Optimizing the translation out-of-SSA with renaming constraints

Fabrice Rastello, F Ferrière, C. Guillon

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

Fabrice Rastello, Ferrière, F. De, Christophe Guillon

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

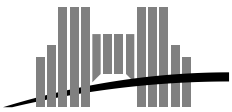
CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Optimizing the translation out-of-SSA with renaming constraints

F. Rastello
F. de Ferrière
C. Guillon

June 2003

Research Report N° 03-35



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

June 2003

Abstract

Static Single Assignment form is an intermediate representation, that uses ϕ-functions to merge values at each confluent points of the control flow graph. ϕ functions are not machine instructions and should be renamed back to `move` operations when translating out-of-SSA form. Without a coalescing algorithm, out-of-SSA translation generates many `move` instructions. In this paper we propose an extension of the algorithm of Leung and George [7] to minimize the ϕ-related copies during the out-of-SSA translation. Leung et al. constructed SSA form for programs represented as native machine instructions, including the use of machine dedicated registers. For this purpose, the out-of-SSA translation contains renaming constraints that are represented using a pinning principle. Pinning the ϕ-function arguments and their corresponding definition to a common resource is a very attractive technique for coalescing variables, even if this is not a true minimization: this article presents a renaming-constraints aware and pinning-based coalescing algorithm. Even without renaming constraints, the `move` instructions minimization problem is still considered an open issue [7, 10]. This article provides also a discussion about the formulation of this problem, its complexity and its motivations. Finally, we implemented our algorithm in the STMicroelectronics Linear Assembly Optimizer [2]. This provides many interesting results when comparing several possible approaches. We also explain, using hand crafted examples, the limitations of Leung's, Sreedhar's and classical register coalescing [6] algorithms.

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 extention de l'algorithme de Leung et George [7] 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 [7, 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 [2], 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 [9] is an intermediate representation, widely used in modern compilers. As opposed to some single assignment representations used on parallel computing [4], the representation we consider here is scalar. We mean by scalar the fact that we allow several consecutive writes to the same memory address. In SSA form, each scalar variable is statically defined only once in a program. Because of this single assignment property, the SSA form contains virtualized registers, and ϕ-functions are introduced to merge different variables that come from the incoming edges at a confluent point of the control flow graph (see Figure 1). Hence the SSA form cannot be used to represent final assembly code and a translation out of SSA should be performed. This transformation replaces ϕ-functions with `move` instructions and part of the virtual registers into dedicated ones when necessary. Apart from the fact that the replacement should be performed carefully whenever optimizations like value numbering has been done while in SSA-form, a naive approach for the out of SSA translation generates a large number of move instructions. This article addresses the problem of optimizing the number of generated copies during this translation phase.

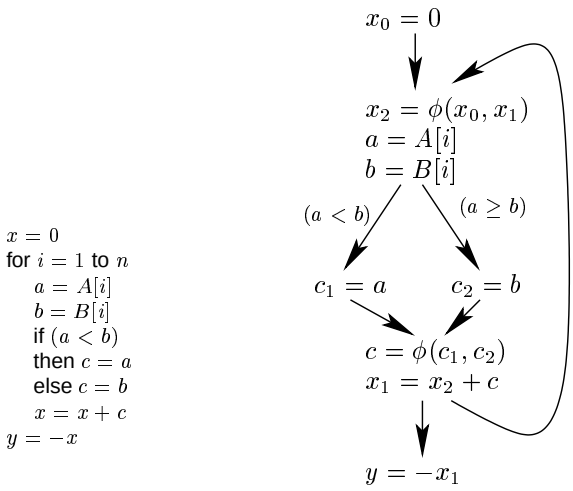


Figure 1: Example of code in non-SSA form and its corresponding SSA form without the loop counter represented

Previous Work Cytron et al. [9] proposed a simple algorithm that replaces a ϕ instruction by copies into the predecessor blocks, and uses Chaitin's coalescing algorithm to reduce the number of copies. Briggs et al. [1] exposed two problems in this algorithm, namely the swap problem and the lost-copy problem, and proposed solutions to these. Sreedhar [13] came with an algorithm that avoids the need for a Chaitin's coalescing algorithm and that can eliminate more copy operations than the previous algorithms. Leung [7] proposed an out of SSA algorithm for a SSA representation at the machine code level. Machine code level representations add naming constraints due to ABI rules on calls, special purpose ABI defined registers, or restrictions imposed on register operands.

Context of the study Our study of an out-of-SSA algorithm was done in the STMicroelectronics Linear Assembly Optimizer (LAO) tool. The purpose of the LAO is to convert a program written in Linear Assembly Input (LAI) language to the basic assembly language that is suitable for assembly, linking and execution. The LAI language is a superset of the assembly language where symbolic register names can be freely used. The LAO will perform register allocation and instruction scheduling on LAI code. It also includes a number of transformations such as induction variables optimizations, global value numbering and optimizations based on range propagation, and uses an SSA intermediate representation to perform most of its optimizations.

The LAO implements scheduling techniques based on software pipelining and superblock scheduling, and uses a *repeated coalescing* [2] register allocator, which is an improvement over the *iterated register coalescing* from George & Appel [6].

The LAO tool is targeted at 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.

Layout of this paper This paper contains five parts: we start by a statement of the problem and a brief description of Leung's algorithm on which our solution is based. Then, we present our solution to the problem of variables coalescing during the out-of-SSA phase. Next, we discuss in several theoretical examples how our algorithm compares to others. Then, we present results that show the effectiveness of our solution on a set of benchmarks to 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 Leung's algorithm and statement of the problem

2.1 Pinning mechanism

Renaming constraints are, in Leung's algorithm, represented using a pinning mechanism. Because the pinning phase described in [7] is restricted to physical registers and because the base-register [1] constraint used can lead to an incorrect pinning (see Figure 14), this paragraph describes our refined version of pinning. Roughly speaking, pinning is a pre-coloring of SSA variables to resources. Pinning can be described using the three following properties:

1. **[Resources]** A program in SSA-form contains only SSA-variables. A resource is a target name for the renaming of SSA-variables. *Resources include physical registers in addition to virtual registers.*
2. **[Operand pinning]** A pinning makes reference to a particular instance of a variable in the program: it can be on the unique definition of a variable or on one of its uses. It means that *for this instance*, the SSA variable must be renamed to the resource.
3. **[Variable pinning]** When a variable is said to be pinned to a resource, we implicitly refer to *the instance of the unique definition of the variable.*

ABI and 2-operand constraints First of all, pinning comes from the Application Binary Interface (ABI) and from the Instruction Set Architecture (ISA) constraints. We have extended the pinning mechanism to the coalescing of variables defined or used on ϕ instructions. This extension is developed further.

In the st120 processor, we are concerned with ISA register renaming constraints and ABI function parameter passing rules. Program 1 expressed in pseudo assembly code provides an example of such constraints. The corresponding SSA representation and its corresponding pinning, after copy folding and dead code, is given in Program 2. In this example and in the rest of this document, the notation $X{\uparrow}^R$ is used to mark that a given instance of a SSA variable X is pinned to a resource R.

This code contains two kinds of constraints:

- *function parameter passing rules*: for example the use of variable A_0 in function call f should be renamed in R0. Identically, the use of F_0 in the .output should be renamed in $R0$.
- *2-operand instruction constraints* (this is how Leung et al. call this kind of constraints): here, because `autoadd` P_1`,` $P0_0$`,` `1` is an auto-increment instruction, the Instruction Set Architecture (ISA) of the target machine imposes that P_1 and $P0_0$ should be renamed to the same resource. Instruction `more` has similar constraint.

