



HAL
open science

Optimizing the translation out-of-SSA with renaming constraints

Fabrice Rastello, F. de Ferrière, Christophe Guillon

► **To cite this version:**

Fabrice Rastello, F. de Ferrière, Christophe Guillon. Optimizing the translation out-of-SSA with renaming constraints. [Research Report] LIP RR-2005-34, Laboratoire de l'informatique du parallélisme. 2005, 2+26p. hal-02102190

HAL Id: hal-02102190

<https://hal-lara.archives-ouvertes.fr/hal-02102190>

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

***Optimizing the translation out-of-SSA with
renaming constraints***

F. Rastello
F. de Ferrière
C. Guillon

August 2005

Research Report N° 2005-34

École Normale Supérieure de Lyon

46 Allée d'Italie, 69364 Lyon Cedex 07, France

Téléphone : +33(0)4.72.72.80.37

Télécopieur : +33(0)4.72.72.80.80

Adresse électronique : lip@ens-lyon.fr



INRIA



Optimizing the translation out-of-SSA with renaming constraints

F. Rastello
F. de Ferrière
C. Guillon

August 2005

Abstract

Static Single Assignment form is an intermediate representation that uses ϕ instructions to merge values at each confluent point of the control flow graph. ϕ instructions are not machine instructions and must be renamed back to move instructions when translating out of SSA form. Without a coalescing algorithm, the out of SSA translation generates many move instructions. Leung and George [8] use a SSA form for programs represented as native machine instructions, including the use of machine dedicated registers. For this purpose, they handle renaming constraints thanks to a pinning mechanism. Pinning ϕ arguments and their corresponding definition to a common resource is also a very attractive technique for coalescing variables. In this paper, extending this idea, we propose a method to reduce the ϕ -related copies during the out of SSA translation, thanks to a pinning-based coalescing algorithm that is aware of renaming constraints. This report provides also a discussion about the formulation of this problem, its complexity and its motivations. We implemented our algorithm in the STMicroelectronics Linear Assembly Optimizer [5]. Our experiments show interesting results when comparing to the existing approaches of Leung and George [8], Sreedhar et al. [11], and Appel and George for register coalescing [7].

Keywords: Static Single Assignment, Coalescing, NP-complete, K-COLORABILITY, Machine code level, register allocation

Résumé

La forme SSA est une représentation intermédiaire de compilateur qui utilise des fonctions virtuelles ϕ pour fusionner les valeurs à chaque point de confluence du graphe de contrôle. Les fonctions ϕ n'existant pas physiquement, elles doivent être remplacées par des instructions `move` lors de la translation en code machine. Sans coalesceur, la translation hors-SSA génère beaucoup de `move`.

Dans cet article, nous proposons une extension de l'algorithme de Leung et George [8] qui effectue la minimisation de ces instructions de copie. Leung et al. proposent un algorithme de translation d'une forme SSA pour du code assembleur, mais non optimisé pour le remplacement des instructions ϕ . Par contre, ils utilisent la notion d'épinglage pour représenter les contraintes de renommage.

Notre idée est d'utiliser cette notion d'épinglage afin de contraindre le renommage des arguments des ϕ pour faire du coalescing. C'est une formulation du problème de coalescing non équivalente au problème initial toujours considéré comme ouvert dans la littérature [8, 10]. Nous prouvons néanmoins la NP-complétude de notre formulation, une conséquence de la preuve étant la NP-complétude du problème initial en la taille de la plus grande fonction ϕ .

Enfin, nous avons implémenté notre algorithme dans le LAO [5], optimiseur d'assembleur linéaire. La comparaison avec différentes approches possibles fournit de nombreux résultats intéressants. Nous avons aussi essayé, à l'aide d'exemples faits à la main, d'expliquer les avantages et limitations des différentes approches.

Mots-clés: forme SSA, fusion de variables, NP-complétude, K-COLORABLE, code assembleur, allocation de registres

1 Introduction

Static Single Assignment The Static Single Assignment (SSA) form is an intermediate representation widely used in modern compilers. SSA comes in many flavors, the one we use is the *pruned* SSA form [4]. In SSA form, each variable name, or virtual register, corresponds to a scalar value and each variable is defined only once in the program text. Because of this single assignment property, the SSA form contains ϕ instructions that are introduced to merge variables that come from different incoming edges at a confluent point of the control flow graph. These ϕ instructions have no direct corresponding hardware instructions, thus a translation out of SSA must be performed. This transformation replaces ϕ instructions with move instructions and some of the variables with dedicated ones when necessary. This replacement must be performed carefully whenever transformations such as copy propagation have been done while in SSA form. Moreover, a naive approach for the out of SSA translation generates a large number of move instructions. This paper¹ addresses the problem of minimizing the number of generated copies during this translation phase.

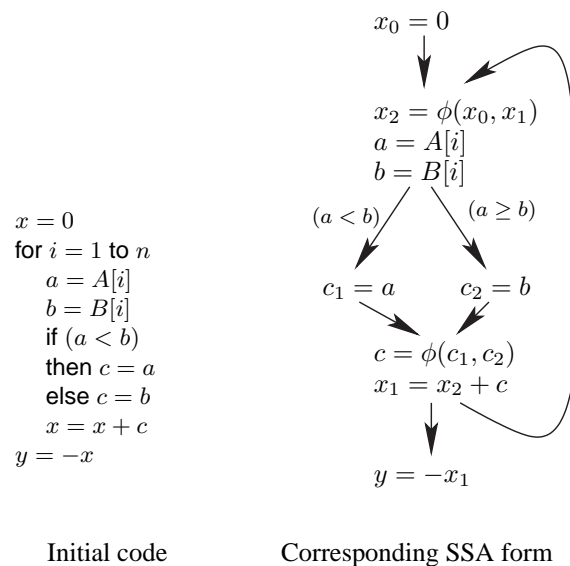


Figure 1: Example of code in non-SSA form and its corresponding SSA form without the loop counter represented

Previous Work Cytron et al. [4] proposed a simple algorithm that first replaces a ϕ instruction by copies into the predecessor blocks, then relies on Chaitin’s coalescing algorithm [3] to reduce the number of copies. Briggs et al. [1] found two correctness problems in this algorithm, namely the swap problem and the lost copy problem, and proposed solutions to these. Sreedhar et al. [11] proposed an algorithm that avoids the need for Chaitin’s coalescing algorithm and that can eliminate more move instructions than the previous algorithms. Leung and George [8] proposed an out-of-SSA algorithm for an SSA representation at the machine code level. Machine code level representations add **renaming constraints** due to ABI (Application Binary Interface) rules on calls, special purpose ABI defined registers, and restrictions imposed on register operands.

¹Many thanks to Alain Darte, Stephen Clarke, Daniel Grund and the reviewers of CGO for very helpful comments on the presentation of this report.

Context of the study Our study of out-of-SSA algorithms was needed for the development of the STMicroelectronics Linear Assembly Optimizer (LAO) tool. LAO converts a program written in the Linear Assembly Input (LAI) language into the final assembly language that is suitable for assembly, linking, and execution. The LAI language is a superset of the target assembly language. It allows symbolic register names to be freely used. It includes a number of transformations such as induction variable optimization, redundancy elimination, and optimizations based on range propagation, in an SSA intermediate representation. It includes scheduling techniques based on software pipelining and superblock scheduling, and uses a *repeated coalescing* [5] register allocator, which is an improvement over the *iterated register coalescing* from George and Appel [7]. The LAO tool targets the ST120 processor, a DSP processor with full predication, 16-bit packed arithmetic instructions, multiply-accumulate instructions and a few 2-operands instructions such as addressing mode with auto-modification of base pointer.

Because of these particular features, an out-of-SSA algorithm aware of renaming constraints was needed. In fact, delaying renaming constraints after the out-of-SSA phase would result in additional move instructions (see Section 5), along with possible infeasibilities and complications. We modified an out-of-SSA algorithm from Leung and George to handle renaming constraints and reduce the number of move instructions due to the replacement of ϕ instructions.

Layout of this paper The paper is organized as follows. Section 2 states our problem and gives a brief description of Leung and George’s algorithm. In Section 3, we present our solution to the problem of register coalescing during the out-of-SSA phase. Section 4 discusses, through several examples, how our algorithm compares to others. In Section 5, we present results that show the effectiveness of our solution on a set of benchmarks, and we finally conclude. This paper contains also two appendices A and B devoted respectively to the refinement of Leung’s algorithm and to the NP-completeness proof of the pinning based coalescing problem.

2 Problem statement and Leung and George’s algorithm

Our goal is to handle renaming constraints and coalescing opportunities during the out of SSA translation. For that, we distinguish **dedicated registers** (such as *R0* or *SP*, the stack pointer) from general-purpose registers that we assume in an unlimited number (we call them **virtual registers** or **variables**). We use a pinning mechanism, in much the same way as in Leung and George’s algorithm [8], so as to enforce the use of these dedicated registers and to favor coalescing. Then, constraints on the number of general-purpose registers are handled later, in the register allocation phase.

2.1 Pinning mechanism

An **operand** is the *textual use* of a variable, either as a write (definition of the variable) or as a read (use in an instruction). A **resource** is either a physical register or a variable. **Resource pinning** or simply **pinning** is a pre-coloring of operands to resources. We call **variable pinning** the pinning of the (unique) definition of a variable. Due to the semantics of ϕ instructions, all arguments (i.e. use operands) of a ϕ instruction are pinned to the same resource as the variable defined (i.e. def operand) by the ϕ .

On the ST120 processor, as in Leung and George’s algorithm, we have to handle Instruction Set Architecture (ISA) register renaming constraints and Application Binary Interface (ABI) function parameter passing rules. Figure 2, expressed in SSA pseudo assembly code, gives an example of such constraints. In this example and in the rest of this paper, the notation $X \uparrow^R$ is used to mark that the

operand X is pinned to the resource R . When the use of a variable is pinned to a different resource than its definition, a move instruction has to be inserted between the resource of the definition and the resource of the use. Pinning the variable to the same resource as its uses has the effect of coalescing these resources (i.e., it deletes the move).

Original code:	SSA pinned code:	Comments:
.input C, P	$S_0 : \text{.input } C \uparrow^{R0}, P \uparrow^{P0}$	Inputs C and P must be in $R0$ and $P0$ at the entry.
load $A, @P++$	$S_1 : \begin{cases} \text{load } A, @P \\ \text{autoadd } Q \uparrow^Q, P \uparrow^Q, 1 \end{cases}$	The second def. of P is renamed as Q in SSA, but P and Q must use the same resource for autoadd, e.g., Q .
load $B, @P++$	$S_2 : \text{load } B, @Q$	
call $D = f(A, B)$	$S_3 : \text{..f } D \uparrow^{R0}, A \uparrow^{R0}, B \uparrow^{R1}$	Parameters must be in $R0$ and $R1$. Result must be in $R0$.
$E = C + D$	$S_4 : \text{add } E, C, D$	
$K = 0x00A12BFA$	$S_5 : \text{make } L, 0x00A1$	
$F = E - K$	$S_6 : \text{more } K \uparrow^K, L \uparrow^K, 0x2BFA$	Operands K & L must use the same resource, e.g., K .
.output F	$S_7 : \text{sub } F, E, K$	
	$S_8 : \text{.output } F \uparrow^{R0}$	Output F must be in $R0$.

