Procedure placement using temporal-ordering information: dealing with code size expansionin

Thierry Bidault, Christophe Guillon, Florent Bouchez, Fabrice Rastello

Laboratoire de l'Informatique du Parallélisme

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

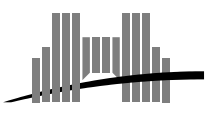
Procedure placement using temporal-ordering information: dealing with code size expansion

Thierry Bidault,
Christophe Guillon,
Florent Bouchez
and Fabrice Rastello

Avril 2004

Research Report N° 04-16

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`



Procedure placement using temporal-ordering information: dealing with code size expansion

Thierry Bidault,
Christophe Guillon,
Florent Bouchez
and Fabrice Rastello

Avril 2004

Abstract

Instruction cache performance is one of the bottle-necks of processor performance. In this paper, we study the effects of procedure placement in memory on a direct-mapped instruction cache. These caches differ from associative memory caches by the fact that each address in the memory is assigned to one and only one address in the cache. This means that two procedures with addresses that share the same place in the cache, and that are called alternatively will create a conflict-miss: one will overwrite the other in the cache. The goal of procedure placement is to minimize these cache-misses. Pettis and Hansen give in [7] a greedy algorithm that doesn't increase the code size. The Gloy and Smith algorithm [3] greatly decreases the number of cache-misses but authorizes gaps between procedures, hence it increases the code size. The latter comprises of two main stages: in the "cache-placement" phase, the procedures are given the location they will occupy in the instruction cache; in the "memory-placement" phase, procedures are placed in memory in such a way that code expansion is minimized, with the constraints of their cache placement. In this article, we prove the NP-completeness of the first stage, and the polynomiality of the second stage of [3]. Indeed, we show that our algorithm provides the optimal solution in a time complexity of $O(nL \log^*(L+n))$ where n is the number of procedures, and L the cache size. Thus nearly linear for a fixed cache size. We also provide an algorithm which, given the cache-placement, quickly returns an approximation of the code expansion. This makes the cache-placement stage take into consideration the final program size. Our modifications to the Gloy and Smith algorithm give on average a code size expansion of 8% over the original program size, while the initial algorithm gave an expansion of 177%. The cache miss reduction is nearly the same as the Gloy and Smith solution with 35% cache miss reduction.

Keywords: Instruction cache, code placement, code size, cache miss, min-matching, hamiltonian-path, profiling

Résumé

Cet article traite du problème de placement de procédures en mémoire pour optimiser l'utilisation d'un cache d'instructions "direct-mapped". Ce type de mémoire cache se distingue des mémoires dites "associatives" par le fait qu'à chaque adresse de la mémoire est associée une unique adresse dans le cache. Ainsi, deux procédures dont les adresses mémoire partagent la même adresse dans le cache et appelées consécutivement créent un "conflit" ou "défaut de cache" : le code de la seconde va écraser le celui de la première. Le but du placement de procédures est de minimiser le nombre de défauts de cache. Pettis et Hansen ont donné dans [7] un algorithme glouton qui n'augmente pas la taille du code ; l'algorithme de Gloy et Smith [3] diminue de beaucoup le nombre de défauts de cache par rapport à [7] mais autorise l'existence de mémoire inutilisée entre les procédures, et donc augmente la taille du code. Ce dernier algorithme est constitué de deux parties principales : la première est une phase de placement dans le cache : chaque procédure se voit attribuer la place qu'elle occupera quand elle sera chargée dans le cache d'instructions ; la seconde partie est une phase de placement en mémoire : les procédures sont placées en mémoire en respectant les contraintes liées au placement dans le cache et de manière à minimiser l'expansion de code. Dans cet article, nous prouvons la NP-complétude de la première partie et la polynomialité de la seconde. En effet, nous exhibons un algorithme qui renvoie la solution optimale au problème de minimisation de l'expansion de code. Sa complexité en temps est en $O(nL \log^*(L+n))$ où n est le nombre de procédures, et L la taille du cache. L'algorithme est donc presque linéaire pour une taille de cache fixée. Nous donnons aussi un outil qui fournit rapidement une approximation de l'expansion de code qui résulte d'un placement dans le cache donné. Ceci permet de prendre en compte la taille finale du programme dans la phase de placement dans le cache. Les modifications apportée à l'algorithme de Gloy et Smith font que celui-ci augmente la taille du code d'environ 8% en moyenne, contre une expansion de code originale d'environ 177%. La réduction du nombre de défauts de cache est quasiment la même que dans l'algorithme original, environ 35% de conflits en moins.