[1] the base-register of a variable is the physical register this variable was renamed from during SSA construction

Program 1 Example of code with ABI constraints

```
// performs f(@(P0),@(P0+1))+R0-0x00A12BFA
.input          R0, P0
// inputs C & P are necessarily on R0 & P0 at the entry
    move        C, R0
    move        P, P0

    load        A, @P++
    load        B, @P++
// calling D=f(A,B)
    move        R0, A
    move        R1, B
    call        f
    move        D, R0
// E=C+D
    add         E, C, D
// K=0x00A12BFA
    make        K,0x00A1
    more        K,0x2BFA
// F=E-K
    sub         F,E,K
// returned F should be on R0
    move        R0, F
.output         R0
```

Program 2 SSA version of Program 1 and its corresponding pinning after copy-folding

```
.input                  R0_0↑R0, P0_0↑P0
OP_1    load            A_0, @P0_0↑P0
        autoadd         P_1↑P_1, P0_0↑P_1, 1
OP_2    load            B_0, @P_1
OP_3    __f             R0_1↑R0, A_0↑R0, B_0↑R1
OP_4    add             E_0, R0_0, R0_1
OP_5    make            K_0, 0x00A1
OP_6    more            K_1↑K_1, K_0↑K_1, 0x2BFA
OP_7    sub             F_0, E_0, K_1
.output                 F_0↑R0
```

2.2 Leung's algorithm

Leung's algorithm is decomposed into three consecutive phases:

1. *the collect phase* where information about renaming constraints is collected: operands are pinned during this phase.
2. *the mark phase* where information about conflicts generated by renaming is collected.
3. *the reconstruct phase* where renaming is performed and copies are inserted to repair values killed by local renaming or by replacement of ϕ functions.

Our pinning based coalescing phase takes place during the collect phase. Then, for a given pinning, the out of SSA translation relies on the mark and reconstruct phases of Leung's algorithm. This paragraph aims to illustrate the kind of transformations that are performed during those two last phases. For this purpose, an example of pinned SSA code and the resulting out of SSA code is given Figure 2.

From this example, we can make the following remarks:

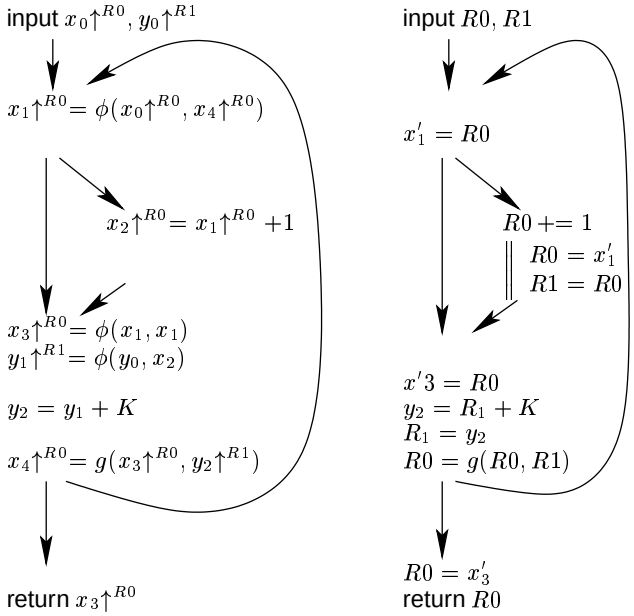


Figure 2: Transformation of already pinned SSA code by Leung's algorithm

- A repair copy is introduced when a pinned variable is killed before its use. This is the case around the call to the function g, where x3 is killed by x4, before being used on the return instruction.
- The algorithm is careful not to introduce redundant copy instructions when a value is already available in the resource it is pinned to. This is shown on the example on the use of x3 in the call to g, where x3 is already available in R0 due to pinning on the ϕ instruction.
- Parallel copies are used to avoid the so-called swap-copy problem. In the example, the copies $R0 = x'_1$; $R1 = R0$ are to be performed in parallel. In sequential code, the copies will be reordered and an intermediate variable will be introduced when necessary. (In this example, the corresponding sequence will be $R1 = R0$; $R0 = x'_1$).
- The main limitation of Leung's algorithm is its inability to coalesce a non-constrained definition and a resource to which a used operand is pinned. As an example, y_2 could have been coalesced to R_1 without creating any interference. As illustrated by Figure 4, this weakness is similar to the ϕ-function replacement coalescing problem where use operands of a ϕ are implicitly pinned to the def operand.

2.3 Correct pinning

As illustrated in Appendix A, renaming rules that are too constrained can lead to conflicting pinning in addition to incorrect pinning. The semantics of a correct pinning is best explained using examples given in Figure 3.

In this figure, (OP_1) and (OP_2) are correct if and only if $a = b$. This is because, two different values can not be contained in a unique resource both at the entry point and at the return point of an instruction. (OP_3) is the special case on ϕ instructions: because the semantics of the set of ϕ instructions of a block is parallel, two different ϕ definitions in a same block cannot be pinned to the same resource.

On the other hand, on most architectures, (OP_4) is a correct pinning. But, the corresponding scheme (OP_5) on a ϕ instruction is forbidden: this is because all use operands of a ϕ instruction are implicitly pinned to the resource the def operand is pinned to[2].

[2]The motivation for this semantic is given in Appendix A

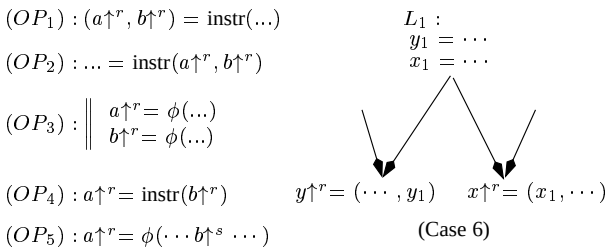


Figure 3: All but (OP_4) are incorrect pinning

Hence, a more subtle incorrect pinning is given by (Case 6): the use operands x_1 and y_1 are operands of a same instruction and the corresponding parallel copies takes place at the end of block L_1. This incorrect pinning would generate the meaningless parallel copies $\left\| \begin{array}{l} r \leftarrow x_1 \\ r \leftarrow y_1 \end{array} \right.$.

Defining precisely what kind of pinning is semantically correct might seems intuitive and unnecessary. But as it is explained in Appendix A, for some dedicated registers like SP (Stack Pointer) for the st120, we might want to rename-back the corresponding variables to their base-registers (as it was *before* translating *to* SSA form). This, added to optimizations like value numbering, can lead to an incorrect pinning. Hence, when dealing with dedicated-register constraints, optimizations should be aware of maintaining a semantically correct SSA code, which is not necessarily trivial since the correctness cannot always be checked locally. For example our value numbering optimization is allowed to coalesce two expressions containing SP based variables only if the defined variables are not SP based.

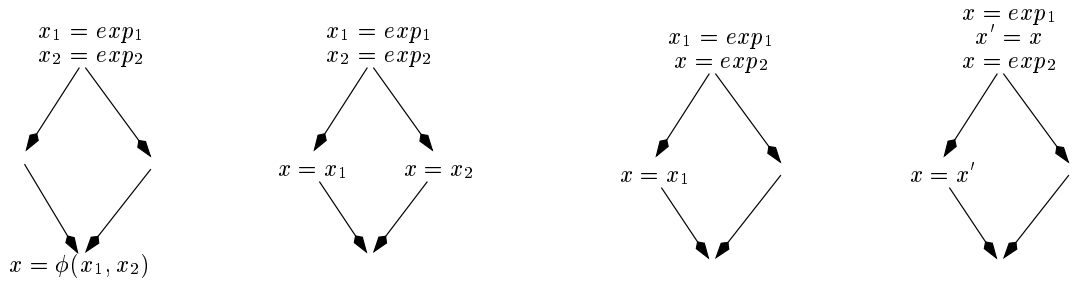
2.4 ϕ coalescing: statement of the problem

As illustrated by Figure 4, pinning potentially prevents the reconstruction phase of Leung's algorithm from inserting useless copies. In this sense, pinning plays the role of a coalescing phase.

Remark that, as opposed to the pinning related to ABI constraints that is applied to a particular instance of a SSA-variable, the copy optimization pinning is applied only to SSA-variable definitions. Hence, *we extensively say that a SSA-variable is pinned to a given resource* and define Resource_def(y) as

$$\begin{cases} r & \text{if the definition of } y \text{ is pinned to } r \\ y & \text{otherwise} \end{cases}$$

Also, for simplicity *we mingle the notions of resource itself and set of variables pinned to it.*



Initial SSA form

Code translated out-of-SSA without pinning

Code translated out-of-SSA if x and x_2 are pinned to the virtual resource x

Code translated out-of-SSA if x, x_1 and x_2 are pinned to the virtual resource x

Figure 4: Usefulness of pinning the definitions of ϕ arguments to a common resource and its counterpart when variables interfere

On the other hand, as illustrated by Figure 4, the pinning of two variables which creates an interference will generate an additional copy (a repair copy) just after the definition of the killed variable. Moreover, if this interference

precedes a ϕ-function where these two variables are used, the replacement of this ϕ-function will still generate the copy that motivated the pinning.

