it is analogous to the fact that, in the SSA form, each use of a variable is reached by an unique definition of this variable.

According to Ottoni and Araújo [12], there are three cases to obtain the elements that reach the ending of a reference r, depending on the type of the reference: use, definition or ϕ function. Suppose LR is the given live range and *s* some consistency state:

- *Case 1*: r: x:= ... | x $\in$ LR
  out[r] = $\{$(x, I)$\}$;
- *Case 2*: r: ... := x | x $\in$ LR

$$out[r] = \begin{cases} \{(x,s)\}, & \text{if } in[r] = \{(x,s)\}, \forall s \\ \{(x,C)\}, & \text{if } in[r] = \{(y,s)\}, y \in (V \cup \{\epsilon\}) - \{x\}, \forall s \\ \{(x,X)\}, & \text{if } in[r] = \{(\omega_i,\sigma_i)\}, \omega_i \in V_\phi \end{cases}$$

- *Case 3*: r: ϕ_i
  out[r] = $\{(\omega_i, \sigma_i)\}$

In Case 1, there is a definition of the variable x. After this reference, the register contains the variable x with a value that is inconsistent with the value of variable x int the memory, because a new value was just defined.

Case 2 is subdivided in three cases, depending on the item that precedes the reference. In the first subcase in[r] is constituted by a state in which x is allocated to the register, or x was kept in the register before the given reference. An use of x will keep x in the register with the same consistency state as before. The second subcase happens when in[r] has another variable y, which belongs to the same live range as x, allocated to the register or the register is empty. In this case, no matter the consistency of y, a load x is necessary before the reference r, this way, the register now will contains x with a value that is consistency with the value in the memory, because it has just been loaded from memory. The third subcase happens when the reference r is reached by a ϕ function. In this case, we can only garantee that the register contains the value of x, in a not determined consistency state, which will depend on the chosen solution of ϕ funcion. This solution could be, for example, the same variable x.

Case 3 deals with ϕ functions, which can be considered a special type of reference that leaves the register in a state given by the solution chosen for the function.

Expressions for the sets *in* e *out* can be used to calculate, iteratively, these sets for all the program points, the same way as the expressions for others data flow analysis are solved, as for example, the liveness analysis.

Taking for granted the property that, after the insertion of ϕ functions, all the reference to some variable that belongs to the live range is reached by an unique register state, the minimum set of necessary instructions to adjust the content of the register can be calculated for each reference that does not depend on the solutuion of ϕ functions. This is the next pass of the algorithm. Suppose $live_{in}$[r] is the set of live variables in the point before the reference r, the following cases, defined by Ottoni and Araújo [12] identify the instructions that does not depend on the solution of ϕ functions:

1. *Case*: r: x:= ... | x $\in$ LR, in[r] = $\{$(v, e)$\}$
   (a) Case (v $\in$ V - $\{$x$\}$) and (e = I) and (v $\in$ live$_{in}$[r]:
   the necessary instruction before r is store v. In this case, the register contains a variable v different of x, that is inconsistent and live in this point in the program, therefore, its value must be stored. As the reference to x is a definition, its value does not have to be loaded from memory.
   (b) Otherwise, no instruction is necessary, or its necessity depends on the solution of some ϕ function.
2. *Case*: r: ... := x | x $\in$ LR, in[r] = $\{$(v, e)$\}$
   (a) Case v = ϵ:
   the necessary instruction before r is a load x. The register is empty, contains no variable, and the reference to x is a use, so it is necessary to load the value of x from memory.
   (b) Case (v $\in$ V - $\{$x$\}$) and ((e $\in$ $\{$C,X$\}$) or ((e=I) and (v $\notin$ live$_{in}$[r]))):
   the necessary instruction before r is load x. The register contains a variable v different from x, that is consistent or its consistency state is undefined, or it is inconsistent but is not live in this point in the program. This way, it is not necessary to store the value of v. The reference to x is a use, then its value must be loaded from memory.