Figure 2: Example of code with renaming constraints: function parameter passing rules (statements S_0, S_3 , and S_8) and 2-operand instruction constraints (statements S_1 and S_6).

2.2 Correct pinning

Figure 3 gives an example of renaming constraints that will result in an incorrect code. On the left of Figure 3, the renaming constraint is that all variables renamed from the dedicated register SP (Stack Pointer) must be renamed back to SP , due to ABI constraints. On the right, after replacement of the ϕ instructions, the code is incorrect. Such problem mainly occurs after optimizations on dedicated registers: SSA optimizations such as copy propagation or redundancy elimination must be careful to maintain a semantically correct SSA code when dealing with dedicated-register constraints. More details on correctness problems related to dedicated registers are given in Appendix A.

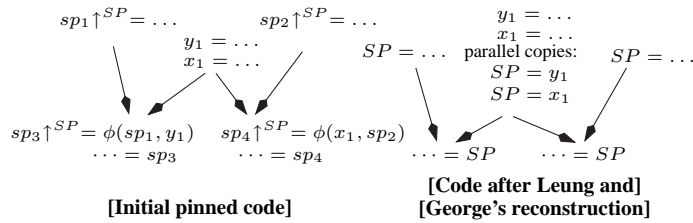


Figure 3: A too constrained pinning can lead to an incorrect code as for the parallel copies here.

Cases of incorrect pinning are given in Figure 5. In this figure, Case 1 and Case 2 are correct if and only if x and y are the same variable. This is because two different values cannot be pinned to a unique resource if both of them must be available at the entry point of an instruction (Case 2) or at the exit point of an instruction (Case 1). A similar case on ϕ instructions is given in Case 3: the set of ϕ instructions at a block entry has a parallel semantics, therefore two different ϕ definitions in the same block cannot be pinned to the same resource. On the other hand, on most architectures, Case 4 is a correct pinning. But, the corresponding scheme on a ϕ instruction (Case 5) is forbidden when $s \neq r$: this is because all ϕ arguments are implicitly pinned to the resource the ϕ result is pinned to. The

motivation for these semantics is given in Appendix A. Finally, Case 6 corresponds to a more subtle incorrect pinning, similar to the problem stressed in Figure 3.

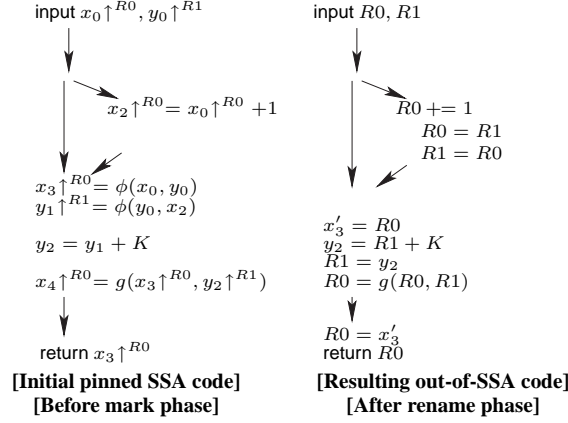


Figure 4: Transformation of already pinned SSA code by Leung and George's algorithm.

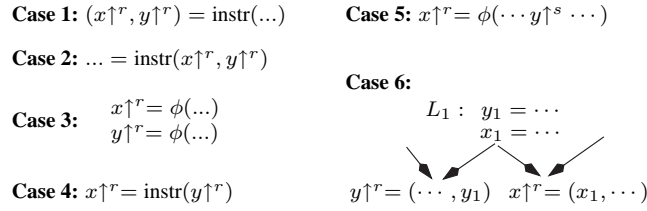


Figure 5: All but Case 4 are incorrect pinning.

2.3 Leung and George's algorithm

Leung and George's algorithm is decomposed into three consecutive phases: the *collect* phase collects information about renaming constraints; the *mark* phase collects information about the conflicts generated by renaming; the *reconstruct* phase performs renaming, inserts copies when necessary and replaces ϕ instructions.

Pinning occurs during the collect phase, and then the out of SSA translation relies on the mark and reconstruct phases. Figure 4 illustrates the transformations performed during those last two phases:

- x_3 is pinned to $R0$ on its definition. But, on the path to its use in the `return`, x_4 is also pinned to $R0$ on the call to g . We say that x_3 is **killed**, and a **repair copy** to a new variable x'_3 is introduced.
- The use of x_3 in the call to g is pinned to $R0$, while x_3 is already available in $R0$ due to a prior pinning on the ϕ instruction. The algorithm is careful not to introduce a redundant move instruction in this case.
- The copies $R0 = R1$; $R1 = R0$ are performed **in parallel** in the algorithm, so as to avoid the so-called swap problem. To sequentialize the code, intermediate variables may be used and the copies may be reordered, resulting in the code $t = R1$; $R1 = R0$; $R0 = t$ in this example.

Now, consider the *non-pinned variable* y_2 of Figure 4 and its use in the definition of x_4 . *The use is pinned to a resource, R_1 , and y_2 could have been coalesced to R_1 without creating any interference.* The main limitation of Leung and George’s algorithm is its inability to do so. The same weakness shows up on ϕ arguments, as illustrated by Figure 6(a): on a ϕ instruction $X = \phi(x_0, \dots, x_n)$, each operand x_i is implicitly pinned to X , while the definition of each x_i may not. **Our pinning-based coalescing is an extension to the pinning mechanism whose goal is to overcome this limitation.**

2.4 The ϕ coalescing problem

As opposed to the pinning due to ABI constraints, which is applied to a textual use of an SSA variable, the pinning related to coalescing is applied only to variable definitions (*variable pinning*). Figure 6 illustrates how this pinning mechanism can play the role of a coalescing phase by preventing the reconstruction phase of Leung and George’s algorithm from inserting move instructions: in Figure 6(b), x_1 and x_2 were pinned to x to eliminate these move instructions; however, this pinning creates an interference, which results in a repair move $x' = x$ along with a move $x = x'$ on the replacement of the ϕ instruction; in Figure 6(c), to avoid the interference, only x_2 was pinned to x , resulting in only one move instruction.

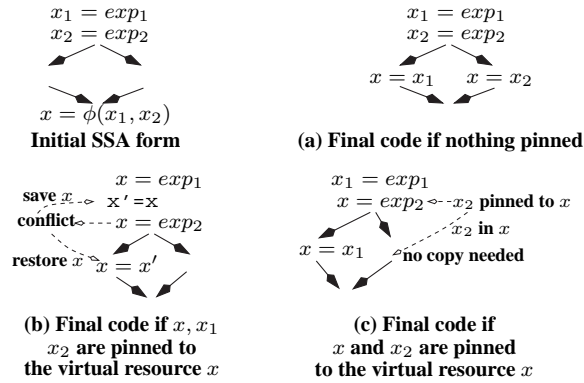


Figure 6: Inability of Leung and George’s algorithm to coalesce $x = x_1$ and $x = x_2$ instructions (a) ; a worst (b) and a better (c) solution using variable pinning of x_1 and x_2 .

Therefore, we will only look for a variable pinning that does not introduce any new interference. In this case, for a ϕ instruction $X = \phi(x_0, \dots, x_n)$, we say that the gain for ϕ is the number of indices i such that the variable x_i is pinned to the same resource as X . Hence, our **ϕ coalescing problem consists of finding a variable pinning, with no new interference (i.e., without changing the number of variables for which a repair move is needed), that maximizes the total gain, taking into account all ϕ instructions in the program.**

Algorithm 1 Main phases of our algorithm.

```

Program_pinning(CFG_Program P)
foreach basic block B in P, in an inner to outer loop traversal
    Initial_G=Create_affinity_graph(B)
    PrePruned_G=Graph_InitialPruning(Initial_G)
    Final_G=BipartiteGraph_pruning(PrePruned_G)
    PrunedGraph_pinning(Final_G)
    
```

3 Our solution

The ϕ coalescing problem we just formulated is NP-complete (see Appendix B for details). Instead of trying to minimize the gain for all ϕ instructions together, our solution relies on a sequence of local optimizations, each one limited to the gain for all ϕ instructions defined at a confluence point of the program. These confluence points are traversed based on an inner to outer loop traversal, so as to optimize in priority the most frequently executed blocks. The skeleton of our approach is given in Algorithm 1.

Let us first describe the general ideas of our solution, before entering the details. For an SSA variable y , we define $\underline{y} = \mathbf{Resource_def}(y)$ as r if the definition of y is pinned to r , or y otherwise. Also, for simplicity, we identify the notion of resource with the set of variables pinned to it. For a given basic block, we create what we call an **affinity graph**. Vertices represent resources; edges represent potential copies between variables that can be coalesced if pinned to the same resource. Edges are weighted to take into account interferences between SSA variables; then the graph is pruned (deleting in priority edges with large weights) until, in each resulting connected component, none of the vertices interfere: they can now be all pinned to the same resource. The rest of this section is devoted to the precise description of our algorithm. A pseudo code is given in Algorithm 2 on page 17. The consecutive steps of this algorithm are applied on the example of Figure 8.

3.1 The initial affinity graph

For a given basic block, the affinity graph is an undirected graph where each vertex represents either a variable or its corresponding resource (if already pinned): two variables that are pinned to the same resource are collapsed into the same vertex. Then, for each ϕ instruction $X = \phi(x_1, \dots, x_n)$ at the entry of the basic block, there is an affinity edge, for each i , $0 \leq i \leq n$, between the vertex that contains X and the vertex that contains x_i .

3.2 Interferences between variables

We define below four classes of interferences that can occur when pinning two operands of a ϕ instruction to the same resource. We differentiate simple interferences from strong interferences: a strong interference generates an incorrect pinning. On the other hand, a simple interference can always be repaired despite the fact that the repair might generate additional copies. The goal is then to minimize the number of simple interferences and to avoid all strong interferences. The reader may find useful to refer, for each class, to Figure 7.

[Class 1] Consider two variables x and y . If there exists a point in the control flow graph where both x and y are alive, then x and y interfere. Moreover, considering the definitions of x and y , one dominates the other (this is a property of the SSA form). If the definition of x dominates the definition of y , we say that the *definition of x is killed by y* . The consequence is that pinning the definitions of x and y to a common resource would result in a repair of x (as in Leung and George’s technique).