Hence, the ϕ-function coalescing problem consists in pinning, for each ϕ, the maximum number of used variables to a common resource while preserving the number of variables that are concerned with a repair. Since the cost of pinning a resource that is concerned with a repair (at least one copy just after the definition plus the restore copy) is *generally* greater than the gain obtained by this pinning (potentially no copies inserted while replacing ϕ-functions), *the problem is to maximize the number of ϕ-function arguments pinned to a common resource while not modifying the current number of killed variables.* Our formal formulation of the ϕ function coalescing problem is GLOBAL PINNING given in Appendix B.

3 Our solution

Our algorithm (Program_Pinning), an heuristic solution to LOCAL PINNING, is based on an inner to outer loop traversal. Each node is treated locally but within each node all ϕ functions are treated together. Hence, for a given node, an affinity graph is created (Initial_G). Every edge that is concerned with an interference is removed from the graph (Graph_InitialPruning). The remaining graph is bipartite with possible interferences between nodes on the same side (PrePruned_G). Vertices are weighted to take into account interferences between SSA variables. Then the graph is pruned (BinaryGraph_pruning) and the elements of each resulting connected components (Final_G) are pinned to the same resource (PrunedGraph_pinning).

The rest of this section is devoted to the precise description of this algorithm and provides a formal definition of interferences between resources. Consecutive steps of this algorithm are applied on the example of Figure 5.

Algorithm 1 ϕ coalesce, using pinning method, all nodes of P using a decreasing depth order traversal.

```
Program_pinning(CFG_Program P)
foreach N downdepth {Nodes of P}
    Initial_G=Create_affinity_graph(N)
    PrePruned_G=Graph_InitialPruning(Initial_G)
    Final_G=BinaryGraph_pruning(PrePruned_G)
    PrunedGraph_pinning(Final_G)
```

3.1 The initial affinity graph

The affinity graph is an undirected graph where each vertex represents either a variable or its corresponding resource: two variables that are pinned to the same resource are collapsed into the same vertex. For each definition $X = \phi(x_1, \ldots, x_n)$ there is an affinity edge between the vertex of X and the vertices of all x_i. The construction of the initial affinity graph is described in Algorithm 2.

Algorithm 2 Construction of initial affinity graph

```
Create_affinity_graph(CFG_Node current_node)
(E, V) = (∅, ∅)
for each X = φ(x_1, ..., x_n) of current_node
    V = V ⋃ {Resource_def(X)}
    for each x ∈ {x_1, ..., x_n}
        V = V ⋃ {Resource_def(x)}
        e = (Resource_def(X), Resource_def(x))
        if (e ∉ E) multiplicity(e)=0
        E = E ⋃ {e}, multiplicity(e)++
return G = (E, V)
```

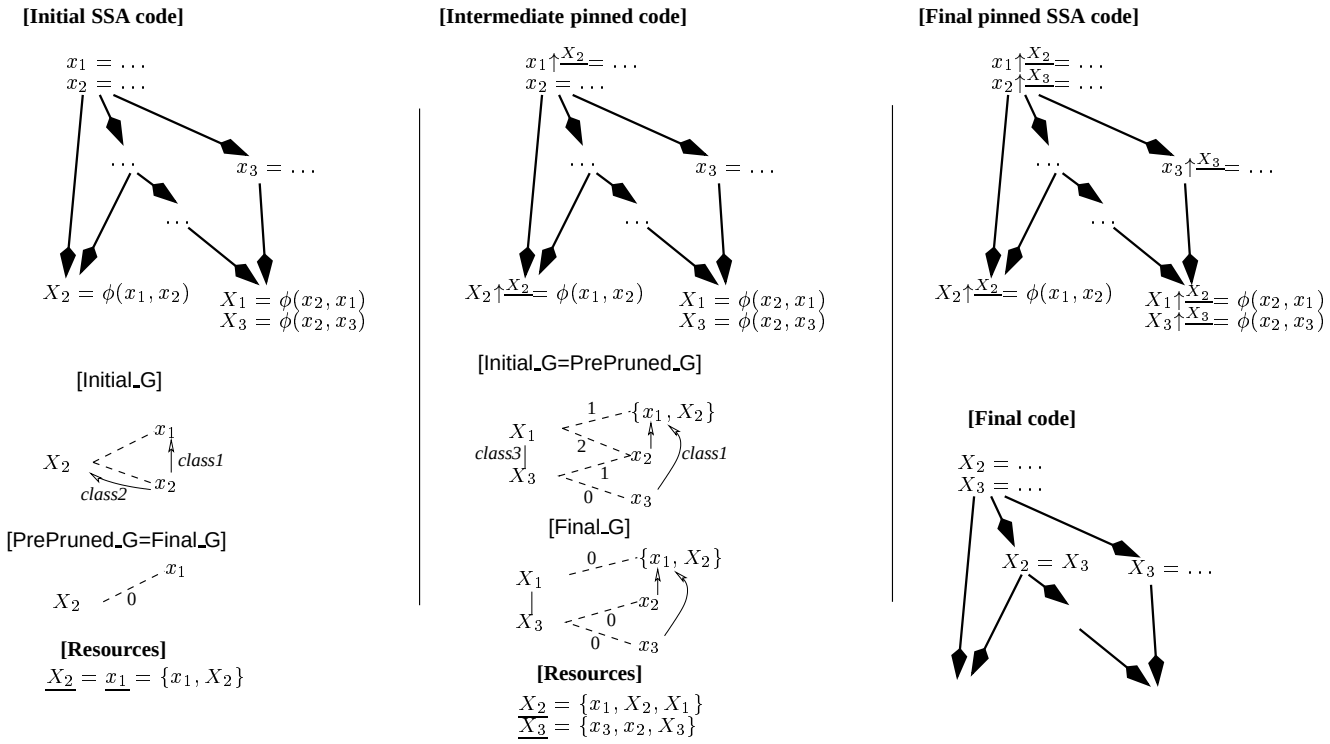


Figure 5: Program_pinning on an example. Affinity and interference edges are respectively represented using dashed and full lines.

3.2 Interferences between variables

We enumerate below the cases where pinning two variables of a ϕ-function to the same resource would create an interference. We differentiate simple interferences from strong interferences: a strong interference generates a conflict that violates the pinning semantics in such a way that it cannot be repaired by the reconstruct phase. 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.

[Case 1] Consider two variables a and b. If there exists a point in the Control-Flow graph where both a and b are alive, then a and b interfere. Moreover, considering the definitions of a and b, one dominates the other. If the definition of a dominates those of b, we say that the *definition of a is killed by b*. The consequence is that pinning the definitions of a and b to a common resource would result in a repair of a.

[Case 2] Consider a (from block B) and b, where a is defined by a ϕ-function $a = \phi(a_1, \ldots, a_n)$ (from block $B_1, \ldots, B_n$). Then each argument a_i is implicitly pinned to a at the point where the use actually occurs, which is at the end of node B_i. Hence, if $b \neq a_i$ and b is live-out of B_i, b and the use of a_i interfere. We say that *the definition of b is killed by a.*

Remark that our definition of liveness is *different from the definition used by Sreedhar et al.*: a ϕ instruction does not occur where it textually appears, but on each predecessor basic blocks instead. Hence, if not used by another instruction, a_i is treated as dead upon the end of block B_i and the entry of block B.

[Case 3] Consider two variables x and y, both defined by ϕ functions. Their respective arguments might interfere in a common predecessor node. Hence, if predecessor B_i associated to x_i is equal to predecessor B_j associated to y_j and $x_i \neq y_j$ we say that *the definitions of x and y strongly interfere*: indeed, as explained in Paragraph 2.3, pinning those two definitions together is incorrect.