(c) Caso (v $\in$ V - $\{$x$\}$) and (e=I) and (v $\in$ live$_{in}$[r]):
the necessary instructions are store v; load x. The register contains a variable v different from x, that is inconsistent and live in this point of the program, therefore the value of v must be stored. The reference to x is an use, so its value must be loaded from memory.

(d) Otherwise, no instruction is necessary, or its necessity dependes on the solution of some ϕ function.

After the insertion of load and store instructions that do not depend on ϕ functions, the instructions which depend on the solutions of ϕ functions must be inserted. This is the final pass of the iteration of the algorithm. The ϕ functions must be solved, the variable attributed to the register and its consistency state must be decided in each point in the program where a ϕ function was inserted.

Given a function $\phi_i = (\omega_i, \sigma_i)$, to calculate the cost of a solution (ω_i, σ_i) = (w,f), we must analyse all the references in the program that:

A1. are reached by ϕ_i, or have (ω_i, σ_i) in its in set; or

A2. are reached by the register with a variable in an undefined state due to the ϕ_i solution; or

A3. reach ϕ_i.

To solve the A1 and A2 Itens, the solution of ϕ_i function must be propagated, the same way it is done in the RCR data flow analysis, and the case analysis presented before must be used to emit instructions that does not depend on the ϕ functions, since the register states in the beginning of hte references will be well determined.

The references in Item A3 are the ones contained in in[ϕ_i]. Then, to evaluate the cost of ϕ_i related to each one of this references r, according to [12], the following case analysis must be done, where the solution of ϕ will be compared to each (v, e) that belongs to in[ϕ_i]:

B1. (w = ϵ) and (v $\in$ V) and (e=I) and (v $\in$ live$_{in}$[ϕ_i]):
the necessary instruction after r is store v. The solutuion of ϕ_i is to have no variable allocated to the register. A variable v that is live and inconsistent and is allocated to the register, must be stored in memory.

B2. (w $\neq \epsilon$) and (v = w) and (f=C) and (e=I):
the necessary instruction after r is store v. The register contains the necessary variable, but it is inconsistent with the memory. The solution demands that the variable be consistent with the memory, so a store instruction is necessary.

B3. (w $\neq \epsilon$) and ((v=ϵ) or ((v $\in$ V - $\{$w$\}$) and ((e=C) or (v $\notin$ live$_{in}$[ϕ_i]))):
the necessary instruction after r is load w. The register is empty, or contains a variable v different of the solution w, but v is consistent or it is not live in this point in the program. Therefore, it is not necessary to store v in the memory, but w must be loaded from memory. Even if f=I, the result of this instruction will be v consistent with the memory, with no additional cost.

B4. (w $\neq \epsilon$) e (v $\in$ V -$\{$w$\}$) e (e=I) e (v $\in$ live$_{in}$[ϕ_i]):
the necessary instructions after r are store v; load w. The register contains a variable v different of the solution w, and v is inconsistent and live in this point in the program. Then, it is necessary to store v in memory, and load w from memory. Even if f=I, the result of these instructions is v consistent with memory, with no additional cost.