[Class 2] Consider a ϕ instruction $y = \phi(\dots, z, \dots)$ in basic block B . Let C be the block where the argument z comes from; textually, the use of z appears in block B (and is implicitly pinned to y), but semantically, it takes place at the end of basic block C (this is where a move instruction, if needed, would be placed). If $x \neq z$ and x is live-out of block C , then x and the use of z interfere and we say that *the definition of x is killed by y* . Note our definition of **liveness**: a ϕ instruction does not occur where it textually appears, but at the end of each predecessor basic block instead. Hence, if not

used by another instruction, z would be treated as dead at the exit of block C and at the entry of block B .

[Class 3] Consider two variables x and y , both defined by ϕ instructions, but not necessarily in the same basic block. Some of their respective arguments (for example x_i and y_j) may interfere in a common predecessor block B . In this case, we say that *the definitions of x and y strongly interfere*: indeed, as explained in Section 2.2, pinning those two definitions together is incorrect.

[Class 4] Consider ϕ instructions $y = \phi(y_1, \dots, y_n)$ and $z = \phi(y_1, \dots, y_n)$, in the same basic block and with the same arguments. Because of Leung and George's repairing implementation, they cannot be considered as identical and we need to consider that they strongly interfere. Notice that a redundancy elimination algorithm should have eliminated this case before. Note that, by definition of Classes 3 and 4, all variables defined by ϕ instructions in the same basic block strongly interfere.

Also, we consider that *variables pinned to two different physical registers strongly interfere*.

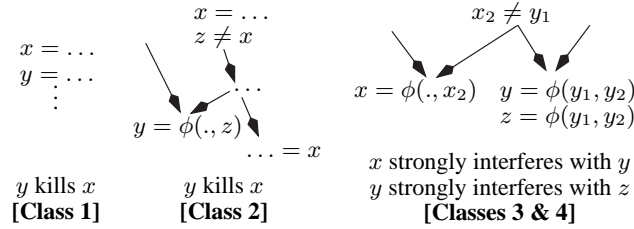


Figure 7: Different kind of interferences between variables.

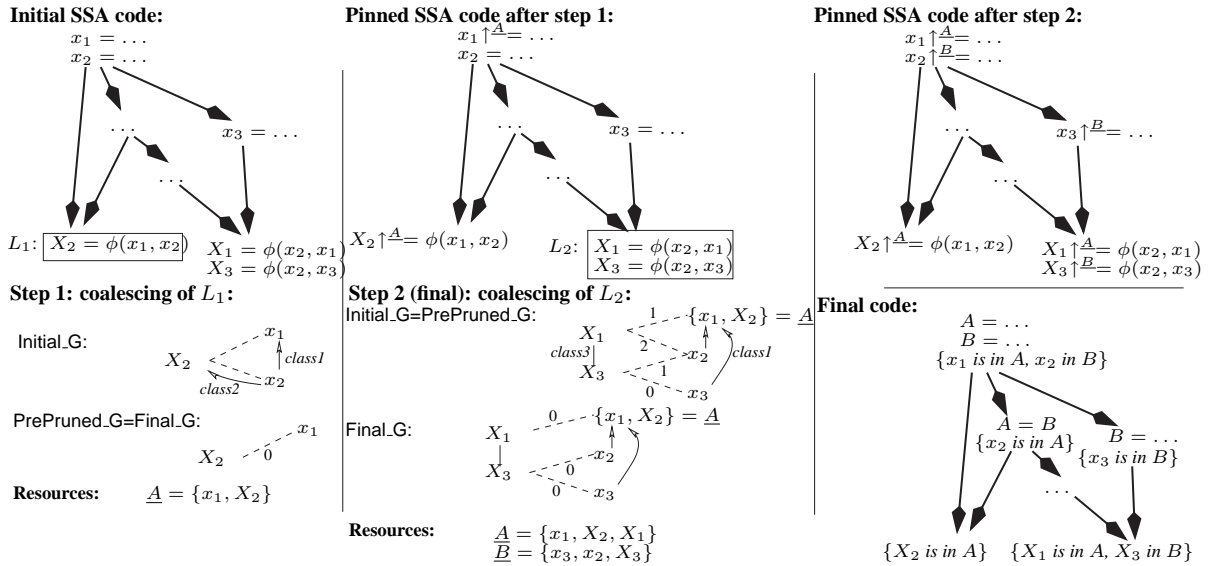


Figure 8: Program pinning on an example. Control-flow graphs are represented for code, with control-flow edges between basic blocks represented with solid black arrows. Affinity graphs are represented for step 1 and 2, with affinity edges represented as dashed gray lines, annotated with a weight, and with interferences edges represented as full gray lines, annotated with the class of the interference.

3.3 Interferences between resources

After the initial pinning (taking into account renaming constraints), a resource cannot contain two variables that strongly interfere. However, simple interferences are possible; they will be solved by Leung and George’s repairing technique. During our iterative pinning process, we keep merging more and more resources, but we make sure not to create any new interference. We say that two resources $\underline{A} = \{x_1, \dots, x_n\}$ and $\underline{B} = \{y_1, \dots, y_m\}$ interfere if pinning all the variables $\{x_1, \dots, x_n\}$ and $\{y_1, \dots, y_m\}$ together creates either a *new* simple interference, or a strong interference, i.e., if there exist x_i and y_j that interfere. This check is done by the procedure `Resource.interfere`; it uses the procedure `Resource.killed` that gives, within a given resource, all the variables already killed by another variable. `Resource.killed` is given in a *formal* description, but obviously the information can be maintained and updated after each merge.

3.4 Pruning the affinity graph

The pruning phase is based on the interference analysis between resources. More formally, the optimization problem can be stated as follows. Let $G = (V, E_{\text{Affinity}})$ be the graph obtained from `Create_affinity_graph` (as explained in Section 3.1): the set V is the set of vertices labeled by resources and E_{Affinity} is the set of affinity edges between vertices. The goal is to prune (edge deletion) the graph G into $G' = (V, E_{\text{Pinned}})$ such that:

Condition 1: the cardinality of E_{Pinned} is maximized;

Condition 2: for each pair of resources $(v_1, v_2) \in V^2$ in the same connected component of G' , v_1 and v_2 do not interfere, i.e., `Resource.interfere`(v_1, v_2) = `false`.

In other words, the graph G is pruned into connected components such that the total number of deleted edges from E_{Affinity} is minimized and no two resources within the same connected component interfere.

First, because of **Condition 2**, all edges (v_1, v_2) in E_{Affinity} such that v_1 and v_2 interfere need to be removed from G . The obtained graph `PrePruned_G` is bipartite. Indeed, let $\{X_i\}_{1 \leq i \leq m}$, with $X_i = \phi(x_{i,1}, \dots, x_{i,n})$, be the set of ϕ instructions of the current basic block B . There are two kinds of vertices in G , the vertices for the definitions $V_{\text{DEFS}} = \{\text{Resource_def}(X_i)\}_{1 \leq i \leq m}$ and the other ones, for the arguments not already in V_{DEFS} , $V_{\text{ARGS}} = \{\text{Resource_def}(x_{i,j})\}_{1 \leq i \leq m, 1 \leq j \leq n} \setminus V_{\text{DEFS}}$. By construction, there is no affinity edge between two elements of V_{ARGS} . Also, because elements of V_{DEFS} strongly interfere together, there remains no edge between two elements of V_{DEFS} . Thus, G is indeed bipartite.

Unfortunately, even for a bipartite affinity graph, the pruning phase is NP-complete in the number of ϕ instructions (see Appendix B). Therefore, we use a heuristic algorithm based on a greedy pruning of edges, where edges with large weights are chosen first. The weight of an edge (x, y) is the number of neighbors of x (resp. y) that interfere with y (resp. x). This has the effect of first deleting edges that are more likely to disconnect more interfering vertices (see details in the procedure `Bipartite-Graph_pruning`). Note that, in the particular case of a unique ϕ instruction, this is identical to the “Process the unresolved resources” of the algorithm of Sreedhar et al. [11].

3.5 Merging the connected components

Once the affinity graph has been pruned, the resources of each connected component can be merged. We choose a reference resource in this connected component, either the physical resource if it exists (in this case, it is unique since two physical resources always interfere), or any resource otherwise. We change all pinnings to a resource of this component into a pinning to the reference resource. Finally,

we pin each variable (i.e., its definition) in the component to this reference resource. The correctness of this phase is insured by the absence of any strong interference inside the new merged resource. A formal description of the algorithm is given by the procedure `PrunedGraph_pinning`. In practice, the update of pinning need be performed only once, just before the mark phase, so requiring only one traversal of the control flow graph. Also note that the interference graph can be built incrementally at each call to `Resource_interfere` and updated at each resource merge, using a simple vertex-merge operation: hence, as opposed to the merge operation used in the iterated register coalescing algorithm [7] where interferences have to be recomputed at each iteration, here each vertex represents a SSA variable and merging is a simple edge union.

We point out that, after this phase, our algorithm relies on the mark and reconstruct phases of Leung and George’s algorithm. But we use several refinements, whose details are given in Appendix A.

4 Theoretical discussion

We now compare our algorithm with previous approaches, through hand crafted examples.

4.1 Our algorithm versus register coalescing

The out-of-SSA algorithm of Briggs et al. [1] relies on a Chaitin-style register coalescing to remove move instructions produced by the out of SSA translation. ABI constraints for a machine code level intermediate representation can be handled after the out of SSA translation by insertion of move instructions at procedure entry and exit, around function calls, and before 2-operand instructions. However, several reasons favor combined processing of coalescing and ABI renaming during the out-of-SSA phase:

[CC1] SSA is a higher level representation that allows a more accurate definition of interferences. For example (see Figure 9), it allows partial coalescing, i.e., the coalescing of a subset of the variable definitions.

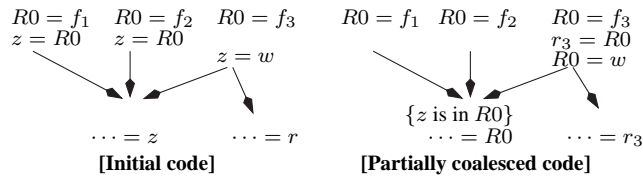


Figure 9: Because the physical register $R0$ and z interfere, [Initial code] cannot be coalesced by Chaitin’s register coalescing; even if the three definitions of $R0$ are constrained to be done on $R0$ (and then even in SSA “ $R0$ ” and “ z ” interfere), the pinning mechanism allows z and $R0$ to be coalesced, we say partially.

[CC2] The classical coalescing algorithm is greedy, so it may block further coalescings. Instead, for each merging point of the control flow graph, our algorithm optimizes *together* the set of coalescing opportunities for the set of ϕ instructions of this point.