Mots-clés: Cache d'instruction, placement de code, taille de code, defaut de cache, min-matching, chemin hamiltonien, profiling

1 Introduction

Instruction cache performance is one of the bottle-necks of processor performance. In this paper, we study the effects of procedure placement in memory on a direct-mapped instruction cache. An instruction cache is one of the level of the memory hierarchy. When an instruction is not in the cache, it has to be fetched from a deeper level of the memory hierarchy, which consumes cycle and power. We worked on the instruction cache of the ST220 processor, which is a VLIW implementation of the LX/ST200 [1] family. This cache is direct-mapped. These caches differ from associative memory caches by the fact that each address in the memory is assigned to one and only one address in the cache: if L is the cache-size, the address number i in the cache can contain only the instructions whose memory address is $kL + i$. The goal of procedure placement is to avoid setting at the same address modulo L two procedures that are often called together, because such configuration creates cache conflicts. Here, we focus on procedure placement that uses execution profile. An execution profile provides temporal-ordering information, which is important as shown in [3]. There are already several procedure placement algorithms in the litterature, as enumerated below. However, the best techniques for reducing cache conflicts increase the global code size, which make themselves unusable in the embedded context. Taking into account code size expansion during procedure placement for conflict-miss minimization is the main goal of this paper.

State of the art of code layout Pettis and Hansen in [7] build a "call-graph" from the execution profile of a program where the vertices are the procedures of the program and the edges are weighted by the number of calls between procedures. From this call-graph their solution to the procedure placement uses a "closest is best" strategy. Two procedures which often call each other must have a different place in the cache. The idea is that this will be the case if they are put one next to the other in memory. Their algorithm is greedy on the call-graph: it chains the procedures linked by the heaviest edge and merges the corresponding vertices in the graph until all vertices have been merged. This solution is effective in reducing the cache conflicts when the size of the program remains reasonable compared to the size of the cache. An essential property of the Pettis and Hansen placement is that it does not incur code size expansion.

Hashemi, Kaeli and Calder in [4] also build a call-graph. The placement algorithm works with compounds that maintain set of procedures and for each procedure the set of unavailable cache location or "colors". The algorithm merges compounds greedily in decreasing order of edge weights. When two compounds are merged, the unavailable set of colors is updated and some hole can be inserted between procedures when there are color conflicts. This more sophisticated placement algorithm leads to better results than [7] for reducing cache conflicts. On the other hand, the drawback compared to the Pettis and Hansen solution is that inserting holes leads to a code size expansion.

Kalamantianos and Kaeli in [5] build a "conflict miss graph" which modelizes better the interaction between procedures. The placement part is identical to [4]. This leads to better results than Hashemi et al. solution for reducing cache conflicts.

Gloy and Smith in [3] build a more detailed graph from an execution trace: the "temporal relationship graph" (TRG). The vertices are the procedures and if P and Q are two vertices, the edge (P, Q) is weighted by an approximation of the number of conflict cache misses between these procedures. The procedure placement algorithm is composed of two phases: first, the cache-relative placement of each procedure is determined. We call this phase *cache-placement*. The second phase performs the final layout of each procedure in memory constrained by the cache-placement. We call this phase *memory-placement*. These two phases are precisely described below. This solution *is the most effective at reducing cache conflicts*, however it results in an large code size expansion which make it unusable for embedded applications.

Our approach We implemented in our compiler toolchain the different solutions presented above and we made quantitative experiments that resulted in the following two statements. The Pettis and Hansen solution is the best trade-off between cache conflict reduction and code size expansion. The Gloy and Smith [3] solution outperforms the later in cache conflict reduction on

some applications. Thus, we decided to improve the procedure placement phase of the solution in [3] in order to reduce the code size expansion. We attack the problem on the two phases of their algorithm: the cache-placement phase and the memory-placement phase.