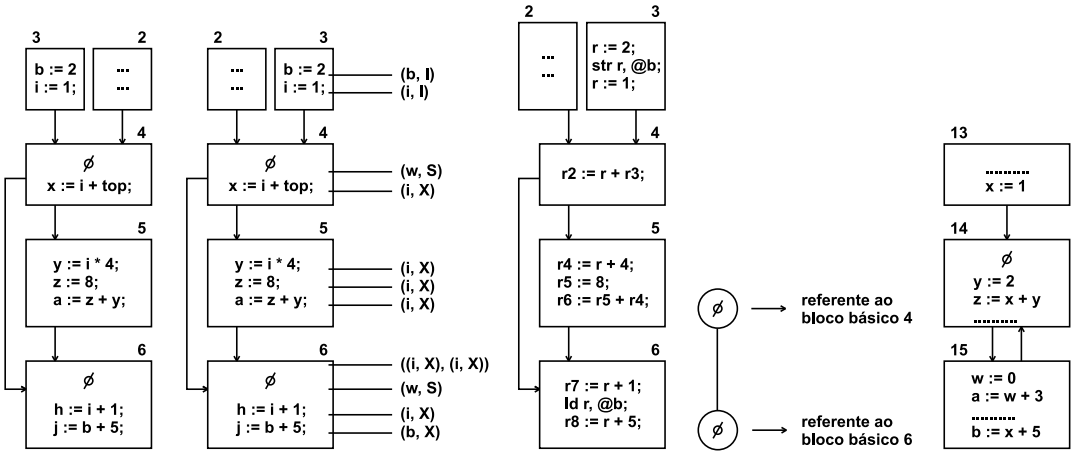

Figura 1: Example of live range growth

B5. Otherwise, no instruction is necessary.

A situation in which the value of a ϕ_i function depends on another ϕ_j function might happens. To solve a ϕ function, in this case, a gluttony strategy that orders the ϕ function using some criterion and then solves then using the ordering can be used. For each ϕ function, we can try all of their possible solutions and choose the one that results in the smallest cost. To calculate the costs of a given ϕ_i function, the solutions of others already solved ϕ_j functions which ϕ_i depends can be used. The others ϕ functions not solved yet, and which ϕ_i depends, must be ignored.

An alternative is to use a dependence ϕ graph (DG_ϕ) to calculate the dependences between the ϕ functions. The DG_ϕ graph is an indirected graph, in which there is one node associate to each ϕ function, and there is an edge (w_i, w_j), between two nodes w_i and w_j, if and only if, w_i is a ϕ function of w_j or otherwise. If the graph is acyclic, the LRO [12] algorithm can be used to determine, in an efficient way the solution to all the ϕ functions.

The complexity of solving the ϕ functions efficiently makes this a NP-hard problem. Then, there is no general, efficient method to solve this functions.

3.1. Example

To ilustrate Ottoni and Araújo method, Figure 1(a) presents a part of a code in the form of a control flow graph. In the iteration explained in the following, we want to join the live ranges *b* and *i*. After the constrution of the live ranges and the insertion of ϕ functions, the first pass of the algorithm is the Register Consistency Reachability. After the RCR analysis, we have the itens presented in Figure 1(b).

The next phase of the algorithm is the emission of instructions that do not depend on ϕ functions. In the basic block 3, the first instruction defines *b*. The second one defines *i*. As *b* and *i* will be joined, they will ocupy the same register. By RCR analysis, before the definition of *i* in the second instruction, the register allocated to the union of *b* and *i* contains the variable *b*, inconsistent with the memory and live in this point of the program, because it is used in basic block 6. Then, we fall in Case 1.a of the case

analysis presented for the emission of instructions that do not depend on ϕ functions, which demands a store *b* instruction before the definition instruction of *i*. In the basic block 6, in the last instruction, variable *b* is used. However, by RCR analysis, the register contains the variable *i*, but *i* is no longer live at this point in the program, so we fall in Case 2.b of the case analysis of emission of instructions that do not depend on ϕ functions, which demands a load *b* before the instruction that uses it.

The next phase of the algorithm is the construction of the ϕ dependences graph for the live ranges to be joined. The form of the graph is shown in Figure 1(d). There is an edge that connects phi_1, presented in basic block 4, with ϕ_2, presented in basic block 6. This edge exists because ϕ_2 depends on the solution of ϕ_1, since the RCR item that reachs ϕ_2, *i* has a consistency state X, that depends on the chosen solution for ϕ_1.

For the solution of ϕ function presented in basic block 4, we have to analyze the references of the program that are reached by ϕ, the ones that are reached by the register with a variable in an undefined state because of the solution of ϕ and the ones that reach ϕ. In the first case, the reference reached by ϕ is *i* in the instruction *x := i + top*. In the second case, all references to *i* in the basic blocks 4, 5 and 6 are reached by the register with a variable, in this case *i*, in an undefined state (X) because of the solution of ϕ. At last, in the third case, the item *(i, I)* reaches ϕ. If the solution of ϕ is to have no variable using it, as the item that reaches ϕ is *(i, I)* and *i* is live at this point in the program, we have a cost of one store *i*. If the solution of ϕ is to have *i* allocated to the register, in a state C, consistent with the memory, we also have a cost of one store *i*. If the solution is *b* in an inconsistent state I, as *b* and *i* are live at this point in the program, we have a cost of one store *i* and one load *b*. Finally, if the solution is *i* in an inconsistent state I, we have a zero cost. Then, this last solution is the chosen one, because the cost for it is 0.