[CC3] The main motivation of Leung and George’s algorithm is that ABI constraints introduce many additional move instructions. Some of these will be deleted by a dead code algorithm, but most of them will have to be coalesced. An important point of our method is the reduction of the overall complexity of the out-of-SSA renaming and coalescing phases: as explained in Section 3.5,

the complexity of the coalescings performed under the SSA representation benefits from the static single definition property.

4.2 Our algorithm versus the algorithm of Sreedhar et al.

The technique of Sreedhar et al. [11] consists in first translating the SSA form into CSSA (*Conventional SSA*) form. In CSSA, it is correct to replace all variable names that are part of a common ϕ instruction by a common name, then to remove all ϕ instructions. To go from SSA to CSSA however, we may create new variables and insert move instructions to eliminate ϕ variable interferences that would otherwise result in an incorrect program after renaming. Sreedhar et al. propose three algorithms to convert to CSSA form. We only consider the third one, which uses the interference graph and some liveness information to minimize the number of generated move instructions. Figures 10-12 illustrate some differences between the technique of Sreedhar et al. and ours.

[CS1] Sreedhar et al. optimize separately the replacement of each ϕ instruction. Our algorithm considers all the ϕ instructions of a given block to be optimized together. This can lead to a better solution as shown in Figure 10.

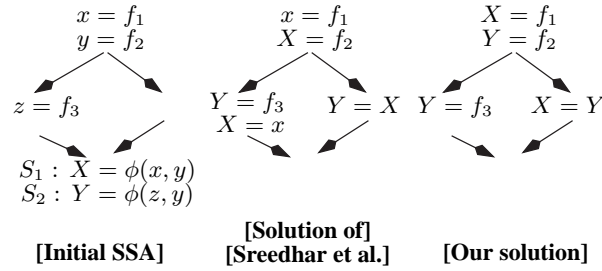


Figure 10: Sreedhar et al. treat S_1 and S_2 in sequence: for S_1 , $\{x, y\}$ interfere so $X = x$ is inserted and $\{y, X\}$ are regrouped in the resource \underline{X} ; for S_2 , $\{z, \underline{X}\}$ interfere so $Y = X$ is inserted and $\underline{Y} = \{z, Y\}$.

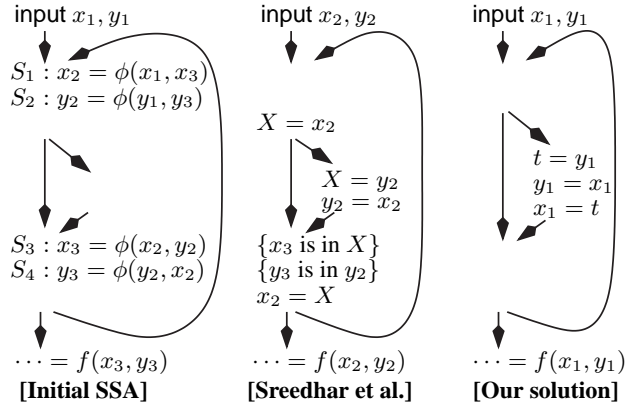


Figure 11: The superiority of using parallel copies. For the solution of Sreedhar et al. we suppose S_1, S_2, S_3 and S_4 were treated in this order.

[CS2] Sreedhar et al. process *iteratively* modify the initial SSA code by splitting variables. By doing so, information on interferences becomes scattered and harder to use. Thanks to pinning, throughout the process we are always reasoning on the initial SSA code. In particular, as illustrated by Figure 11, we can take advantage of the parallel copies placement.

[CS3] Finally, because our SSA representation is at machine level, we need to take into account ABI constraints. Figure 12 shows an example where we make a better choice of which variables to coalesce by taking the ABI constraints into account.

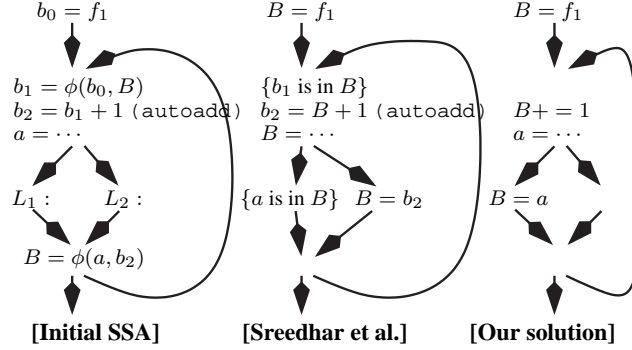


Figure 12: $\{a, b_2\}$ interfere: without the ABI constraints information, adding the move on block L_1 or L_2 is equivalent. Sreedhar et al. may make the wrong choice: treating the ABI afterward would replace the autoadd into $B = B + 1$; $b_2 = B$ (because $\{B, b_2\}$ interfere) resulting in one more move.

4.3 Limitations

Below are several points that expose the limitations of our approach:

[LIM1] Our algorithm is based on Leung and George’s algorithm that decides the place where move instructions are inserted. Also, we use an approximation of the cost of an interference compared to the gain of a pinning. Hence, even if we could provide an optimal solution to our formulation of the problem, this solution would not necessarily be an optimal solution for the minimization of move instructions.

[LIM2] As explained in Section 2.3, the main limitation of Leung and George’s algorithm is that, when the use of a variable is pinned to a resource, it does not try to coalesce its definition with this resource. This can be avoided by using a pre-pass to pin the variable definitions. But, as illustrated by Figure 13, repairing variables that are introduced during Leung and George’s repairing phase cannot be handled this way.

[LIM3] As explained in Appendix B, our ϕ coalescing problem is NP-complete. Note also that a simple extension of the proof shows the NP-completeness of the problem of minimizing the number of move instructions.

[LIM4] Finally, in the case of strong register pressure, the problem becomes different: coalescing (or splitting) variables has a strong impact on the colorability of the interference graph during the register allocator phase (e.g. [9]). But this goes out of the scope of this paper.

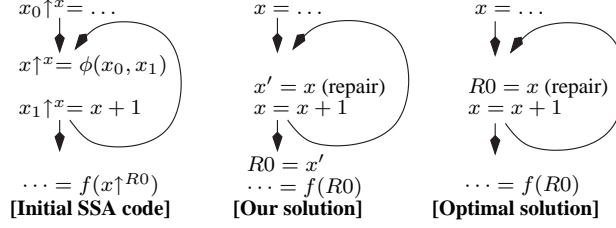


Figure 13: Limitation of Leung and George’s repairing process: the repairing variable x' is not coalesced with further uses.

5 Results

We conducted our experiments on several benchmarks represented in LAI code. Since the LAI language supports predicated instructions, the LAO tool uses a special form of SSA, named ψ -SSA [13], which introduces ψ instructions to represent predicated code under SSA. In brief, ψ instructions introduce constraints similar to 2-operands constraints, and are handled in our algorithm in a special pass where they are converted into a “ ψ -conventional” SSA form. This does not change the results presented in this section.

In the following, *VALcc1* and *VALcc2* refer to the same set of C functions compiled into LAI code with two different ST120 C compilers. This set includes about 40 small functions with some basic digital signal processing kernels, integer Discrete Cosine Transform, sorting, searching, and string searching algorithms. The benchmarks *example1-8* are small examples written in LAI code specifically for the experiment. *LAI Large* is a set of larger functions, most of which come from the efr 5.1.0 vocoder from the ETSI [6]. Finally, *SPECint* refers to the SPEC CINT2000 benchmark [12].

To show the superiority of our approach, we have implemented the following algorithms:

[Leung] The algorithm of Leung and George contains the collect, and the mark-reconstruct (say **out-of-pinned-SSA**) phases. For some reasons given further, the collect phase has been split into three parts, namely **pinning_{SP}** (collect constraints related to the dedicated register *SP*), **pinning_{ABI}** (collect remaining renaming constraints) and **pinning ϕ** (our algorithm). Each of these pinning phases can be activated or not, independently.

[Sreedhar] The algorithm of Sreedhar et al. has been implemented with an additional pass, namely **pinning_{CSSA}**. The pinning_{CSSA} phase pins all the operands of a ϕ to a same resource, and allows the out-of-pinned-SSA phase to be used as an out-of-CSSA algorithm.

[Naive_{ABI}] is an algorithm that adds when necessary `move` instructions locally around renaming constrained instructions. This pass can be used when the pinning_{ABI} pass has not been activated.

[Coalescing] Finally, we have implemented a repeated register coalescer [5]. As for the iterated register coalescer it is a conservative coalescer when used during the register allocation phase. But, outside of the register allocation context like here, it is an aggressive coalescing that does not take care of the colorability of the interference graph.

As already mentioned in Section 2.2, coalescing variables constrained by a dedicated register like the *SP* register can generate incorrect code. Similarly, splitting the SSA web of such variables poses some problems. Hence, it was not possible to ignore those renaming constraints during the out-of-SSA phase and to treat them afterwards. That explains the differentiation we made between pinning_{SP} and pinning_{ABI} passes: we choose to always execute pinning_{SP}. Also, we tried to modify the algorithm of Sreedhar et al. to support *SP register* constraints. However, our implementation still performs some illegal variable splitting on some codes: the final non-SSA code contains fewer `move`

instructions, but is incorrect. Such cases mainly occurred with SPECint, and thus *the SPECint figures for the experiments including the algorithm of Sreedhar et al. must be taken only as an optimistic approximation of the number of move instructions.*

	Experiments						out-of-pinned-SSA	NativeABI	Coalescing
		Sreedhar	pining _{CSSA}	pining _{SP}	pining _{ABI}	pining _{ϕ}			
Table 2	$\underline{L_\phi}+C$			•		•		•	
	C			•		•		•	
	$S_\phi+C$	•	•	•		•		•	
Table 3	$\underline{L_\phi, ABI}+C$			•	•	•		•	
	$S_\phi+L_{ABI}+C$	•	•	•	•	•		•	
	$L_{ABI}+C$			•	•	•		•	
	C			•		•	•	•	
Table 4	$\underline{L_\phi, ABI}$			•	•	•			
	S_ϕ	•	•	•		•	•		
	L_{ABI}			•	•	•			

Table 1: Details of implemented versions

Tables 2-5 compares the number of resulting move instructions on the different out-of-SSA algorithms detailed in Table 1. In particular, we illustrate here comparisons [CC1-3] and [CS1-3] exposed in Section 4. In the tables, values with + or - are always relative to the first column of the table.