The cache-placement problem The cache-placement phase finds the cache-relative position of each procedure in order to minimize cache conflicts. Cache-relative position is the offset, in cache lines, of the beginning of the procedure from the beginning of the cache.

Our first contribution to this problem is to formalize it as the MIN-CACHE-MISS problem in section 2 and to prove that it is NP-complete in Appendix B.

The solution to this problem given in [3] is an heuristic solution which works well in practice. The algorithm processes the Temporal Relationship Graph (TRG), a previously computed call-graph enhanced with temporal information, in decreasing order of weight. It repeatedly merges the two nodes of the current maximum edge until no edges remain. A TRG node contains a set of procedures. For each of these procedures, the node holds its previously-computed *node-relative* cache offset. Initially a node contains only one procedure, whose node-relative offset is 0. So, when merging two nodes, a cache offset for the second node is computed, to minimize a metric between the sets of procedures of the first and second nodes: the all L possible offsets are scanned, and the best one is kept. Once this offset is found, all the individual node-relative offsets of the procedures in the second node are shifted by this offset. This solution only focuses on reducing

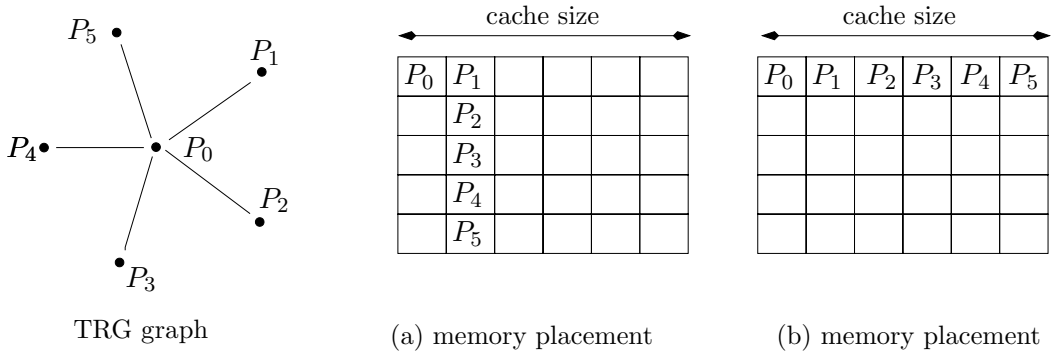


Figure 1: An example of a TRG that could result in a layout of 5 cache-size (a) whereas a layout within 1 cache-size (b) would provide an equivalent miss-rate cost.

the cache conflicts. For instance, the TRG graph in Figure 1 could result in a large code size expansion whereas there exists a less consuming code size solution without increasing the miss rate. Our contribution to this solution is to make this phase sensitive to code size expansion. This is the goal of section 3.

The memory-placement problem The memory-placement phase finds the final procedure layout, i.e. the address in memory of each procedure, given the previously computed cache-placement offsets. In this final layout all procedures are placed in order to minimize the overall code size. The heuristic given in [3] is a greedy algorithm: the algorithm begins with a procedure with an offset of 0; once a procedure is placed, to find the next one, all procedures are scanned and the one having the nearest offset to the end of the already placed procedures is chosen. This algorithm is not optimal[1].

Our first contribution to this phase is to formally state this problem as the MIN-STRING problem in section 2. Then we give an optimal solution to the problem, namely STRING-MERGE in section 4, i.e. we guarantee that for the given cache-placement the code expansion of our memory placement is minimal. We also give in section 4 an efficient implementation of this STRING-MERGE

[1]We can prove that it is optimal within an additive constant of $2L$: the proof is similar to the proof of optimality of MATCHING-GREEDY. There are examples where the bound $2L-2$ is reached. Because our codes have a size of only few times the cache size, the code size expansion due to the use of the greedy algorithm can be consequent.

algorithm in time $O(nL\log^*(L+n))$ where n is the number of procedures and L the number of lines in the cache.