[Case 4] Consider two ϕ definitions x and y within the same block. Because of Leung's repairing implementation, two definitions with the same arguments $x = \phi(z_1, \ldots, z_n)$ and $y = \phi(z_1, \ldots, z_n)$ cannot be considered as identical. Hence, they should always strongly interfere.

The different cases are illustrated in Figure 6. The corresponding algorithm is described in Algorithm 3 and Algorithm 4.

Algorithm 3 Evaluates if two variables simply interfere. Returns true if a kills b.

```
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 = φ(a_1 : B_1, ..., a_n : B_n)
    for i = 1 to n
        if b is live out of B_i and b ≠ a_i
            return true {Case 2}
return false
```

Algorithm 4 Evaluates if two variables strongly interfere. Should be adapted to the specificities of the ISA.

```
Variable_stronglyInterfere(Variable a, Variable b)
if a and b are defined by φ-functions
    let a : B_a = φ(a_1 : B_{a,1}, ..., a_n : B_{a,n})
    let b : B_b = φ(b_1 : B_{b,1}, ..., 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 ≠ b_j return true {Case 3}
    return false
else if a and b are defined in the same instruction
        let (··· a ··· b ···) = instr(···)
        return true
return false
```

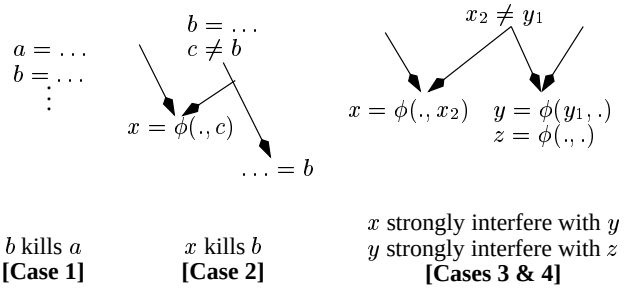


Figure 6: Different kind of interferences between variables.

3.3 Interferences between resources

An interference between two resources $\underline{A} = \{a_1, \ldots, a_n\}$ and $\underline{B} = \{b_1, \ldots, b_m\}$ means that pinning all the variables $\{a_1, \ldots, a_n\}$ and $\{b_1, \ldots, b_m\}$ together would create either a *new* simple interference, or *any* strong interference.

The result is given by Resource_interfere (Algorithm 6). It uses Resource_killed (Algorithm 5 gives a formal description, but obviously this information can be maintained and updated after each merge) that returns variables already killed within a resource set (remark that for the lost-copy problem a variable is killed by itself).

Algorithm 5 Returns the set of variables killed within $\underline{A}$.

```
Resource_killed(Resource A)
let A = {a_1, ..., a_n}
killed_withinA =
    {a_i ∈ A | ∃a_j ∈ A, Variable_kills(a_j, a_i)}
return killed_withinA
```

Algorithm 6 Interference between resources.

```
Resource_interfere(Resource A, Resource B)
let A = {a_1, ..., a_n}
let B = {b_1, ..., b_m}
let killed_withinA = Resource_killed(A)
let killed_withinB = Resource_killed(B)
if A and B are physical resources
    if A ≠ B return true
for all (a, b) ∈ A × B
    if a ∉ killed_withinA and Variable_kills(b, a)
        return true
    if b ∉ killed_withinB and Variable_kills(a, b)
        return true
    if Variable_stronglyInterfere(a, b)
        return true
return false
```

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 $G = (E_{Affinity}, V)$ be the graph obtained from Create_affinity_graph: V is the set of vertices labeled by resources and $E_{Affinity}$ is the set of affinity edges between vertices.
- Let $(E_{Interference}, V)$ be the graph of interferences as defined by Resource_interfere.
- The goal is to prune (edge deletion) the graph G into $G' = (E_{pinned}, V)$ such that

  condition 1 the cardinality of E_{pinned} is maximum

  condition 2 for each *connected* variables $(v_1, v_2) \in V^2$ of G', $(v_1, v_2) \notin E_{Interference}$

  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 there is no edge from $E_{Interference}$ within each connected component.

Trivially, because of **condition 2**, all edges from $E_{Affinity} \bigcap E_{Interference}$ can be removed from G (Algorithm 7). The obtained graph PrePruned_G is bipartite: indeed, consider the set of ϕ-functions of current node $\{X_i = \phi(x_{i,1}, \ldots, x_{i,n}), 1 \leq i \leq$ There are two kinds of vertex in G, the one from the definitions $V_{DEFS} =$ Resource_def$(\{X_1, \ldots, X_m\})$ and the others $V_{ARGS} =$ Resource_def$(\{x_{1,1}, \ldots, x_{m,n}\}) \setminus V_{DEFS}$. By construction there is no 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}.

As explained in Appendix B, the pruning phase is NP-complete in the size of ϕ-functions. Our algorithm is an heuristic based on a greedy pruning of edges. In the particular case of a unique ϕ function, it is identical to the "`Process the unresolved resources`" of Sreedhar's algorithm. For a binary affinity graph (with the trivial pruning already performed), the algorithm is given by the BinaryGraph_pruning procedure (Algorithm 8).

Algorithm 7 Initial pruning. We obtain a bipartite graph.

```
Graph_InitialPruning(Graph (V, E))
foreach (x1, x2) ∈ E,
    if (Resource_interfere(x1, x2))
        E −= (x1, x2)
return (V, E)
```

Algorithm 8 Prunes (E, V) such that: (1) foreach connected vertices $(\underline{x_1}, \underline{x_2}) \in E^2$, Resource_interfere$(\underline{x_1}, \underline{x_2})$ =false; (2) the remaining V has a "maximal" cardinality.

```
BinaryGraph_pruning(Binary_Multi_Graph (V, E))
{ Evaluates the weight for each edge }
for all e ∈ E, weight(e)=0
for all ((x, x1), (x, x2)) ∈ E^2 such that x1 ≠ x2
    if Resource_interfere(x1, x2)
        weight((x, x1))+=multiplicity((x, x2))
        weight((x, x2))+=multiplicity((x, x1))

{Prunes in decreasing weight order
  and update the weight}
while weight(e_p)> 0
let e_p = (X, x) such that
    ∀e ∈ E, weight(e_p)≥weight(e)
do
    E −= e_p
    for all e = (X, y) ∈ E
        weight(e)−=multiplicity(e_p)
    for all e = (Y, x) ∈ E
        weight(e)−=multiplicity(e_p)

return (V, E)
```

3.5 Merging the connected components

Once the affinity graph has been pruned, resources of each connected components can be merged. To merge one resource into another, all operands of the program pinned to the first resource should be pinned to the second one. The correctness of this phase is insured by the absence of any strong interference inside the new merged resource. To complete the coalescing of ϕ-functions, the definition of each variable in a resource should be pinned to this resource. A formal description of the algorithm is given by the procedure PrunedGraph_pinning (Algorithm 9). In practice, the update of pinning can be performed only once, just before the mark phase, so requiring only one traversal of the control flow graph. Another remark is that the interference graph can be built incrementally at each call to Resource_interfere and updated at each resource merge, using a simple vertices merge operation: hence, as opposed to the merge operation used in the repeated register coalescing algorithm where interferences have to be recomputed at each iteration, here each vertex represents a SSA variable and merging is a simple edge union.