Comparison without ABI constraints Table 2 compares different approaches when renaming constraints are ignored. As explained above only non-SP register related constraints, which we improperly call ABI constraints, could be ignored in practice. Columns $S_\phi+C$ vs $\underline{L_\phi}+C$ illustrate points [CS1-2]. Columns C vs $\underline{L_\phi}+C$ illustrate points [CC1-2]. Point [CC1] is also illustrated by $S_\phi+C$ vs C. In those experiments, our algorithm is better or equal in all cases, except for the SPECint benchmark with the algorithm of Sreedhar et al.. But $S_\phi+C$ are optimistic results as explained before. It shows the superiority of our approach in the absence of ABI constraints.

benchmark	$\underline{L_\phi}+C$	C	$S_\phi+C$
VALcc1	193	+59	+3
VALcc2	170	+44	+13
example1-8	14	+3	+3
LAI.Large	438	+44	+48
SPECint	6803	+3135	-59

Table 2: Comparison of move instruction count with no ABI constraint.

Comparison with renaming constraints Table 3 shows the variation in the number of move instructions of various out-of-SSA register coalescing algorithms, when all renaming constraints are taken into account. Comparison of $S_\phi+L_{ABI}+C$ and $L_{ABI}+C$ vs $\underline{L_\phi, ABI}+C$ confirms points [CS3]. Column C shows the importance of treating the ABI with the algorithm of Leung et al.: many move instructions could not be removed by the dead code and aggressive coalescing phases. Our algorithm leads to less move instructions in all cases which shows the superiority of our approach with renaming constraints.

benchmark	$L_{\phi,ABI+C}$	$S_{\phi}+L_{ABI+C}$	L_{ABI+C}	C
VALcc1	242	+7	+3	+386
VALcc2	220	+15	+29	+449
example1-8	15	+3	+3	+18
LAI_Large	1085	+26	+62	+634
SPECint	23930	+413	+482	+38623

Table 3: Comparison of move instruction count with renaming constraints.

Compilation time Repeated register coalescing is an expensive optimization phase in terms of time and space; its complexity is proportional to the number of move instructions in the program. Almost all coalescings are handled by our algorithm during the out of SSA translation. As explained in [2] the creation and the maintenance of the interference graph is highly simplified under the SSA form. Hence, as mentioned in Point [CC3], the more move instructions are handled at the SSA level, the lower is the compilation time for the overall coalescing. Table 4 gives an evaluation of the number of move instructions that would remain after the out-of-SSA phase if only naive techniques were applied for the ϕ replacement (which we denote ϕ moves) and for the renaming constraints treatment (which we improperly call ABI moves). Hence, it gives an evaluation of the cost of running a repeated register coalescing after one simple SSA rename back phase. We did not provide timing figures for the overall out-of-SSA and register coalescing phase for the different experiments because our implementation is too experimental and not optimized enough to give usable results.

benchmark	$L_{\phi,ABI}$	S_{ϕ} ABI moves	L_{ABI} ϕ moves
VALcc1	277	+593	+690
VALcc2	245	+926	+749
example1-8	16	+38	+34
LAI_Large	1447	+4543	+6161
SPECint	36882	+249481	+260095

Table 4: Comparison of move instruction count before the repeated register coalescing phase.

Variations on our algorithm Table 5 compares small variations in the implementation of our algorithm. The base column reports *weighted move* count, where move instructions are given a weight equal to 5^d , d being the nesting level, i.e. depth, of the loop the move belongs to. 5^d is an arbitrary

benchmark	base	depth	opt	pess
VALcc1	1109	+1	+4	+1484
VALcc2	877	+1	+8	+1716
example1-8	32	+0	+0	+4
LAI_Large	17594	+60	+7	+22116
SPECint	1652065	-1798	+7258	+3038712

Table 5: Weighted count of move instructions on variants of our algorithm.

weight that corresponds to a static approximation where each loop would contain 5 iterations.

Our first variation (*depth*) is based on the simple remark that in our initial implementation we prioritized the ϕ instructions according to their depth, instead of the depth of the `move` instructions they will generate. For this variation, we use a new `Create_affinity_graph` procedure (Algorithm 3) with a depth constraint that calls `Program_pinning` with decreasing depth. This leads to a very small improvement on SPECint and a small degradation for LAI.Large. This result confirms the observation we made that affinity and interference graphs are not complex enough to motivate a global optimization scheme.

Our second (*opt*) and third (*pess*) variations use relaxed definitions of interferences, respectively *optimistic* and *pessimistic* (Algorithm 4). It is interesting to note that optimistic interferences only incur a relatively small increase in the number of `move` instructions while significantly reducing the complexity of the computation of the interference graph.

6 Conclusion

This paper presents a pinning-based solution to the problem of register coalescing during the out-of-SSA translation phase. We explain and demonstrate why considering ϕ instruction replacement and renaming constraints together results in an improved coalescing of variables, thus reducing the number of `move` instructions before instruction scheduling and register allocation. We show the superiority of our approach both in terms of compile time and number of copies compared to solutions composed of existing algorithms (Sreedhar et al., Leung and George, Briggs et al., repeated register coalescing). These experiments also show that the affinity and interference graphs are usually quite simple, which means that a global optimization scheme would bring very little improvement over our local approach. Finally, we implemented some small variations of our algorithm, and observed that an optimistic implementation of interferences still provides good results with a significant reduction in the complexity of the computation of the interference graph. During this work, we also improved slightly the mark and reconstruct phases of Leung and George's algorithm, which we rely on.

References

- [1] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software – Practice and Experience*, 28(8):859–881, July 1998.
- [2] Z. Budimlic, K. Cooper, T. Harvey, K. Kennedy, T. Oberg, and S. Reeves. Fast copy coalescing and live-range identification. In *SIGPLAN International Conference on Programming Languages Design and Implementation*, pages 25–32. ACM Press, June 2002.
- [3] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, 1982.
- [4] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451 – 490, 1991.
- [5] B. Dupont de Dinechin, F. de Ferrière, C. Guillon, and A. Stouthinin. Code generator optimizations for the ST120 DSP-MCU core. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 93 – 103, 2000.

- [6] European Telecommunications Standards Institute (ETSI). GSM technical activity, SMG11 (speech) working group. Available at <http://www.etsi.org>.
- [7] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3), May 1996.
- [8] A. L. Leung and L. George. Static single assignment form for machine code. In *SIGPLAN International Conference on Programming Languages Design and Implementation*, pages 204 – 214, 1999.
- [9] J. Park and S.-M. Moon. Optimistic register coalescing. In *IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 196–204, 1998.
- [10] M. Sassa, T. Nakaya, M. Kohama, T. Fukukoa, and M. Takahashi. Static Single Assignment form in the COINS compiler infrastructure.
- [11] V. Sreedhar, R. Ju, D. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Static Analysis Symposium, Italy*, pages 194 – 204, 1999.
- [12] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2000 benchmarks. Available at <http://www.spec.org/cpu2000/CINT2000/>.
- [13] A. Stoutchinin and F. de Ferrière. Efficient static single assignment form for predication. In *34th annual ACM/IEEE international symposium on Microarchitecture*, pages 172–181. IEEE Computer Society, 2001.
- [14] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *MICRO – International Symposium on Microarchitecture*, pages 1–11, New York, NY, USA, 1994. ACM Press.

Algorithm 2 Formal description of our algorithm.**Create_affinity_graph**(CFG_Node current_node)

```

(E, V) = ( $\emptyset$ ,  $\emptyset$ )
for each  $X = \phi(x_1, \dots, x_n)$  of current_node
  V = V  $\cup$  {Resource_def(X)}
  for each  $x \in \{x_1, \dots, x_n\}$ 
    V = V  $\cup$  {Resource_def(x)}
  e = (Resource_def(X), Resource_def(x))
  if ( $e \notin E$ ) multiplicity(e)=0
    E = E  $\cup$  {e}, multiplicity(e)++
return G = (E, V)

```

Graph_InitialPruning(Graph (V, E))

```

foreach ( $x_1, x_2$ )  $\in E$ ,
  if (Resource_interfere( $x_1, x_2$ ))
    E -= ( $x_1, x_2$ )
return (V, E)

```

BipartiteGraph_pruning(Bipartite_Multi_Graph (V, E))

```

{ Evaluates the weight for each edge }
for all  $e \in E$ , weight(e)=0
for all ( $(x, x_1), (x, x_2)$ )  $\in E^2$  such that  $x_1 \neq x_2$ 
  if Resource_interfere( $x_1, x_2$ )
    weight( $(x, x_1)$ ) += multiplicity( $(x, x_2)$ )
    weight( $(x, x_2)$ ) += multiplicity( $(x, x_1)$ )
{ Prunes in decreasing weight order
  and update the weight }
while weight( $e_p$ ) > 0
let  $e_p = (X, x)$  such that
   $\forall e \in E$ , weight( $e_p$ )  $\geq$  weight(e)
do
  E -=  $e_p$ 
  for all  $e = (X, y) \in E$ 
    weight(e) -= multiplicity( $e_p$ )
  for all  $e = (Y, x) \in E$ 
    weight(e) -= multiplicity( $e_p$ )
return (V, E)

```

PrunedGraph_pinning(Graph G, Program P)

```

foreach  $V \in \{\text{connected components of G}\}$ 
  let  $\underline{u} = \bigcup_{v \in V} v$ 
  let  $\underline{w} = \begin{cases} v_i & \text{if } v_i \in V \text{ is a physical resource} \\ \underline{u} & \text{otherwise} \end{cases}$ 
  foreach (OP)  $d_1, \dots = instr(a_1, \dots) \in P$ 
    foreach  $d_i$  such that  $d_i \in \underline{u}$ 
      pin  $d_i$  to  $\underline{w}$  in (OP)
    foreach  $a_i \uparrow^r$  such that  $r \in V$ 
      replace  $r$  by  $\underline{w}$ 

```

Variable_kills(Variable a, Variable b)

```

if the definition of b dominates those of a
  and a and b interfere
  return true {Case 1}
if a is defined as  $a = \phi(a_1 : B_1, \dots, a_n : B_n)$ 
  for  $i = 1$  to n
    if b is live out of  $B_i$  and  $b \neq a_i$ 
      return true {Case 2}
return false

```

Variable_stronglyInterfere(Variable a, Variable b)

```

if a and b are defined by  $\phi$  instructions
  let  $a : B_a = \phi(a_1 : B_{a,1}, \dots, a_n : B_{a,n})$ 
  let  $b : B_b = \phi(b_1 : B_{b,1}, \dots, b_m : B_{b,m})$ 
  if  $B_a = B_b$  return true {Case 4}
  for  $i = 1$  to n
    if  $B_{a,i}$  is a predecessor of  $B_b$ 
      let  $B_{a,i} = B_{b,j}$ 
      if  $a_i \neq b_j$  return true {Case 3}
  return false
else if a and b are defined in the same instruction
  let  $(\dots a \dots b \dots) = instr(\dots)$ 
  return true
return false

```

Resource_killed(Resource \underline{A})

```

let  $\underline{A} = \{a_1, \dots, a_n\}$ 
killed_withinA =
   $\{a_i \in \underline{A} \mid \exists a_j \in \underline{A}, \text{Variable\_kills}(a_j, a_i)\}$ 
return killed_withinA

```

Resource_interfere(Resource \underline{A} , Resource \underline{B})

```

let  $\underline{A} = \{a_1, \dots, a_n\}$ 
let  $\underline{B} = \{b_1, \dots, b_m\}$ 
let killed_withinA = Resource_killed( $\underline{A}$ )
let killed_withinB = Resource_killed( $\underline{B}$ )
if  $\underline{A}$  and  $\underline{B}$  are physical resources
  if  $\underline{A} \neq \underline{B}$  return true
for all  $(a, b) \in \underline{A} \times \underline{B}$ 
  if  $a \notin \text{killed\_withinA}$  and Variable_kills(b, a)
    return true
  if  $b \notin \text{killed\_withinB}$  and Variable_kills(a, b)
    return true
  if Variable_stronglyInterfere(a, b)
    return true
return false

```

Algorithm 3 Construction of initial affinity graph with a depth constraint.