Layout of the article In section 2, we formally state the two problems introduced above: Min-Cache-Miss for the cache-placement phase and Min-String for the memory-placement phase. In section 3, we give our solution to the cache-placement phase. In section 4, we give our solution to the memory-placement phase. In section 5, we give experimental results.

2 Statement of the problems

In this section we state formally the two problems Min-Cache-Miss and Min-String. We also give some preliminary definitions that will be used in the following sections.

2.1 The Min-Cache-Miss problem

Even if it is not exact with architectures which support instruction prefetching, as it is done in the article of Gloy and Smith [3], our modelization supposes that cache-misses only depend on the cache placement of the code. Hence, we will use the following notations: L is the cache size, $l(F)$ is the code size of the procedure F; a cache-placement of the procedures is a function o where for each procedure F, $o(F)$ is the cache address of the beginning of the procedure. With those notations, a procedure F would use the cache-addresses covered by a cyclic interval $I_o(F)$ of size $l(F)$:

Definition 1 (cyclic interval) *A cyclic interval I is determined by an offset $o(I)$ and an end $e(I)$ ($I = [o(I); e(I))$). Both are in $\mathbb{Z}/L\mathbb{Z}$ where L is a constant. The size (or length) of a cyclic interval is $l(I) = l(F) \mod L = (e(I) - o(I)) \mod L$.*

Figure 2 gives an example of five cyclic intervals and their cache-placment in a chache of size $L = 14$.

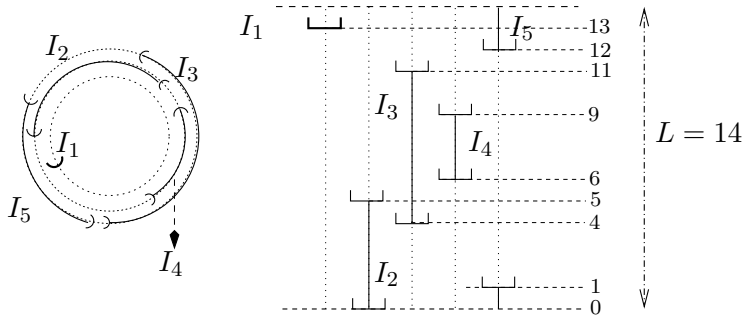


Figure 2: An example of cyclic intervals and its corresponding modular representation. The convention for interval I_1 is a 0 size interval.

Definition 2 (collision) *Given two procedures A and B. The collision $col(A, B)$ between A and B is true if and only if either $l(A) \geq L$ or $l(B) \geq L$ or their corresponding cyclic intervals intersect ie $[o(A), e(A)) \cap [o(B), e(B)) \neq \emptyset$.*

Definition 3 (trace) *The trace of a program execution is the sequence of the fonctions it went through. Given the following program :*

```
program S
  do i = 1, 7
    if condition(i) then call A
    else call B
```

A trace of S would be a word of the regular expression $(S(A|B))^7 S$.

Definition 4 (cache miss) *Given a trace $\mathcal{T} = F_{i_1}F_{i_2}\dots F_{i_T}$. Let $t \in \{1,\dots,T\}$, there is a cache-miss at time t, if and only if there is a cold miss or a conflict miss. It is a cold miss whenever it is the first occurence of F_{i_t} in $\mathcal{T}$, i.e. at the first invocation of F_{i_t}. A conflict miss is when between times $t' < t$, where t' is the largest time such that $F_{i_{t'}} = F_{i_t}$, there is a time t'' with $t' < t'' < t$ such that $col(F_{i_t}, F_{i_{t''}})$ is true, i.e. $F_{i_{t''}}$ has evicted F_{i_t} from the cache. We set $\widehat{Miss}(t)$ to 1 in case of a cache miss, and to 0 otherwise.*

Definition 5 (allocation cost) *The allocation cost for a trace $\mathcal{T} = F_{i_1}F_{i_2}\dots F_{i_T}$ is $\widehat{Miss}_{\mathcal{T}}(o) = \sum_{t=1}^{T} \widehat{Miss}(t)$.*