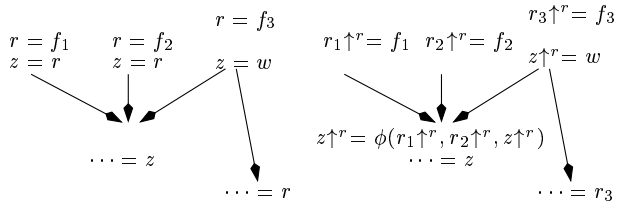
4 Theoretical discussion

4.1 Our algorithm vs register coalescing

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

Algorithm 9 Merge connected components and pin definitions when necessary

PrunedGraph_pinning(Graph G, Program P)
foreach $V \in$ {connected components of G}
  let $\underline{u} = \bigcup_{\underline{v} \in V} \underline{v}$
  let $\underline{w} = \begin{cases} \underline{v_i} & \text{if } \underline{v_i} \in V \text{ is a physical resource} \\ \underline{u} & \text{otherwise} \end{cases}$
  foreach $(OP)\, d_1, \cdots = instr(a_1, \ldots) \in P$
    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 $\underline{w}$

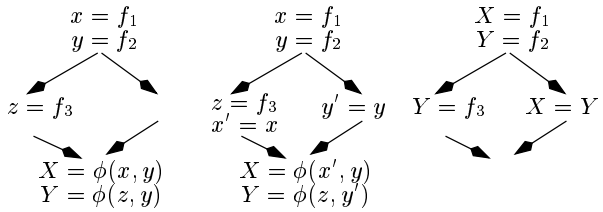


[Initial code] **[Pined SSA form]**

Figure 7: Partial coalescing: the left hand-side code cannot be coalesced by a classical coalescer, whereas the corresponding pinned SSA code will result in less move instructions. In this case, the only copy will be generated by a repair of r_3.

- SSA form is a higher level representation that allows a more accurate definition of interferences. For example, as illustrated by Figure 7, it allows partial coalescing. By partial coalescing, we mean coalescing of a subset of a variable's definitions.
- Classical coalescing algorithm is greedy, so it may block further coalescings. Instead, our algorithm locally considers a set of possible coalescing and optimizes the number of coalesced variables.
- The main motivation of Leung's algorithm is that ABI constraints introduce another bunch of copy instructions. Some of these will be deleted by a dead code algorithm, but most of them will have to be coalesced. The goal of our work is to reduce the overall complexity of the out of SSA renaming and coalescing phases. Compared to the use of a repeated coalescer, we reduce the complexity of the coalescing phase by doing it on the SSA representation, thus benefiting from the static single definition property.

4.2 Our algorithm vs Sreedhar



[Initial SSA] **[CSSA after Sreedhar]** **[Our solution]**

Figure 8: Comparison on example7: our solution optimizes all ϕ-functions together whereas Sreedhar treats them one after the other.

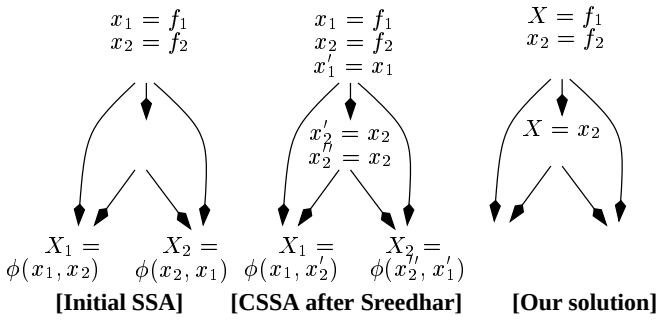


Figure 9: Comparison on example6: Sreedhar's algorithm generates copies that cannot be reused in the rest of the process.

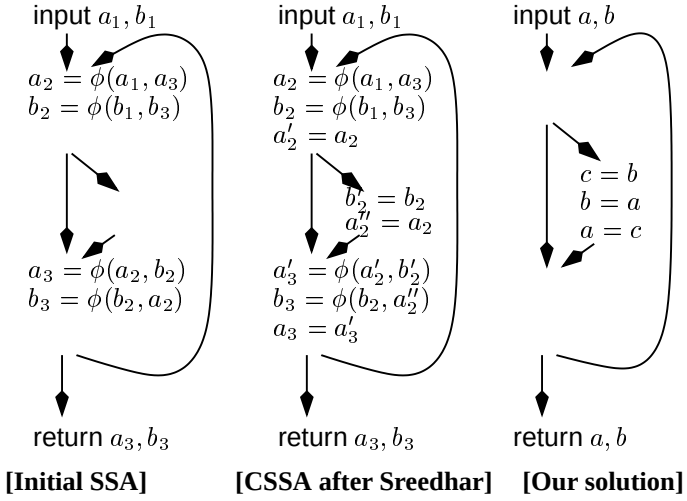


Figure 10: Comparison on example8: the superiority of using parallel copies.

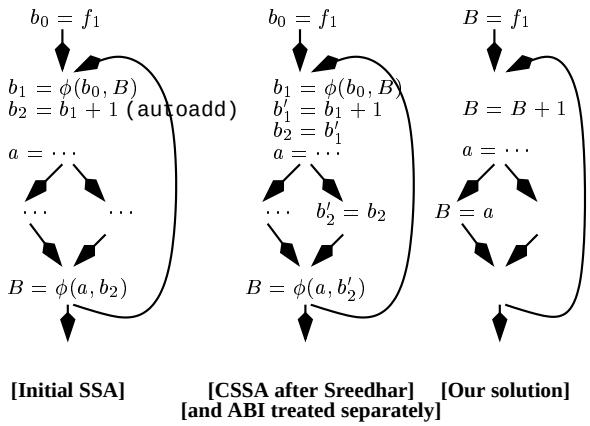


Figure 11: Comparison on exampleABI: if ABI is treated after ϕ-function translation, even optimally, the result is generally worth.

Here we compare our approach against the algorithm from Sreedhar et al. in [13]. Sreedhar's algorithm uses what is called a *Conventional SSA* (CSSA) form, which has the following important property: *all variables which appear in a same ϕ-function can be replaced by a representative variable.* In other words, in the translation out of CSSA form, all occurences of a variable that appears as an argument of a ϕ-function are renamed into the definition of that ϕ-function, and the ϕ-function is discarded. In the examples Figure 8-11, appart from renaming and removal of ϕ functions, no other instruction is added or removed.

The translation from a general SSA form to a CSSA form may create new variables and insert copy instructions to eliminate ϕ variable interferences that would otherwise result in an incorrect program after renaming. Sreedhar proposes three algorithms to convert to CSSA form. We only consider the last one which uses the interference graph and liveness information to minimize the number of generated copy instructions. Below are cases where his and our algorithms produces different results.

- Our algorithm considers the ϕ-functions for a whole block together in order to compute pinning. Figure 8 shows a case where this results in one less copy instruction than Sreedhar's algorithm which consider each ϕ-functions one after the other.
- Our algorithm performs pinning using interferences information on an unchanged SSA form, whereas Sreedhar's process creates different variables that may hold the same value. Figure 9 shows an example where this process yields to a situation where interferences cannot be removed.
- In addition to the two points above, Figure 10 illustrates the advantage we have by generating move instructions after the pinning phase is complete. This example also demonstrates the use of parallel copies.
- Finally, because our SSA representation is at machine level, we need to take into account ABI constraints. Figure 11 shows an example where a better choice of which variables to coalesce together can be taken if the ABI constraints are considered.

4.3 Limitations

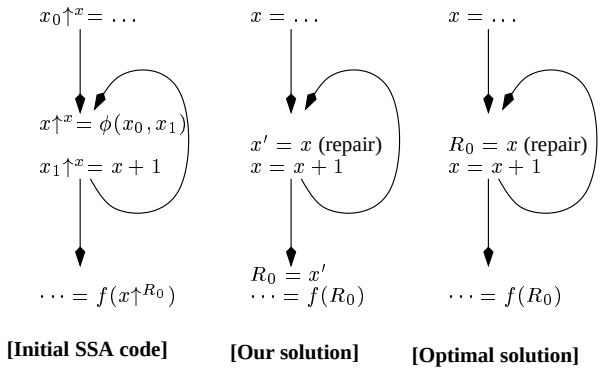


Figure 12: Limitation of Leung's repairing process: the repairing variable x' is not coalesced with further uses.

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

- Our algorithm is based on Leung's algorithm that imposes the place where copy 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 copy instructions.
- As explained in section 2.2, the main limitation of Leung's algorithm is its incapacity to coalesce a non-constrained definition and a resource to which a used operand is pinned. This can be worked-around using a prepass pinning of concerned variable definitions. But as illustrated by Figure 12, repairing variables that are introduced during Leung repairing phase cannot be handled this way.

- Another point is, as explained in Appendix B, that our formulation of the problem is NP-complete. Note also that a simple extension of the proof shows the NP-completeness of the number of move instructions minimization problem.
- The last point is that in 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. [8]).

Because our algorithm relies on the mark and reconstruct phases of Leung's algorithm, a refinement of this algorithm is provided in Appendix A.