Create_affinity_graph(CFG_Node *current_node*,
Integer *depth*)

$(E, V) = (\emptyset, \emptyset)$
 for each $X = \phi(x_1, \dots, x_n)$ of *current_node*
 $V = V \cup \{\text{Resource_def}(X)\}$
 for each $x \in \{x_1, \dots, x_n\}$
 let Node_ x : $x = \dots$
 if $\text{depth}(\text{Node}_x) \neq \text{depth}$
 continue
 $V = V \cup \{\text{Resource_def}(x)\}$
 $e = (\text{Resource_def}(X), \text{Resource_def}(x))$
 if $(e \notin E)$ $\text{multiplicity}(e) = 0$
 $E = E \cup \{e\}$, $\text{multiplicity}(e)++$
 return $G = (E, V)$

Algorithm 4 Optimistic and pessimistic definition of interferences.

Variable_kills_optimistic(Variable a , Variable b)
 let Node_ a : (Def_ a) $a = \dots$
 let Node_ b : (Def_ b) $b = \dots$
 if $(a \neq b)$ and (Def_ b dominates Def_ a) and
 $(b \in \text{liveout}(\text{Node}_a))$
 return true {Case 1}
 if a is defined as $a = \phi(a_1 : B_1, \dots, a_n : B_n)$
 for $i = 1$ to n
 if b is live out of B_i and $b \neq a_i$
 return true {Case 2}
 return false

Variable_kills_pessimistic(Variable a , Variable b)
 let Node_ a : (Def_ a) $a = \dots$
 let Node_ b : (Def_ b) $b = \dots$
 if $(a \neq b)$ and (Def_ b dominates Def_ a) and
 $((b \in \text{livein}(\text{Node}_a)) \text{ or } (\text{Node}_a = \text{Node}_b))$
 return true {Case 1}
 if a is defined as $a = \phi(a_1 : B_1, \dots, a_n : B_n)$
 for $i = 1$ to n
 if b is live out of B_i and $b \neq a_i$
 return true {Case 2}
 return false

Appendix A: Limitations and refinement of Leung’s algorithm

Once pinning has been performed, our algorithm relies on Leung’s *mark* and *reconstruct* algorithms to restore the code into non-SSA form. Critical edges are subject to a particular treatment in Leung’s algorithm. But as illustrated by Figure 14, the solution is not robust enough when dealing with aggressive pinning. The goal of this appendix is to propose a clearest semantic for ϕ functions, and to modify Leung’s algorithm accordingly.

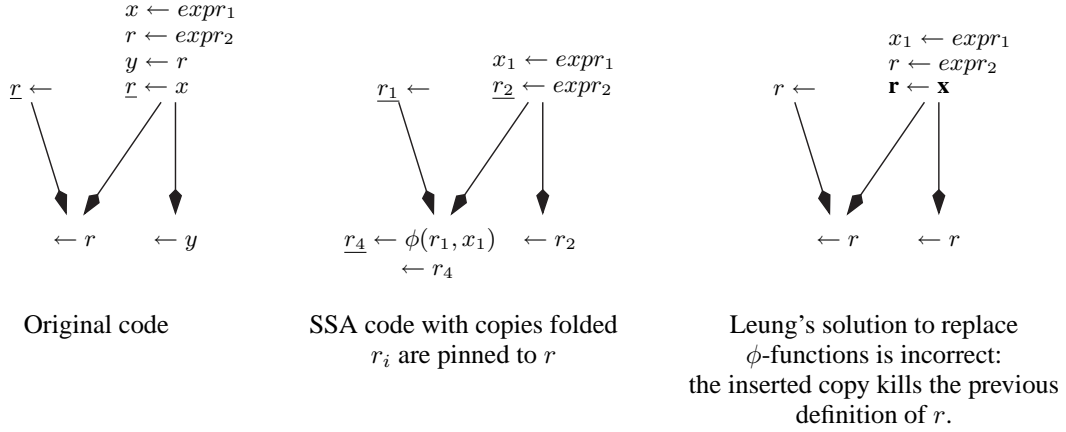


Figure 14: The ϕ -function replacement conflict problem

To begin with, let us consider a ϕ definition $B : y = \phi(\dots)$. The semantic used by Leung et al. is that this definition is distributed over each predecessor of B . Hence, in a certain sense multiple definitions of Y coexist and therefore may conflict. Because conflicts for simple variables (not resources) are not taken into account in Leung’s algorithm, the lost copy problem has a special treatment that corresponds to the lines below the “(*fix problem related to critical edge*)”. Here, the copy $cp_3 := cp_3 \cup \{w \leftarrow z\}$ is incorrect (probably a typo) and it is difficult to fix to obtain a correct and efficient code. Instead, that whenever the definition of y is not pinned to any resource, we propose to create a virtual resource \underline{y} and to pin this definition to it. This fix takes place in the COLLECT procedure.

Another consequence of Leung’s ϕ function semantic is that whenever y has to be repaired, the repairing copies are also distributed over each predecessors of B . Hence conflicts can occur and those repairing copies cannot be used further B (which explains the need to introduce another repairing copy w). Because we found no a priori motivation to do so, we propose to place the repairing copy of y just after its definition instead. Hence, our new semantic of a given $B : y \uparrow^r = \phi(x_1 : B_1, \dots, x_n : B_n)$ (where y is always pinned to a resource r) definition is the following:

- at the end of each block B_i , there is a new virtual instruction that defines no variable but that uses $x_i \uparrow^r$.
- at the beginning of the block B , the ϕ instruction contains no use arguments, but defines $y \uparrow^r$.
- all the “virtual uses” of the end of each block have a parallel semantic i.e. are considered all together.

The consequence is a simplification of the code: whenever instructions of a block have to be traversed then ϕ functions definitions, normal instructions uses, normal instructions definitions and ϕ functions uses (of each successors) are considered consecutively.

The refined code is given below, modified code is written using the \triangleright sign.

Finally, we would like to outline the problem with dedicated register pinning. Indeed, we could find in Leung's collect phase the code "if y was renamed from some dedicated register r during SSA construction then $must_def[y] = r\dots$ ". As illustrated by Figure 15 copy-folding performed on dedicated register definition can lead to an incorrect pinning. Because of its non local property, this inconsistency is not trivial to detect while doing the optimization. Freezing optimizations when dealing with dedicated registers is a solution to this problem. On the other hand the semantic is not necessarily strict enough to justify such a decision and pinning may be performed correctly while being aware of this specific semantic. Hence because dedicated registers related pinning that is semantic aware can be very complex, we have intentionally removed this part from the COLLECT procedure and delegated it to a previous pinning phase.

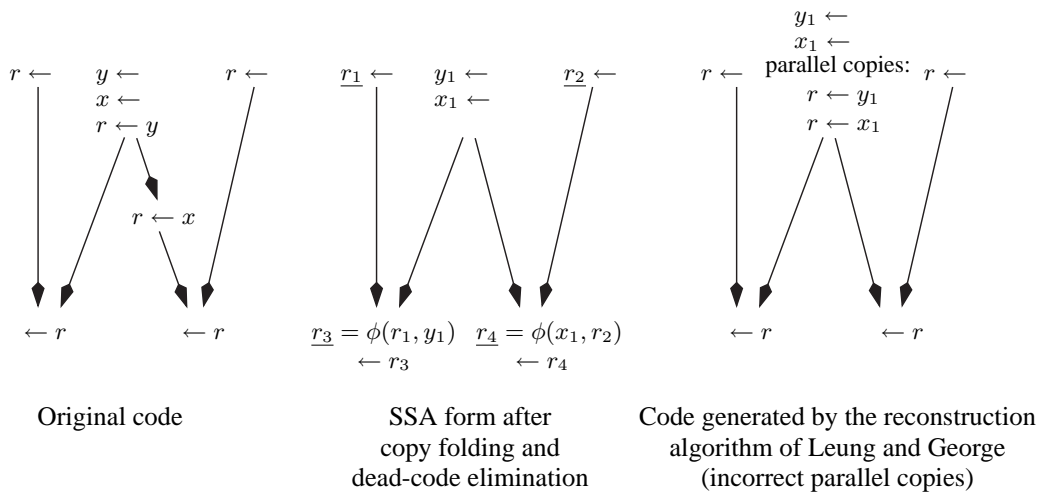


Figure 15: The *parallel-copies conflict problem* is generated by a too constrained pinning

```

procedure COLLECT
initialize all entries of must_def and must_use to  $\perp$ 
R :=  $\emptyset$ 
for b  $\in$  basic blocks do
  for i  $\in$   $\phi$ -functions in b do
    let  $i \equiv y \leftarrow \phi(x_1 \dots x_n)$ 
     $\triangleright$  if pinned_def(i, 1)
     $\triangleright$  then let r be the dedicated register required
     $\triangleright$  else let r = y
      must_def[y] = r
      R := R  $\cup$  {r}
  for i  $\in$  non- $\phi$ -functions in b do
    let  $i \equiv y_1 \dots y_m \leftarrow op(x_1 \dots x_n)$ 
    for j := 1 to m do
      if pinned_def(i, j) then
        let r be the dedicated register required
        must_def[yj] := r
        R := R  $\cup$  {r}
      for j := 1 to n do
        if pinned_use(i, j) then
          let r be the dedicated register required
          must_use[i][j] := r
          R := R  $\cup$  {r}

procedure MARKINIT
for b  $\in$  basic blocks do
for r  $\in$  R do
  sites[r] :=  $\emptyset$ 
for r  $\in$  R do
  last[b][r] :=  $\top$ 
  phi[b][r] :=  $\top$ 
for i  $\in$   $\phi$ -functions in block b do
  let  $i \equiv y \leftarrow \phi(x_1 \dots x_n)$ 
  if must_def[y] = r  $\neq \perp$  then
    phi[b][r] = last[b][r] = y
    sites[r] := sites[r]  $\cup$  {b}
for i  $\in$  normal instructions in block b do
  let  $i \equiv y_1 \dots y_m \leftarrow op(x_1 \dots x_n)$ 
  for j := 1 to n do
    if must_use[i][j] = r  $\neq \perp$  then
      last[b][r] := xj
      sites[r] := sites[r]  $\cup$  {b}
    for j := 1 to m do
      if must_def[yj] = r  $\neq \perp$  then
        last[b][r] := yj
        sites[r] := sites[r]  $\cup$  {b}
   $\triangleright$  for b'  $\in succ_{cfg}(b)$  do
   $\triangleright$  let b be the jth predecessor of b'
   $\triangleright$  for i  $\in$   $\phi$ -functions in b' do
   $\triangleright$  let r  $\equiv must\_def$ [y]
   $\triangleright$  last[b][r] = xj
   $\triangleright$  sites[r] := sites[r]  $\cup$  {b}