To solve the ϕ function of basic block 6, we have that this ϕ function depends on the ϕ_1 function in basic block 4, which is already solved as *(i, I)*. So, this is the reference that reaches ϕ. The reference reached by ϕ is an use of *i* in the instruction *h := i + 1*. The ones that are reached by the register with a variable in an undefined state because of ϕ function is an use of *b* in the following instruction. The case analysis to choose the ϕ function is equally to the one done for the solution of the prior ϕ function, then the solution of this ϕ function is *(i, I)*, which has a cost of zero. After allocating the register *r* to the union of *b* and *i*, and emiting the instructions, we have the resulting code shown in Figure 1(c).

4. Proposed Algorithm

The LRO [12] heuristic used to determine in an eficienttly way the solution for all the ϕ functions when the DG_ϕ graph is acyclic, was not explored in this version of the implementation. Despite its effectiveness, it is only used when the solution of a ϕ function depends on another ϕ function and when the DG_ϕ graph is acyclic. These situations do not cover all the possible cases. Then, just the brute force heuristic was implemented.

First of all, we implemented an algorithm that, for the solution of ϕ function, uses the last case analysis for the references that are reached by ϕ, as described in Section 3.

To analyze the references being reached by ϕ, in order to save one visit to the code of the program, the references in the ϕ function successors instructions that have a register with a variable in an undefined state (X) due to the solution of ϕ or have (ω_i, σ_i) in their *in* set are searched.

This implementation of the algorithm did not produce efficient results when the joined live ranges were simultaneous live and their references were intercalated inside a basic

block. To solve this problem, we have proposed to insert interferences between live variables, and live ranges that interfere with each other cannot be joined.

5. Implementation Environment

In order to test the Global Register Allocation Based in Live Range Growth method, we have implemented it in the Machine SUIF [9] (MachSUIF), a flexible and extensible structure to the construction of compilers back-ends. In MachSUIF, a minimum back-end is composed of: generation of low level code, generation of machine specific code, register allocation, ending of generation of machine specific code, generation of assembly code. Several optimizations pass can be inserted during this sequence, as for example, scheduling, constants folding, generation of code in the SSA form. MachSUIF allows the development of new optimization passes and the introduction of new architectures in an easy way.

The Register Allocation method used by MachSUIF is the method of George and Appel [7]. This method intercalate the simplification phase with the Briggs coalesce heuristic, making the allocation algorithm more aggressive. This method is called Iterated Register Coalescing. This compiler pass of register allocation implemented in MachSUIF was substituted by the method implemented in this paper, based in Live Range Growth. The resulting code of the two methods were compared and the result is presented in Section 6.

The target processor for the experiments was Alpha. It has a load/store RISC architecture, with 64 machine registers, which 54 are allocable. Inside these 54, 23 are genereal proposal registers and 31 are floating-point. The used benchmark were: bubsort, resolution of fibonacci series, bubsort, eratostenes crive, matrix multiplication, depth-first seek, among otheres. These programs are very used applications and garantee the covery of several cases of control flow graphs, number of candidates variables to allocation and dependences among ϕ functions. The comparison criterion between both methods of Register Allocation, one based in Live Range Growth and the other based on George and Appel technic, was the number of intructions that access the memory (load and store) inserted in the resulting code of these allocations.