5 Results

We conducted our experiments on different sets of benchmarks represented in LAI code. Since the LAI language supports predicated instructions, the LAO tool uses a special form of the SSA representation, named ψ-SSA [12], which introduces ψ-functions to represent predicated code under SSA. Without going into the details of the out-of-ψ-SSA algorithm, we can say that ψ-functions introduce constraints similar to 2-operands constraints, and are handled in our algorithm in a special pass where they are converted into a "ψ-conform" SSA form.

In the following tables *VALcc1* and *VALcc2* refer to the same set of C functions compiled into LAI code with two different ST120 C compilers. These sets include 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. The benches *example7* and *example8* are presented respectively in Figure 8 and Figure 10. The bench *example6* is a variant of Figure 9, since a basic block with three successors cannot be expressed in the LAI language. *LAI_Large* is a set of larger functions, most of which come from the efr 5.1.0 vocoder from the ETSI [3]. Finally, *SPECint* refers to the SPEC CINT2000 benchmark [11].

benches	U+C	B+C	S+C
VALcc1	193	+59	+3
VALcc2	170	+44	+13
example1	1	+0	+0
example2	1	+0	+0
example3	3	+0	+0
example4	1	+0	+0
example5	3	+0	+0
example6	1	+1	+1
example7	1	+0	+1
example8	3	+2	+1
LAI_Large	438	+44	+48
SPECint	6803	+3135	-59

Table 1: Comparison of copy instruction count with no ABI constraint.

In order to perform experiments with Sreedhar's algorithm (S+C, S+L+C and S+A), we have implemented a modified version of this algorithm to include support for low level SSA representation with ABI constraints. This version is correct in most cases, but it still performs some illegal variables splitting on some code. This results in missed interferences, and the final code contains less copy instructions and is incorrect. Such cases mainly occurred with SPECint, and thus SPECint figures after Sreedhar's algorithm must be taken only as an optimistic approximation of the number of copy instruction.

Comparison without ABI-constraints Table 1 shows the variation in number of copy instructions of SSA rename back coalescing algorithms based on Briggs or Sreedhar compared to our algorithm, when ABI renaming constraints are ignored. In this table, U+C, B+C and S+C corresponds respectively to our algorithm (U), Briggs' algorithm (B) and Sreedhar's (S) algorithm, all followed by a repeated register coalescing (+C). In this experiment, our algorithm is

better or equal in all cases, except for the SPECint benchmark with Sreedhar's algorithm. However, as explained above, dedicated register constraints and predicated definitions cannot be trivially ignored and repaired in a further pass, and our implementation of Sreedhar's algorithm produces incorrect code in some cases. In absence of ABI constraints, this experiment is interesting on two points. One point is the interest of optimizing the coalescing on each ϕ-functions instead of running a greedy coalescing on each move. This is the comparison between the optimizing algorithm from Sreedhar (S+C) and the simple algorithm from Briggs (B+C). The other point is the interest of performing the coalescing and the repairing phases separately instead of doing these two operations for each ϕ-functions together. This is the comparison between our algorithm (U+C) and the algorithm from Sreedhar (S+C).

benches	U+C	S+L+C	L+C	B+A+C
VALcc1	242	+7	+3	+386
VALcc2	220	+15	+29	+449
example1	1	+0	+0	+2
example2	1	+0	+0	+2
example3	3	+0	+0	+2
example4	2	+0	+0	+1
example5	3	+0	+0	+2
example6	1	+1	+1	+3
example7	1	+1	+0	+2
example8	3	+1	+2	+4
LAI_Large	1085	+26	+62	+634
SPECint	23930	+413	+482	+38623

Table 2: Comparison of copy instruction count with ABI constraints on all benches.

Comparison with renaming constraints Table 2 shows the variation in number of copy instructions of various SSA rename back coalescing algorithms, with ABI renaming constraints. In this table, (U) corresponds to our algorithm i.e. ϕ renaming and ABI constraints treated together. (S+L) corresponds to Sreedhar followed by Leung, i.e. ϕ renaming optimization separated from the treatment of ABI constraints. (L) corresponds to Leung alone i.e. the ϕ coalescing is left to the repeated register coalescing (C). Finally, (B+A) corresponds to Briggs followed by a naive treatment of ABI constraints i.e. all the coalescing is left to (C). Our algorithm performs better in all cases. In second position comes the Sreedhar + Leung algorithm, i.e. phi and ABI optimized renaming performed separately. However, as already mentioned, our modifications to Sreedhar's algorithm do not handle ABI constraints correctly in all cases, resulting in optimistic figures in our experiments. The results show the interest of treating ABI constraints and ϕ-function coalescing together.

Compilation time As explained in Section 3.5, our solution is better than a simple repeated register coalescing in terms of time and space complexity because interference graph in SSA form is simplified (see [5] for more details). Table 3 gives an evaluation of the number of copy instructions that would remain after the SSA rename back phase if only naive techniques would be applied for the ϕ-function replacement or ABI constraints. In addition, it gives an evaluation of the cost of running a repeated register coalescer after one or the other 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

benches	U	S+A ABI	L PHI
VALcc1	277	+593	+690
VALcc2	245	+926	+749
example1-8	16	+38	+34
LAI_Large	1447	+4543	+6161
SPECint	36882	+249481	+260095

Table 3: Order of magnitude.

benches	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 4: Weighted count of move instructions on test versions.

because our implementation is too experimental and not optimized enough to give usable results. In this table, (S+A) represents the number of move instructions introduced by a naive correction of ABI constraints, (L) the number of move instructions introduced by a naive replacement of ϕ-functions, all relatively to our algorithm (U).

Variations on our algorithm Table 4 compares small variations in the implementation of the algorithm. This table 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.

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 12) with a depth constraint which calls Program_pinning with decreasing depth. This results in a very short improvement on SPECint and a small worsening on 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 fuzzy definitions of interferences, respectively optimistic (Algorithm 10) and pessimistic (Algorithm 11). It is interesting to note that optimistic interferences only incur a relatively small increase in number of move while reducing significantly the complexity of the computation of the interference graph.

Algorithm 10 Optimistic definition of interferences

```
Variable_kills_optimistic(Variable a, Variable b)
let Node_a: (Def_a) a = ...
let Node_b: (Def_b) b = ...
if (a ≠ b) and (Def_b dominates Def_a) and
   (b ∈ liveout(Node_a))
    return true {Case 1}
if a is defined as a = φ(a_1 : B_1, ..., a_n : B_n)
    for i = 1 to n
       if b is live out of B_i and b ≠ a_i
          return true {Case 2}
return false
```

Algorithm 11 Pessimistic definition of interferences

```
Variable_kills_pessimistic(Variable a, Variable b)
let Node_a: (Def_a) a = ...
let Node_b: (Def_b) b = ...
if (a ≠ b) and (Def_b dominates Def_a) and
   ((b ∈ livein(Node_a)) or (Node_a = Node_b))
    return true {Case 1}
if a is defined as a = φ(a_1 : B_1, ..., a_n : B_n)
    for i = 1 to n
       if b is live out of B_i and b ≠ a_i
          return true {Case 2}
return false
```

Algorithm 12 Construction of initial affinity graph with a depth constraint.

```
Create_affinity_graph(CFG_Node current_node,
                      Integer depth)
(E, V) = (∅, ∅)
for each X = φ(x_1, ..., x_n) of current_node
    V = V ⋃ {Resource_def(X)}
    for each x ∈ {x_1, ..., x_n}
        let Node_x: x = ...
        if depth(Node_x)≠depth
            continue
        V = V ⋃ {Resource_def(x)}
        e = (Resource_def(X), Resource_def(x))
        if (e ∉ E) multiplicity(e)=0
        E = E ⋃ {e}, multiplicity(e)++
return G = (E, V)
```

6 Conclusion

This paper presents a pinning-based solution to the problem of variables coalescing during the out-of-SSA renaming phase. We explain and demonstrate why considering the ϕ-functions renaming and ABI constraints together results in an improved coalescing of variables, thus reducing the number of copy instructions before instruction scheduling and register allocation. Hence, 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, Leung, Briggs, repeated register coalescing). These experiments also show that the affinity and interference graphs are usually pretty simple, which means that a global optimization scheme would bring very little improvement over our local approach. Finally, we implemented small variations of our algorithm and an interesting remark is that an optimistic implementation of interferences, using live-range analysis, already provides good results while reducing significantly the complexity of the computation of the interference graph. During this work we also improved slightly the mark and reconstruct phases of Leung's algorithm which we rely on. A refined version of this algorithm is provided in Appendix A.