Definition 6 (Min-Cache-Miss$(\mathcal{T}, P, k, L)$) *Given a trace $\mathcal{T}$ of a program $P = F_{i_1}F_{i_2}\dots F_{i_m}$, k an integer and a cache of fixed size L, find an allocation o for its procedures so that $\widehat{Miss}_{\mathcal{T}}(o) < k$.*

This problem is NP-complete. The complete proof is in Appendix B, but we will give the main ideas: cache-misses can be represented by a complete graph where the nodes are the procedures of the program, and the edges are weighted by an approximation of the number of cache conflicts between procedures (TRG in [3]). This approximation is exact on some particular configurations. In such cases, if all the procedures have the size 1, the problem is similar to finding a coloration of the nodes which minimizes the sum of the weights of edges whose nodes have the same color (i.e. the corresponding procedures are at the same place in the cache). This problem corresponds to the MAX-kCUT problem [2] which reduces to Graph K-Colorability when all edges have the same weight. Hence the proof of NP-completeness uses a reduction from Graph K-Colorability [2]: given any instance of Graph K-Colorability, i.e. given the graph G, we construct a program S such that finding the best procedure cache-placement for S provides a L-coloration of G.

2.2 The Min-String problem

This paragraph provides a formal definition of the memory-placement problem namely the Min-String problem. Functions on which respective offsets have been provided by the cache-placement phase are represented by *cyclic intervals* on $\mathbb{Z}/L\mathbb{Z}$. Functions have then to be "ordered" in memory. The sum of all gaps between procedures induced by the offset constraint is the code size expansion: we say, the *cost* of the obtained mapping. The goal is to minimize this cost.

Definition 7 (cost) *The cost $\widehat{c}$ of a couple of intervals (I_i, I_j) is equal to the size of the cyclic gap $[e(I_i); o(I_j))$, i.e.*

$$\widehat{c}(I_i, I_j) = (o(I_j) - e(I_i)) \mod L$$

Definition 8 (string, cost of a string, length of a string) *Given a set $\mathcal{I} = \{I_1,\dots,I_n\}$ of cyclic intervals, a string is an ordering $I_{i_1},\dots,I_{i_n}$ of $\mathcal{I}$. This string is denoted by $I_{i_1}I_{i_2}\dots I_{i_n}$. Let S be a string $I_{i_1}\dots I_{i_n}$, its cost is $\widehat{C}(S) = \sum_{1\le k<n} \widehat{c}(I_{i_k}, I_{i_k+1})$, i.e. the total size of the cyclic gaps between the intervals of the string. The length of S is $l(S) = \widehat{C}(S) + \sum_{1\le k\le n} l(I_{i_k})$, i.e. the total length of the intervals and gaps of the string.*

Definition 9 (Min-String) *Let $\mathcal{I} = \{I_1,\dots,I_n\}$ be a set of cyclic intervals of $\mathbb{Z}/L\mathbb{Z}$. Find a string of $\mathcal{I}$ such that its cost is minimized.*

3 Cache-placement phase sensitive to code size expansion

In this article, we do not address the problem of minimizing cache misses. Instead we aim at modifying the algorithm of Gloy and Smith [3] to take into account code size expansion. Hence, we work on the TRG graph and consider it like if it were an exact representation of the final cache miss cost. Formally, the TRG is a non complete graph where each vertex is labeled by a procedure. Edges are weighted with positive costs: the weight of an edge $\widehat{\text{Miss}}(P, Q)$ is an approximation of

the conflict miss cost we would pay if procedures P and Q share a common place in the cache. A cache-placement o associates to each vertex P a cyclic interval $\mathrm{I}_o(P)$ of size $l(P)$. The conflict miss cost of a cache-placement can be defined as follow:

Definition 10 (conflict miss cost) *Let $G = (V, E)$ be a TRG graph, o a cache-placement. Then the cost of o is*

$$\widehat{Miss}_{TRG}(o) = \sum_{(P,Q)\in E,\, I_o(P) \bigcap I_o(Q)\neq\emptyset} \widehat{Miss}(P, Q)$$

The problem addressed by Gloy and Smith is to find an allocation o that minimizes $\widehat{\mathrm{Miss}}_{TRG}(o)$. This problem, namely the SHIP-BUILDING problem is still NP-complete. Their solution is a greedy algorithm that merges agregates of vertices into larger agregates where the relative placement within an agregate is fixed by the merge. All L possibilities to merge two agregates together are compared using a cost for this agregate cache-placement. One of our goal will be to modify this cost to take into account the code size expansion. The pseudo-code for the heuristic of Gloy and Smith is given in algorithm 1.