6. Results Evaluation

In the implementation, the choice of which pairs of live ranges will have their union tested is very important. For example, it is interesting to choose one live range of each path in the control flow graph that reachs the basic block that contains the ϕ function, so, this way, we try to garantee the possibility of the joined live ranges will be not live at the same point in the program, which produces a smaller possibility of insertion of instructions that access the memory. The Figure 1(e) illustrate this fact, in this figure, the basic blocks 13 and 15 are antecessors to basic block 14, which contains the ϕ function. If a live range that has a reference in 13 and another one that has a reference in 15 are chosen to have their union cost calculated, we are choosing a live range of each path in the control flow graph that reachs the basic block that contains the ϕ function. In our implementation, that was the chosen approach, observing that the variables contained in the live ranges must be of the same type and they cannot interfere with each other. A variable interfere with another one if both of them are operands in the same instruction. When this happens, they cannot share the same register, because, this way, they would be using the same register at the same time. If this choise of live ranges to have their union tested cannot be possible,

all the live ranges of the same type and that does not interfere with each other are tested, independently of the path in the control flow graph.

The first important observed result was the fact that the algotithm of Ottoni and Araújo [12], despite of Chaitin allocation, does not implement the transformation known as Register Coalesce. This transformation is a variation of the copy propagation, and eliminates copies from one register to another one. In this transformation, register copies instructions are searched in the intermediary code in such a way that, the copies instructions have the form $s_j \leftarrow s_i$, and s_i and s_j does not interfere with each other, and s_j and s_i are not stored in memory between the copy attribution and the end of the rotine. After these instructions are found, the Register Coalescing look for the instruction that wrote in s_i and changes it to put its result in s_j in the place of s_i, and removes the copy instruction [11]. This way, s_i does not exist anymore in the program code.

We have observed that the number of live ranges were large in the implemented method, besides that, the code generated by this method contained many unnecessary register copies instructions. Therefore, we have decided to implement the Register Coalesce transformation in the Allocation Based on Live Range Groth Method. The transformation was implemented in the beggining of the algorithm, after the webs and the live ranges were calculated (they were necessay to check the existency of interferences between the possible candidates to the coalesce). After the transformation, the webs and the live ranges are recalculated. The result was very good, reducing the number of live range, in average, in 40%. We could observe, this way, that this transformation is very important to this method, not just by the elimination of unnecessary copies instructions, but also by the reducing of the number of live ranges.

For example, in the quicksort algorithm, before the Coalesce transformation was implemented, the method of Allocation Based in the Live Range Growth identified 109 live ranges, and besides of that, the generated code contained may unnecessary copies instructions, as the total, there were 156 instructions in the generated code. After the implementation of the Register Coalesce, the number of live ranges identified diminished to 64. The generated code contained 112 instructions, so, the Register Coalesce eliminated 44 unnecessary copies instructions.

The second important observation was the fact that, in some situations, the code that the method based in live range growth generated contained load and store instructions, while the one generated by the coloring graph method did not have instructions of these kind. This situations ocorred every time that the union of the smallest cost was a union of two live ranges in which:

1. at least one of them was presented in both paths that were considered;
2. the references to the live ranges were intercalated in the basic block that contained them;
3. one of them remained live at the end of the basic block.

For example, in the code shown by Figure 1(e), the iteration of the algorithm will join the live ranges that contain *x* and *w*. To solve the ϕ function presented in the basic block 14, the references that reachs it and the ones that are reached by it are considered. The references (to the live ranges in question) that reach ϕ are a definition of *x* in the basic block 13 and an use of *x* in the basic block 15, antecessors blocks of basic block 14. The reference to the live ranges in question that is reached by ϕ is an use of *x* in the basic block 14. So, analyzing the cost of the possible solutions of ϕ, we find that the one with the smaller cost is the pair *(x, I)*, the variable *x* in a state consistency of I, generated by its

definition, which has a cost 0. However, in the next iteration of the algorithm, after x and w are joined, in the phase of emission of instructions that do not depend on the solution of a ϕ function, we have that, in the end of basic block 14, the register allocated to the joined live range, with x and w, contains the x variable in a state inconsistency with the memory. The next reference to one of the variables presented in the live range is a definition of w in the beggining of basic block 15. As the register contains x in an inconsistent state, and x is live, we need to insert an instruction of store x before the instruction that defines w. After the definition of w, there is an use of it, and after this use, there is an use of x. The register now, contains w in an inconsistent state with the memory. As w is no longer live, it is necessary to insert an instruction of load x before its use, because we have to load the value of x from the memory to the register, so it can be used. So, the union of x and w had a cost of 2 instructions of access to memory, that were not captured in the resolution of the ϕ function. This cost was just captured in the next iteration, in the emission of instructions that do not depend on the ϕ functions phase.