```

```

procedure MARK
MARKINIT()
for r  $\in$  R do
  repair_name[r] =  $\perp$ 
  repair_sites[r] =  $\emptyset$ 
for b  $\in$  basic blocks do
for r  $\in$  R do
  avail[r] := avin[b][r]
for i  $\in$  normal instructions in b do
  let  $i \equiv y_1 \dots y_m \leftarrow op(x_1 \dots x_n)$ 
  for j := 1 to n do USE(i, j, xj)
  for j := 1 to m do DEFINE(yj)
for b'  $\in succ_{cfg}(b)$  do
  let b be the jth predecessor of b'
  for i  $\in$   $\phi$ -functions in b' do
    let  $i \equiv y \leftarrow \phi(\dots x_j \dots)$ 
    USE(i, j, xk)

procedure USE(i, j, x)
in_place[i][j] = false
if must_use[i][j]  $\neq \perp$  and avail[must_use[i][j]] = x then
  in_place[i][j] = true
return
if must_def[x]  $\neq \perp$  and avail[must_def[x]]  $\neq x$  then
  if repair_name[x] =  $\perp$  then
    repair_name[x] := a new SSA name
    repair_sites[x] := repair_sites[x]  $\cup$  {i}
  if must_use[i][j]  $\neq \perp$  then
    avail[must_use[i][j]] := x

procedure DEFINE(x)
if must_def[x]  $\neq \perp$  then avail[must_def[x]] := x

avin[b][r] =
  
$$\begin{cases} \perp & \text{if } b \text{ is the entry} \\ x & \text{if } \phi[i][r] = x \neq \text{top} \\ \bigcap_{b' \in pred_{cfg}(b)} \text{avin}[b'][r] & \text{if } b \in DF^+(sites[r]) \\ \text{avout}[idom(b)][r] & \text{otherwise} \end{cases}$$


```

$$avout[b][r] = \begin{cases} x & \text{if } last[b][r] = x \neq \top \\ \text{avin}[b][r] & \text{otherwise} \end{cases}$$

```

procedure LOOKUP( $i, x$ )
if  $stacks[x]$  is empty then
  if  $must\_def[x] \neq \perp$  then
    return  $must\_def[x]$ 
  else return  $x$ 
else
  let  $(y, sites) = top(stacks[x])$ 
  if  $i \in sites$  then return  $y$ 
  else if  $must\_def[x] \neq \perp$  then return  $must\_def[x]$ 
  else return  $x$ 

```

```

procedure RENAME_USE( $i, j, x, copies$ )
let  $y = LOOKUP(i, x)$ 
let  $r = must\_use[i][j]$ 
if  $in\_place[i][j]$  then
  rewrite the  $j$ th input operand of  $i$  to  $r$ 
else if  $r \neq \perp$  then
   $copies := copies \cup \{r \leftarrow y\}$ 
  rewrite the  $j$ th input operand of  $i$  to  $r$ 
else
  rewrite the  $j$ th input operand of  $i$  to  $y$ 
return  $copies$ 

```

```

procedure RENAME_DEF( $i, j, y, copies$ )
let  $r = \text{if } must\_def[y] = \perp \text{ then } y \text{ else } must\_def[y]$ 
rewrite the  $j$ th output operand  $y$  to  $r$ 
if  $repair\_name[y] = tmp \neq \perp$  then
  push  $(tmp, repair\_sites[y])$  onto  $stacks[y]$ 
   $copies := copies \cup \{tmp \leftarrow r\}$ 
return  $copies$ 

```

```

procedure RECONSTRUCT( $b$ )
for  $i \in \phi$ -functions in  $b$  do
  let  $i \equiv y \leftarrow \phi(x_1 \dots x_n)$ 
   $cp_4 = \emptyset$ 
   $cp_4 = RENAME\_DEF(i, 1, y, cp_4)$ 
  insert parallel copies  $cp_4$  after  $i$ 
for  $i \in$  normal instructions in  $b$  do
  (* rewrite instructions *)
  let  $i \equiv y_1 \dots y_m \leftarrow op(x_1 \dots x_n)$ 
   $cp_1 := \emptyset$ 
  for  $j := 1$  to  $n$  do
     $cp_1 := RENAME\_USE(i, j, x_j, cp_1)$ 
  insert parallel copies  $cp_1$  before  $i$ 
   $cp_2 := \emptyset$ 
  for  $j := 1$  to  $m$  do
     $cp_2 := RENAME\_DEF(i, j, y_j, cp_2)$ 
  insert parallel copies  $cp_2$  after  $i$ 
   $cp_3 := \emptyset$ 
  (* compute  $\phi$ -copies *)
for  $b' \in succ_{cf_g}(b)$  do
  let  $b$  be the  $k$ th predecessor of  $b'$ 
  for  $i \in \phi$ -functions in  $b'$  do
    let  $i \equiv y \leftarrow \phi(x_1 \dots x_k \dots x_n)$ 
   $cp_3 := \emptyset$ 
  for  $j := 1$  to  $n$  do
     $cp_3 := RENAME\_USE(i, j, x_j, cp_3)$ 
  insert parallel copies  $cp_3$  at the end of block  $b$ 
for  $b' \in succ_{dom}(b)$  do
  RECONSTRUCT( $b'$ )
  Restore  $stacks[]$  to its state
  at the beginning of this call

```

Appendix B: NP-completeness results

This appendix is devoted to the proof of LOCAL PINNING NP-completeness. Also, because this proof can be extended to the global problem GLOBAL PINNING the corresponding proof is provided. Remark that the result is valid with or without renaming constraints. For simplicity, proofs are made without renaming constraints. Also, additional remarks concerning the coalescing problem complexity in its general form are provided.

We start with a few definitions.

Definition 1 (GLOBAL PINNING) *Consider a SSA program P containing no initial pinning and a set of ϕ definitions $X_i = \phi(x_{i,1}, \dots, x_{i,n_i})$. Let us denote by $DEFS = \{X_1, \dots, X_n\}$, $ARGS_i = \{x_{i,1}, \dots, x_{i,n_i}\}$, $ARGS = \bigcup_i ARGS_i$ and $\mathcal{V} = DEFS \cup ARGS$. Find a partitioning of \mathcal{V} into disjoint sets R_1, \dots, R_m such that*

$$(CK): \bigcup_{v \in \mathcal{V}} \text{Resource_killed}(\{v\}) = \bigcup_{1 \leq j \leq m} \text{Resource_killed}(R_j) \quad (\text{no more killed variable})$$

$$(CS): \forall 1 \leq j \leq m, \forall (x, y) \in R_j^2, \neg \text{Variable_stronglyInterfere}(x, y) \quad (\text{no strong interference})$$

$$(CM): \text{card} \left[\left(\bigcup_{1 \leq i \leq n} DEFS \times ARGS_i \right) \cap \left(\bigcup_{1 \leq i \leq m} R_i^2 \right) \right] \quad \text{is maximized}$$

Definition 2 (LOCAL PINNING) *Consider a program P with some pinning already performed and a set of ϕ definitions $X_i = \phi(x_{i,1}, \dots, x_{i,n_i})$ within the same block B . Let us denote by $DEFS = \{X_1, \dots, X_n\}$, $ARGS_i = \{x_{i,1}, \dots, x_{i,n_i}\}$, $ARGS = \bigcup_i ARGS_i$, $\mathcal{V} = DEFS \cup ARGS$ and \underline{x} the set of variables pinned to the same resource than $x \in \mathcal{V}$. Find a partitioning of the set of resources $\underline{\mathcal{V}}$ into disjoint sets $\underline{R}_1, \dots, \underline{R}_m$ such that*

$$\bigcup_{v \in \underline{\mathcal{V}}} \text{Resource_killed}(v) = \bigcup_{1 \leq j \leq m} \text{Resource_killed}(\bigcup \underline{R}_j) \quad (\text{no more killed variable})$$

$$\forall 1 \leq j \leq m, \forall (x, y) \in \left(\bigcup \underline{R}_j \right)^2, \neg \text{Variable_stronglyInterfere}(x, y) \quad (\text{no strong interference})$$

$$\text{card} \left[\left(\bigcup_{1 \leq i \leq n} DEFS \times ARGS_i \right) \cap \left(\bigcup_{1 \leq i \leq m} \left(\bigcup \underline{R}_i \right)^2 \right) \right] \quad \text{is maximized}$$

Theorem 1 GLOBAL PINNING is NP-complete in the size of ϕ functions.

Proof of Theorem 1 We prove the theorem using the polynomial reduction to MAXIMUM-INDEPENDENT-SET. Hence, let us consider a graph $G = (V, E)$ where $V = \{x_1, \dots, x_n\}$. We aim to find a maximum independent set $I \subset V$ i.e.

$$\begin{cases} \text{card } I \text{ is maximum} \\ \text{for each } (x_i, x_j) \in I^2, (x_i, x_j) \notin E \end{cases}$$

Let us build the corresponding instance of GLOBAL PINNING :

- For all $x_i \in V$, consider a block B_i which contains a definition of x_i
- For all $(x_i, x_j) \in E$ consider
 1. two blocks $B_{i_{ij}}$ and $B_{j_{ij}}$ with predecessor B_i and resp. B_j . A third block B_{ij} with predecessors $B_{i_{ij}}$ and $B_{j_{ij}}$.
 2. a definition $a_{j_{ij}}$ in block $B_{i_{ij}}$ and a definition $a_{i_{ij}}$ in block $B_{j_{ij}}$
 3. two ϕ functions in block B_{ij} : $C_{i_{ij}} = \phi(x_i, a_{i_{ij}})$
 $C_{j_{ij}} = \phi(a_{j_{ij}}, x_j)$
- A block B , with predecessors B_1, \dots, B_n , which contains the instruction $X = \phi(x_1, \dots, x_n)$

For this program,

- there is an affinity between X and all x_i ; for each $(x_i, x_j) \in E$ there are affinities between $C_{i_{ij}}$ and x_i and between $C_{i_{ij}}$ and $a_{i_{ij}}$
- interferences are between all couple $a_{i_{ij}}$ and x_j and between all couple $C_{i_{ij}}$ and $C_{j_{ij}}$.