References

[1] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, July 1998.

[2] B. de Dinechin, F. de Ferriere, C.Guillon, and A.Stoutchinin. Code generator optimizations for the ST120 DSP-MCU core. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 93 – 103, 2000.

[3] European Telecommunications Standards Institute (ETSI). Gsm technical activity, smg11 (speech) working group. http://www.etsi.org.

[4] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[5] Tim Harvey and Keith Cooper. Fast copy coalescing and live range identification. In *Proc. ACM SIGPLAN Conference on Prog. Language Design and Implementation (PLDI'02)*. ACM Press, June 2002.

[6] A.W. Appel L. George. Iterated register coalescing. *ASM TOPLAS*, 18(3), May 1996.

[7] 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.

[8] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *IEEE PACT*, pages 196–204, 1998.

[9] 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.

[10] Masataka Sassa, Toshiharu Nakaya, Masaki Kohama, Takeaki Fukukoa, and Masahito Takahashi. Static Single Assignment form in the COINS compiler infrastructure.

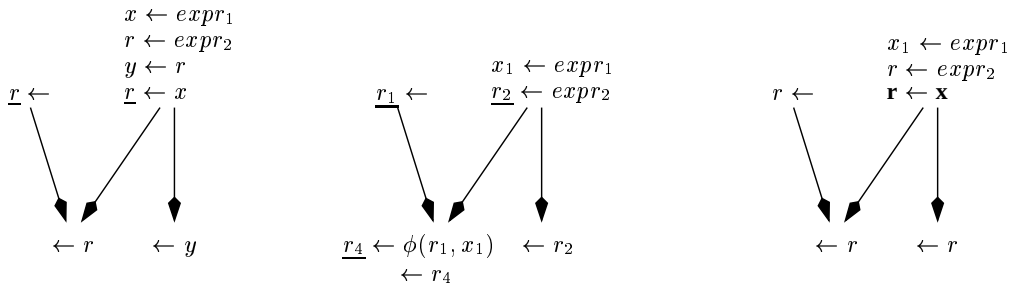
[11] Standard Performance Evaluation Corporation (SPEC). Spec cint2000 benchmarks. http://www.spec.org/cpu2000/CINT2000/.

[12] Arthur Stoutchinin and Francois de Ferriere. Efficient Static Single Assignment form for predication. In *International Symposium on Microarchitecture*. ACM SIGMICRO and IEEE Computer Society TC-MICRO, 2001.

[13] 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.

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 13, 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.



Original code

SSA code with copies folded
r_i are pinned to r

Leung's solution to replace
ϕ-functions is incorrect:
the inserted copy kills the previous
definition of r.

Figure 13: 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 $cp_3 := cp_3 \bigcup \{w \leftarrow z\}$ 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 $must\_def[y] = r$...". As illustrated by Figure 14 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.

```
procedure COLLECT
initialize all entries of must_def and must_use to ⊥
R := ∅
for b ∈ basic blocks do
    for i ∈ φ-functions in b do
        let i ≡ y ← φ(x_1 ... x_n)
▷       if pinned_def(i, 1)
▷           then let r be the dedicated register required
▷           else let r = y
        must_def[y] = r
        R := R∪{r}
    for i ∈ non-φ-functions in b do
        let i ≡ y_1 ... y_m ← op(x_1 ... x_n)
        for j := 1 to m do
            if pinned_def(i, j) then
                let r be the dedicated register required
                must_def[y_j] := r
                R := R∪{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∪{r}
```

```
procedure MARKINIT
for b ∈ basic blocks do
for r ∈ R do
    sites[r] := ∅
    for r ∈ R do
        last[b][r] := ⊤
        phi[b][r] := ⊤
    for i ∈ φ-functions in block b do
        let i ≡ y ← φ(x_1 ... x_n)
        if must_def[y] = r ≠ ⊥ then
            phi[b][r] = last[b][r] = y
            sites[r] := sites[r] ∪ {b}
    for i ∈ normal instructions in block b do
        let i ≡ y_1 ... y_m ← op(x_1 ... x_n)
        for j := 1 to n do
            if must_use[i][j] = r ≠ ⊥ then
                last[b][r] := x_j
                sites[r] := sites[r] ∪ {b}
        for j := 1 to m do
            if must_def[y_j] = r ≠ ⊥ then
                last[b][r] := y_j
                    sites[r] := sites[r] ∪ {b}
▷   for b' ∈ succ_cfg(b) do
▷       let b be the jth predecessor of b'
▷       for i ∈ φ-functions in b' do
▷           let r ≡ must_def[y]
▷           last[b][r] = x_j
▷           sites[r] := sites[r] ∪ {b}
```

```
procedure MARK
MARKINIT()
for r ∈ R do
    repair_name[r] = ⊥
    repair_sites[r] = ∅
for b ∈ basic blocks do
    for r ∈ R do
        avail[r] := avin[b][r]
    for i ∈ normal instructions in b do
        let i ≡ y_1 ... y_m ← op(x_1 ... x_n)
        for j := 1 to n do USE(i, j, x_j)
        for j := 1 to m do DEFINE(y_j)
    for b' ∈ succ_cfg(b) do
        let b be the jth predecessor of b'
        for i ∈ φ-functions in b' do
            let i ≡ y ← φ(... x_j ...)
            USE(i, j, x_k)
```

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

```
procedure USE(i, j, x)
in_place[i][j] = false
if must_use[i][j] ≠ ⊥ and avail[must_use[i][j]] = x then
    in_place[i][j] = true
    return
if must_def[x] ≠ ⊥ and avail[must_def[x]] ≠ x then
    if repair_name[x] = ⊥ then
        repair_name[x] := a new SSA name
    repair_sites[x] := repair_sites[x] ∪ {i}
if must_use[i][j] ≠ ⊥ then
    avail[must_use[i][j]] := x
```

```
procedure DEFINE(x)
if must_def[x] ≠ ⊥ then avail[must_def[x]] := x
```

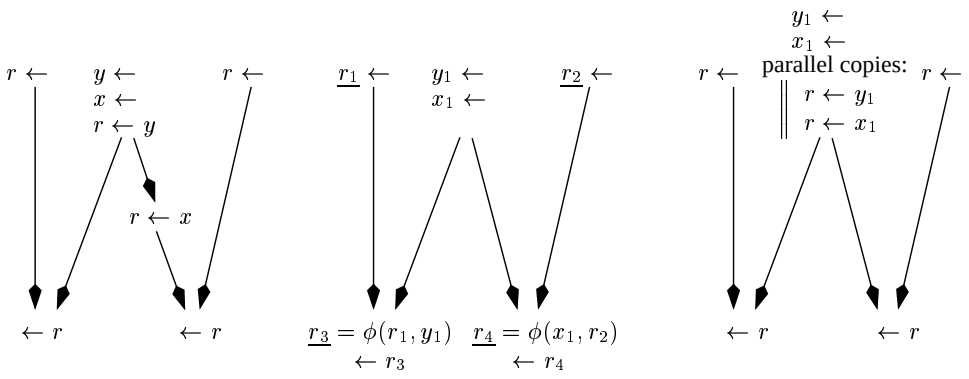
$$avin[b][r] = \begin{cases} \bot \text{ if } b \text{ is the entry} \\ x \text{ if } phi[i][r] = x \neq top \\ \bigcap_{b' \in pred_{cfg}(b)} avout[b'][r] \text{if } b \in DF^+(sites[r]) \\ avout[idom(b)][r] \text{ otherwise} \end{cases}$$

```
procedure LOOKUP(i, x)
if stacks[x] is empty then
    if must_def[x] ≠ ⊥ then
      return must_def[x]
    else return x
else
    let (y, sites) = top(stacks[x])
    if i ∈ sites then return y
    else if must_def[x] ≠ ⊥ 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 jth input operand of i to r
else if r ≠ ⊥ then
    copies := copies ∪ {r ← y}
    rewrite the jth input operand of i to r
else
    rewrite the jth input operand of i to y
return copies

procedure RENAME_DEF(i, j, y, copies)
let r = if must_def[y] = ⊥ then y else must_def[y]
rewrite the jth output operand y to r
if repair_name[y] = tmp ≠ ⊥ then
    push (tmp, repair_sites[y]) onto stacks[y]
    copies := copies ∪ {tmp ← r}
return copies
```

```
procedure RECONSTRUCT(b)
for i ∈ φ-functions in b do
    let i ≡ y ← φ(x_1 ... x_n)
▷   cp_4 = ∅
▷   cp_4 = RENAME_DEF(i, 1, y, cp_4)
▷   insert parallel copies cp_4 after i
for i ∈ normal instructions in b do
    (* rewrite instructions *)
    let i ≡ y_1 ... y_m ← op(x_1 ... x_n)
    cp_1 := ∅
    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 := ∅
    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 := ∅
(* compute φ-copies *)
for b' ∈ succ_cfg(b) do
    let b be the kth predecessor of b'
    for i ∈ φ-functions in b' do
      let i ≡ y ← φ(x_1 ... x_k ... x_n)
▷     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' ∈ succ_dom(b) do
    RECONSTRUCT(b')
      Restore stacks[] to its state
        at the beginning of this call
```