The solution of ϕ function related to this union in the first implementation of the algorithm, did not retreat the cost of the inserted store and load instructions because, for the solution of a ϕ function, the references analyzed and considered are the references that reach and that are reached by the ϕ function, and this represents the last references in the basic block that are predecessor to the one that contains the ϕ function or the first instruction in the proper basic block or their sucessors that use or define the live range in question. The fact is, in this case, that the predecessor basic block, the sucessor one or the basic block itself contained both live ranges intercalated, but only the last reference to one of these live ranges is considered for the solution of ϕ function, in the case of the predecessor block, and the first reference to one of these live ranges is considered in the case of the sucessor block or the block itself. This way, the cost of the change of variables in the live range to ocupy the register is not analyzed. Therefore, this situation happened when the variables belonging to the joined live range were lives at the same time.

To solve this problem, interferences between live variables were inserted. Live ranges that interfere among each other cannot be joined. For example, in the Figure 1(e), as x *and* w are lives at the same time, an interference between them is inserted, so these live ranges cannot be joined. This solution eliminated this problem, and the code generated by the method became efficient also in this situation. The problem with this solution is that, this way, the method becomes closer to the graph coloring method, which works with interferences between live variables.

To solve this problem, we have also implemented the extension of the analyzis of the cost of the resolution of the ϕ functions to all the basic blocks in the program, according as the resolution of itens A1 and A2, and not just to the references that reach or are reached by ϕ. So, the phase of the emission of instructions that do not depend on the resolution of a ϕ function is performed, considering that the live ranges that are having their union tested is already joined. If some instruction is inserted, the number of these inserted instructions is added to the cost of the union of the live ranges. This way, the cost of the intercalation of the variables will be computed, the cost of the union will become larger, and possibly this union will not be the chosen one. In the Figure 1(e), the cost of the union of x *and* w is 2, because the phase of the emission of instructions that do not depend on ϕ functions if x *and* w are joined is considered, so, the costo of the insertion of a store x *and a load* x in the basic block 15 is considered. Probably, there will be another union whose cost

is smaller, and this one will be the chosen one. This solution surely increases the cost of the compilation, but it solves the problem explained above without becoming closer to the solution adopted by the graph coloring. This solution was implemented and tested in the Allocation Method Based in Live Range Growth, and the generated code became more efficient.

The most important observation was the fact that the Register Allocation Algorithm Based in Live Range Growth achieved more efficients results than the Graph Coloring Algorithm in some situations, for example, in the presence of a lots of nested loops and when global variables remained lives and very referenced in all the program code, also being very referenced inside the loops. In the case of the nested loops, their bounds were spilled to memory, as long as many registers were not allocated inside the loop. In the case of the global variables lives and very referenced inside loops, besides the indexes and the bounds of the loop, the global variables which were most referenced were also spilled to memory. In these cases, the live ranges that represent the global variables have a spill cost larger due to the many interferences that exist between them and others variables in the interference graphs and due to the fact that they are referenced inside the loops. The allocator, then, spills the live ranges with a smaller cost, which are the bounds of the loops. In some situations, this is not enough, and some global variables are also spilled.