Algorithm 1

GloySmith_cache_placement(G)
  {G is a weighted TRG graph}
  let $G' = (V', E')$ with $[\{P\} \in V' \Leftrightarrow P \in V]$ and $[(\{P\}, \{Q\}) \in E' \Leftrightarrow (P, Q) \in E]$
  forall $\{P\} \in V'$
    $o(P) = 0$
  forall $e = (\{P\}, \{Q\}) \in E'$
    $w(e) = \widehat{\mathrm{Miss}}(P, Q)$
  while card$V' > 1$
    let $(A, B) \in E'$ s.t. $w(A, B) = \max_{e\in E'} w(e)$
    let $A = \{A_1, \ldots, A_p\}$ and $B = \{B_1, \ldots, B_q\}$
    forall $\lambda \in [0, L)$
      $\widehat{C}_\lambda(A, B) = \widehat{\mathrm{Miss}}_\lambda(A, B) = \sum_{(A_i,B_j)\in A\times B \text{ s.t. } \mathrm{I}_{o(A_i)} \bigcap \mathrm{I}_{o(B_j)}\neq\emptyset} \widehat{\mathrm{Miss}}(A_i, B_j)$
    let λ_{min} s.t. $\widehat{C}_{\lambda_{min}}(A, B) = \min_{\lambda\in[0,L)} \widehat{C}_\lambda(A, B)$
    forall $B_i \in B$
      $o(B_i) = (o(B_i) + \lambda_{min}) \mod L$
    merge A and B in G' with
      forall $C \in V'$
        $w(A \bigcup B, C) = w(A, C) + w(B, C)$
  return o

In this section we present our modifications to the cache-placement phase. The objective is to find a trade-off between cache conflict reduction and code size expansion. In order to account for code size expansion, we modify the cost of a merge $\widehat{C}_\lambda$ of the GloySmith_cache_placement algorithm. For this we define two metrics, the running time expansion cost $\widehat{\mathrm{Exp}}_{time}$ and the code size expansion cost $\widehat{\mathrm{Exp}}_{size}$. We will first give definitions for the new cost metrics and then we will present our method for estimating the $\widehat{\mathrm{Exp}}_{size}$ metric.

3.1 Cost of a merge, running time and code size expansion costs

We define our new cost metric for a merge as

Definition 11 (cost of a merge) *Let $G = (V, E)$ be a TRG graph and $G' = (V', E')$ the merged graph before the current merge. Let $(A, B) \in E'$ the edge considered for the merge. We suppose $o(A) = 0$, and we define $\widehat{C}_\lambda(A, B)$ the cost of the merge where $o(B) = \lambda$ as*

$$\widehat{C}_\lambda(A, B) = \alpha\widehat{Exp}_{time_\lambda}(A, B) + (1 - \alpha)\widehat{Exp}_{size_\lambda}(A, B)$$

where $\widehat{Exp}_{time_\lambda}(A, B)$ is the estimated running time expansion cost and $\widehat{Exp}_{size_\lambda}(A, B)$ is the estimated code size expansion cost for the merge of A and B.

The α value is a real parameter in $[0,1]$ to control the trade-off between running time and code size expansion. Note that for $\alpha = 1$ the cache-placement phase will give the same placement as the original cache-placement algorithm 1. With $\alpha = 0$ only code size expansion is minimized and in this case all temporal information is lost, giving a solution similar to the Pettis and Hansen "closest is best" solution .