Original code

SSA form after copy folding and dead-code elimination

Code generated by Leung's reconstruction algorithm (incorrect parallel copies)

Figure 14: The *parallel-copies conflict problem* is generated by a too constrained pinning

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.

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 $X_i = \phi(x_{i,1}, \cdots, x_{i,n_i})$. Let us denote by $DEFS = \{X_1, \cdots, X_n\}$, $ARGS_i = \{x_{i,1}, \cdots, x_{i,n_i}\}$, $ARGS = \bigcup_i ARGS_i$ and $\mathcal{V} = DEFS \cup ARGS$.*
Find a partitioning of $\mathcal{V}$ into disjoint sets $R_1, \cdots, R_m$ such that

$$\text{(CK):} \quad \bigcup_{v \in \mathcal{V}} \mathsf{Resource\_killed}(\{v\}) = \bigcup_{1 \le j \le m} \mathsf{Resource\_killed}(R_j) \quad \textit{(no more killed variable)}$$

$$\text{(CS):} \quad \forall 1 \le j \le m, \forall (x,y) \in R_j^2, \neg \mathsf{Variable\_stronglyInterfere(x,y)} \quad \textit{(no strong interference)}$$

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

Definition 2 (LOCAL PINNING **)** *Consider a program P with some pinning already performed and a set of ϕ definitions $X_i = \phi(x_{i,1}, \cdots, x_{i,n_i})$ within the same block B. Let us denote by $DEFS = \{X_1, \cdots, X_n\}$, $ARGS_i = \{x_{i,1}, \cdots, x_{i,n_i}\}$, $ARGS = \bigcup_i ARGS_i$, $\mathcal{V} = DEFS \cup ARGS$ 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 $\underline{R_1}, \cdots, \underline{R_m}$ such that*

$$\bigcup_{\underline{v} \in \underline{\mathcal{V}}} \mathsf{Resource\_killed}(\underline{v}) = \bigcup_{1 \le j \le m} \mathsf{Resource\_killed}(\bigcup \underline{R_j}) \quad \textit{(no more killed variable)}$$

$$\forall 1 \le j \le m, \forall (x,y) \in \left(\bigcup \underline{R_j} \right)^2, \neg \mathsf{Variable\_stronglyInterfere(x,y)} \quad \textit{(no strong interference)}$$

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

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

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

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

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. a definition a_{ji} in block B_i
  2. a definition a_{ij} in block B_j
  3. a block B_{ij} with predecessors B_i and B_j and containing the code
     $C_{ij} : B_{ij} = \phi(x_i : B_i, a_{ij} : B_j)$
     $C_{ji} : B_{ij} = \phi(a_{ji} : B_i, x_j : B_j)$

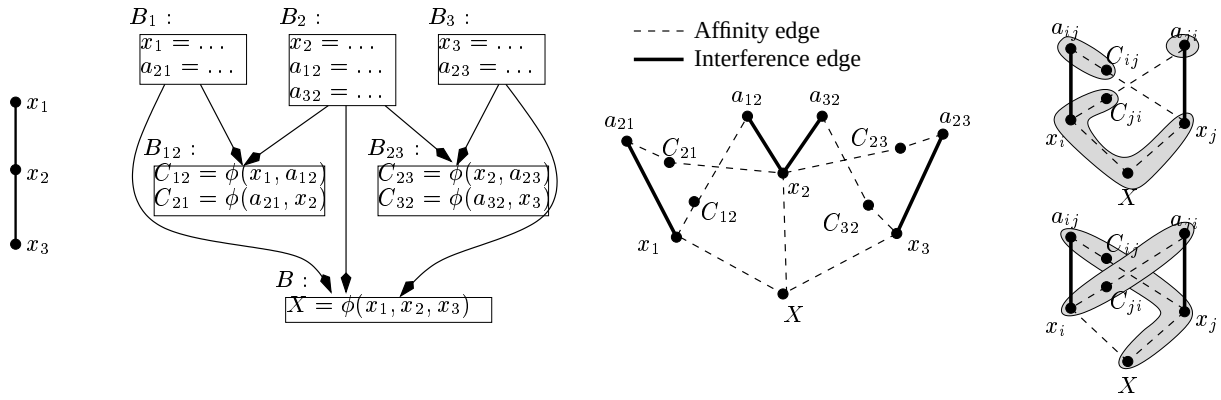
- A block B, with predecessors $B_1, \ldots, B_n$, which contains the instruction $X : B = \phi(x_1 : B_1, \ldots, x_n : B_n)$

For this program,

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

Consider an optimal solution to GLOBAL PINNING with $\underline{X}$ the resource containing X. First, we show by contradiction that $\underline{X}^2 \bigcap E = \emptyset$ i.e. $\underline{X} - \{X\}$ is an independent set. Hence,suppose that $(x_i, x_j) \in \underline{X}^2 \bigcap E$. Then, because of condition (CK), $a_{ij} \notin \underline{x_i} = \underline{X}$. Otherwise the component $\{a_{ij}, x_i, X, x_j\}$ would contain an interference (a_{ij}, x_j). Identically $a_{ji} \notin \underline{x_j} = \underline{X}$. Hence, consider the modified solution where x_i is removed from $\underline{X}$, $\underline{x_i} = \{C_{ij}, x_i, a_{ij}\}$ and $\{C_{ji}, x_j, a_{ji}\} \subset \underline{x_j}$. This solution is strictly better since it increases by two the objective function (CM). Which contradicts the hypothesis that the solution is optimal.

Hence, $\underline{X} - \{X\}$ is an independent set of G. Finally, because maximizing the cardinality of $\underline{X}$ maximizes the cardinality of kept affinity with X, $\underline{X} - \{X\}$ is an Maximal Independent Set. ■



[Initial Graph for] [MAX-INDEPENDANT-SET] | **[Corresponding program for] [GLOBAL-PINNING]** | **[Corresponding affinities] [and interferences]** | **[$\underline{X} - \{X\}$ is an] [independant set]**

Figure 15: Example for the NP-completeness proof of GLOBAL PINNING

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 $\{\underline{X}, \underline{x_1}, \ldots, \underline{x_n}\}$ provides with $\underline{X} - \{X\}$ an independent set for G. ■

Remarks Remark that for the program of GLOBAL PINNING and LOCAL PINNING proofs, the solution provided is *strictly* optimal in term of move coalescing[3] optimization and move instruction[4] minimization. This proves the NP-completeness in the size of ϕ functions for those two problems.

[3]The coalescing problem consist of coalescing a maximum number of move instructions already placed so that the remaining program is still correct

[4]The move instruction minimization has more freedom than a simple coalescing since it can perform partial coalescing, code motion, etc.