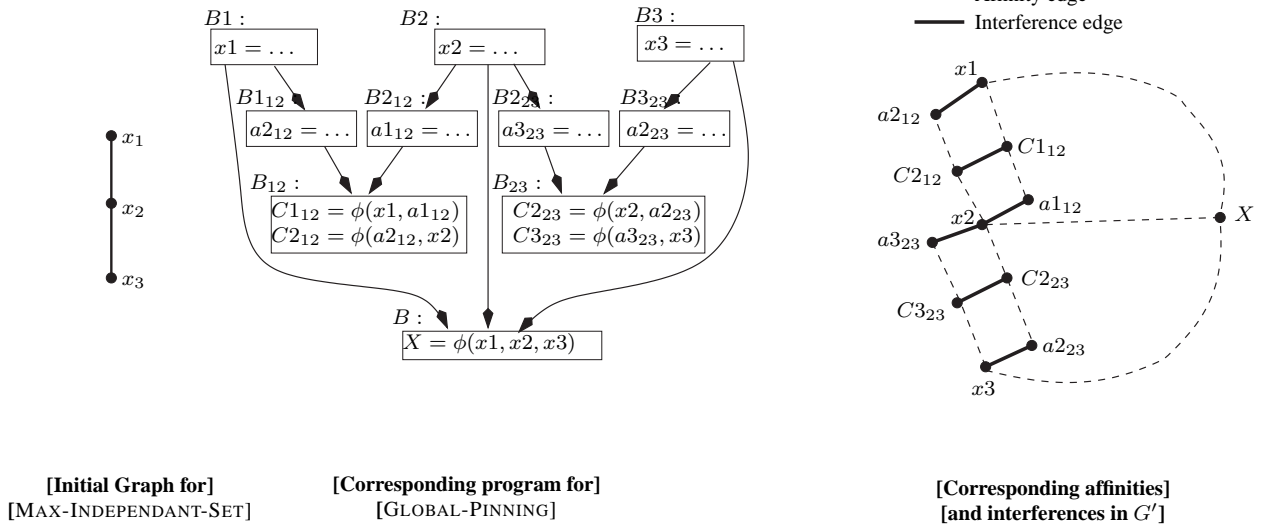


Figure 16: Reduction for the NP-completeness proof of GLOBAL PINNING : G' contains $|E_a| = 4 * |E| + |V| = 11$ affinity edges.

On this program, GLOBAL PINNING can be seen as follow: find a correct coloring of the affinity-interference graph ($G' = (V', E_a, E_i)$) that maximizes the number of affinity edges (dashed edges: E_a) between two vertices of the same color. A coloring is correct whenever there is no interference (full edges: E_i) between two vertices of the same color. To each color corresponds a resource. This new graph G' contains four affinity edges for each edge of the initial graph G ($4|E|$ on the whole), and one affinity edge for each vertex of G ($|V|$ on the whole). See Figure 16 for a simple example. Consider an optimal solution to GLOBAL PINNING with \underline{X} the resource containing X . $I = \underline{X} \cap V$ is the set of vertices of G that have the same color than X . Let $(x_i, x_j) \in E$ and consider the corresponding four affinity edges. As illustrated by Figure 17 if x_i and x_j are in I (so they get the same color), from those four affinity edges only a maximum of two can be satisfied. Hence, the cost of this solution can be bounded by

$$\hat{C}(I) \leq |I| + 4|E| - \sum_{(x_i, x_j) \in E \cap I^2} 2$$

Now, consider a solution where I is an independent set and all other x_i have different colors, then by coloring remaining vertices as illustrated in Figure 17 the upper bound $|I| + 4|E|$ is reached. Hence, the solution is optimal if and only if I is a maximum independent set. ■

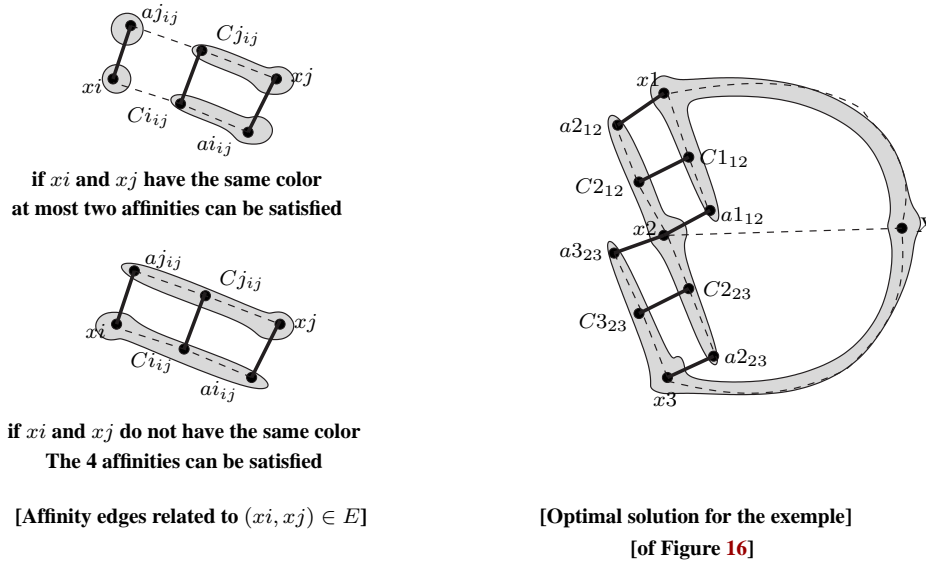


Figure 17: I is an independent set.

Theorem 2 LOCAL PINNING is NP-complete.

Proof of Theorem 2 The proof uses the same reduction than for GLOBAL PINNING : for a given graph $G = (V, E)$ we consider the same program and suppose that blocks B_{ij} have already been performed. Block B remains. At this stage we have $\underline{x_i} = \{C_{ij}, x_i, a_{ij}\}$ for all i and $\underline{X} = \{X\}$. Hence, $(x_i, x_j) \in E$ if and only if $\underline{x_i}$ interfere with $\underline{x_j}$. So, the optimal partitioning of $\{\underline{X}, \underline{x_1}, \dots, \underline{x_n}\}$ provides with $\underline{X} \cap V$ an independent set for G . ■

The MAX-COALESCE problem is NP-complete To finish with, let us define formally the coalescing problem (MAX-COALESCE): consider a program with virtual variables and move instructions between some of those variables. To get a stronger result, consider that this program has been obtained by a basic out-of-SSA translation i.e. no coalescing have been performed during this reconstruction phase. Consider the corresponding interference graph, and affinity graph: two variables interfere² if they cannot be assigned to a common resource; two variables share an affinity if they are move related. Interferences are represented using full edges and affinities are represented using dashed edges. Affinity edges can be weighted to reflect the number of related move instructions and branch frequency prediction (e.g. [14]). To simplify the discussion, we consider that all blocks have an equal execution frequency, but the results given below are correct even if execution frequencies are taken into account.

Now, the coalescing problem consists on coloring correctly this graph using an unbounded number of colors. A correct coloring is a coloring such that two variables with the same color do not interfere. Variables within the same color-set are said to be coalesced. Each move instruction between two coalesced variables can be removed. The goal is to maximize the number of such move instructions, which exactly corresponds to maximizing the weighted number of dashed edges between coalesced variables. As illustrated by Figure 18, the previous NP-completeness proof can be easily modified to prove the NP-completeness of MAX-COALESCE problem.

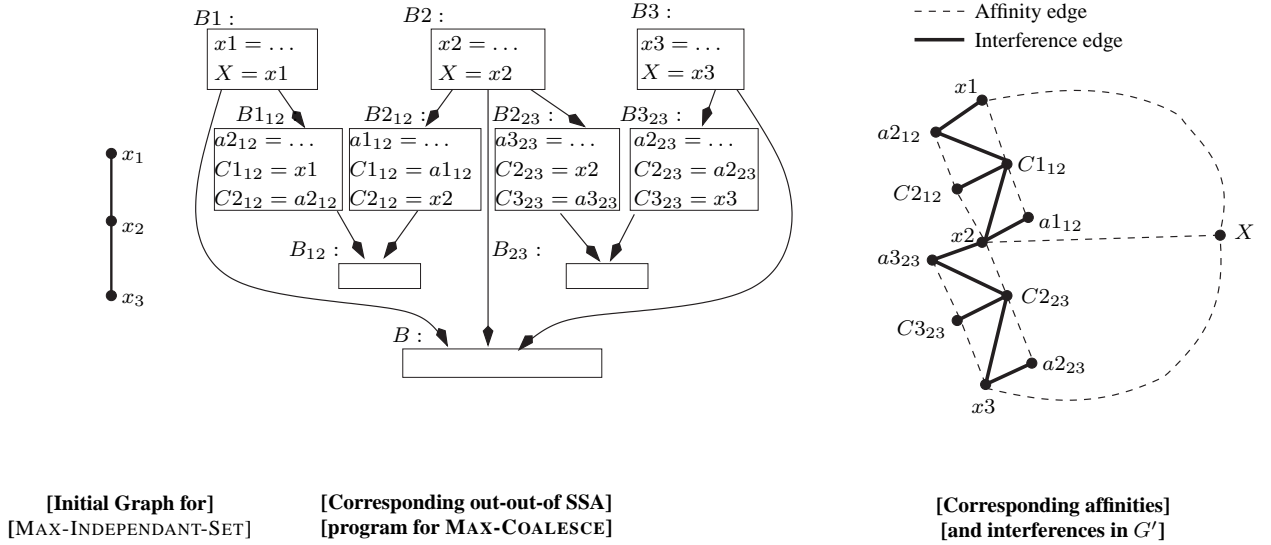


Figure 18: Reduction for the NP-completeness proof of MAX-COALESCE: the corresponding affinity/interference graph is similar to the one obtained for the NP-completeness proof of GLOBAL PINNING .

The MIN-MOVE problem is NP-complete One can wonder if the complexity comes from the out-of-SSA algorithm itself, from the place of move instructions that replace ϕ functions. In particular one can imagine performing code-motion and live-range splitting in order to get an even better result in term of number of move instructions. We name this problem MIN-MOVE. Unfortunately, the previous example with a similar reasoning also leads to the NP-completeness of the MIN-MOVE problem.

²Remark that on our interference model X and $x1$ of Figure 18 do not interfere simply because they share the same value whereas on some classical definitions they do. This can be easily overcome by splitting edges between x_i and X