We define the running time expansion cost for a merge as

$$\widehat{\text{Exp}}_{time_\lambda}(A,B) = \frac{\widehat{\text{Miss}}_\lambda(A,B) \times \text{Time}_{miss} + \text{Time}_{min}}{\text{Time}_{min}}$$

where Time_{min} is an estimation of the minimal running time for any placement, $\widehat{\text{Miss}}_\lambda(A,B)$ is the cost of the placement defined in algorithm 1 and Time_{miss} is the running time cost of a single miss. The Time_{min} value is constant for a given trace and is estimated in our implementation as the number of cycles of the input trace with a perfect cache that never causes conflict miss. Thus the $\widehat{\text{Exp}}_{time_\lambda}(A,B)$ can be computed trivially in the merge phase.

The code size expansion cost is estimated as

$$\widehat{\text{Exp}}_{size_\lambda}(A,B) = \frac{\widehat{\text{Size}}_\lambda(A,B)}{\widehat{\text{Size}}_{min}(A,B)}$$

where $\widehat{\text{Size}}_{min}(A,B)$ is the minimal memory-placement size for any cache-placement and $\widehat{\text{Size}}_\lambda(A,B)$ is an estimation of the memory-placement size for a cache-placement o that verifies $o(A) = 0$ and $o(B) = \lambda$. The $\widehat{\text{Size}}_{min}(A,B)$ is estimated as the sum of the size of the procedures to be merged, thus $\widehat{\text{Size}}_{min}(A,B) = \sum_{P_i \in A \bigcup B} l(P_i)$. It is the final size of the memory-placement with no cache-placement constraint. The $\widehat{\text{Size}}_\lambda(A,B)$ is explained in next section. An implementation could use the solution to the MIN-STRING problem that we give in this paper, however, while our solution is nearly linear in the number of functions for a constant cache size L, it is too costly for the cache-placement algorithm as it has to be run for each λ value at each merge thus Ln times.

Our solution provides a fast estimation of the memory-placement cost given a cache-placement.

3.2 Estimating the memory-placement cost

To estimate $\widehat{\text{Size}}_\lambda(A,B)$ we compute an estimation of the cost of the memory-placement of $A \bigcup B$ given a cache-placement o. We will call our estimation $\widehat{\text{Size}}_o$.

We first introduce the interval histogram as

Definition 12 (histogram) *Let o be the cache-placement for a set of procedures $\{P_1, \ldots, P_n\}$. Let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be the set of cyclic intervals such that $I_i = I_o(P_i)$. We define the histogram for o as the function*

$$\forall h \in \{0, \ldots, L-1\}, \mathcal{H}_o(h) = \sum_{I_i \in \mathcal{I}} (1 \text{ if } h \in I_i, \quad 0 \text{ otherwise})$$

The histogram gives for each cache location the number of distinct intervals that are mapped to this location, i.e. the number of procedures that will share this location in the cache. We define $\mathcal{H}max_o$ the value such that $\mathcal{H}max_o = max_{h \in \{0,\ldots,L-1\}} \mathcal{H}_o(h)$, i.e. the largest number of procedures that will share the same location in the cache. As we have at least one location in the cache where $\mathcal{H}max_o$ procedures must be placed, we need at least $\mathcal{H}max_o - 1$ cache size in memory to hold the procedures. Thus for all valid memory-placement string S_o we have

$$l(S_o) \geq (\mathcal{H}max_o - 1)L$$

In case $\mathcal{H}max_o = 1$ and there are only few intervals, this metric is clearly too optimistic. In that case, $l(S_o)$ is tightly bounded by the size of the smallest cyclic interval I_{cover} that verifies $\forall I_i \in \mathcal{I}$, $I_i \subset I_{cover}$. In the more general case our approximation can be refined to

$$l(S_o) \geq (\mathcal{H}max_o - 1)L + \mathcal{H}cover_o$$

where $\mathcal{H}cover_o$ is defined as

Definition 13 (minimum cover) *Let $\mathcal{H}_o$ be the histogram for a placement o, we define a minimum cover interval as a cyclic interval I_{cover} of minimum length $\mathcal{H}cover_o = l(I_{cover})$ with $\forall h$ s.t. $\mathcal{H}_o(h) = \mathcal{H}max_o$, $h \in I_{cover}$.*

Thus, for instance, in the case where the α parameter of our merge cost metric is 0, i.e. the procedure placement is biased toward code size reduction, at each merge step the $\widehat{\text{Size}}_o$ value will be minimal for a placement that inserts no gap. Thus the final cache-placement will be such that an optimal memory-placement will give no size expansion.