	Load and Store				Live Ranges	
Algoritmo	CG	1CDA	2CDA	3CDA	SC	CC
test	0	0	0	0	21	12
fibonacci	0	0	0	0	19	8
bubsort	0	4	0	0	109	81
quicksort	2	0	0	0	109	64
bfirst	4	2	0	0	140	66
integer	0	2	0	0	142	102
knight	0	0	0	0	171	99
float1	0	0	0	0	41	20
matrixmul	7	0	0	0	149	83
matrixmul2	21	0	0	0	199	127
point2	0	0	0	0	132	70
rsieve	0	0	0	0	55	27
whetsto	70	4	0	0	102	64
whetsto3	139	-	6	6	198	134
gauss-seidel	3	0	0	0	167	97

Table 1: load and store instructions and number of live ranges before and after the Register Coalesce

Table 1 shows, for each one of the tested algorithms of the benchmark, the number of load and store instructions obtained: I) by the method of graph coloring (CG), II) by the first implementation of the method based in the live range growth, without considering all the basic blocks in the calculation of the cost of ϕ (1CDA), III) by the implementation of the method based in the live range growth with interferences between live variables (2CDA), IV) by the implementation of the method that considers all the basic blocks in the calculation of the ϕ cost [12] (3CDA). The last two columns show the number of identified live ranges in the implementations without (SC) and with (CC) the transformation of Register Coalesce.

We did not do comparisons between the two methods (graph coloring and based in live range growth) in relation to the run time, due to the fact that the graph coloring method

is optimized, while the method base in live range growth is not optimized, its studies has just begun.

7. Conclusions

We evaluated, with this work, the effectiveness of the Global Register Allocation Method Based in Live Range Growth. Throughout their implementation and tests compared to the Global Register Allocation Based in Graph Coloring, we have discovered deficiencies and proposed and implemented solutions to them. The generated code has become more efficient with the proposed improvements, but the method is more expensive than the graph coloring method, because at each iteration it visits at least three times the code of the program: to do the RCR Analysis, to the emission of the instructions that do not depend on the ϕ functions, to the solution of the ϕ functions for each pair of tested live ranges. And also it visits the code of the program to constructed the dependence graph between ϕ functions (DG_ϕ).

The most important result obtained by this work was the fact the the Algorithm Based in Live Range Growth is more efficient than the Algorithm Based in Graph Coloring in situations which there are many nested loops and many global variables which remained lives during all the program and are referenced inside loops, nested or not. While the allocator based in graph coloring generates a lot of spills, the allocator based in live range growth does not generate instructions that access the memory in these cases. This happens because of the fact that the spill decisions of the graph coloring are estimated, we do not know for sure what will happen after these spills, even because there are no flow control analysis in this method. Therefore, the method based in live range growth, besides doing flow control analysis, knows exactly what will happen if two live ranges is joined, so their spill decisions are more realistic, they are based in real measurements.

We also concluded that the Method Based in Live Range Growth is more efficient than the Method of George and Appel in the cases explained above. There is, however, a relation effectiveness versus cost between these two methods.

Referências

[1] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, 1992.

[2] David Callahan and Brian Koblenz. Register allocation by hierarchical tiling. *Proceedings of the ACM SIGPLAN91*, 1991.

[3] G. J. Chaitin. Register allocation and spilling via graph coloring. *IBM Research*, 1982.

[4] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. *Proceedings of the ACM SIGPLAN84*, 1984.

[5] Marcelo Silva Cintra. Alocação global de registradores de endereçamento usando cobertura do grafo de indexação e uma variação da forma ssa. Master's thesis, Universidade Estadual de Campinas, 2000.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Denneth Zadeck. An efficient method of computing static single assignment form. *Proceedings of the ACM SIGPLAN89*, 1989.

[7] Lal George and Andrew W. Appel. Iterated register coalescing. *Proceedings of the ACM TOPLAS96*, 1996.

[8] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. *PLDI89*, 1989.

[9] Glenn Holloway and Michael D. Smith. Machine suif, 2000. http://www.eecs.harvard.edu/hube/research/machsuif.html.

[10] Kathleen Knobe and F. Kenneth Zadeck. Register allocation using control trees. Technical report, Dept. of Comp. Sci., Brown Univ., Providente, RI, 1992.

[11] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[12] Guilherme Lima Ottoni. Alocação global de registradores de endereçamento para referências a vetores em dsps. Master's thesis, Universidade Estadual de Campinas, 2002.

[13] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. *Proceedings of the ACM SIGPLAN98*, 1998.