4 Optimal solution to the memory-placement problem

This part is devoted to the description of our solution to the memory-placement phase namely the String-Merge algorithm. Proofs and details are provided in Appendix A. Section 4.1 introduces notations and exposes the relationship between our Min-String problem, the NP-complete Hamiltonian-Path problem and the polynomial Min-Matching problem. Section 4.2 presents our algorithm which uses the solution to a particular Hamiltonian-Circuit problem that itself uses the solution to a particular Min-Matching problem.

4.1 Cyclic intervals, string, cycle and matching

In this part and in the following we consider the set of cyclic intervals denoted by $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$. $\widehat{c}$ is the cost defined on page 4.

4.1.1 String, path and cycle

Let us consider the complete weighted and oriented graph $G_{\mathcal{I}} = (V, V^2)$ with n vertices labeled by $I_1, \ldots, I_n$. Each vertex (I_i, I_j) is weighted by $\widehat{c}(I_i, I_j)$. Then, the minimum string is an Hamiltonian path of minimum weight. We say *minimum string* because usually on the Hamiltonian path problem extremums I_{i_1} and I_{i_n} are fixed. By adding a vertex I_0 to $G_{\mathcal{I}}$ with a zero weight on all edges coming from and going to this vertex, Min-String($\mathcal{I}$) relies on MIN-HAMILTONIAN-CIRCUIT($G_{\mathcal{I}} \bigcup \{I_0\}$) [6]. Like this our problem seems to be NP-complete. An issue is toward the metric properties of our graph: there exists approximation results on the metric version of the traveling salesman problem with triangular inequality. But this version is still APX-complete and in our case the metric is cyclic and the triangular inequality property is not checked everywhere. Hence, as far as we know, till now there are no known results for our optimization problem, and it looks like a difficult problem.

Fortunately, our cyclic-metric version of the traveling salesman problem is polynomial:

Definition 14 (cycle, Hamiltonian circuit, cost of a circuit) *let $\mathcal{I} = \{I_1, \ldots, I_n\}$ be a set of cyclic intervals. Let $\mathcal{I}' = \{I_{i_1}, I_{i_2}, \ldots, I_{i_m}\}$ be a subset of $\mathcal{I}$, and $I_{i_1}, I_{i_2}, \ldots, I_{i_m}$ an ordering of this subset. Then, $\circlearrowleft(I_{i_1}, I_{i_2}, \ldots, I_{i_m})$ is a cycle of $\mathcal{I}$. Notice that $\circlearrowleft(I_{i_k}, I_{i_{k+1}}, \ldots, I_{i_m}, I_{i_1}, \ldots, I_{i_{k-1}})$ would also denote the same cycle. If $m = n$ then it is said to be an Hamiltonian circuit. The cost of a circuit is $\widehat{C}(\circlearrowleft(I_{i_1}, \ldots, I_{i_n})) = \widehat{c}(I_{i_1}, I_{i_2}) + \cdots + \widehat{c}(I_{i_{n-1}}, I_{i_n}) + \widehat{c}(I_{i_n}, I_{i_1})$. Notice that it is the cost of the string $\widehat{C}(I_{i_1} \ldots I_{i_n})$ plus the cost of the "closing gap" $\widehat{c}(I_{i_n}, I_{i_1})$.*

The following definition states the cyclic-metric minimum Hamiltonian cycle problem:

Definition 15 CMetric-MH-Circuit *Let $\mathcal{I}$ be a set of cyclic intervals, $G_{\mathcal{I}}$ the corresponding cyclic-metric weighted complete graph. Find the Hamiltonian cycle of minimum weight. In other words find an ordering $(i_1, \ldots, i_n)$ such that $\widehat{C}(\circlearrowleft(I_{i_1}, \ldots, I_{i_n}))$ is minimized.*

Because $G_{\mathcal{I}} \bigcup \{I_0\}$ as defined above is not a cyclic-metric graph, CMetric-MH-Circuit *is not strictly equivalent to our* Min-String *problem. But, a first approach shows at most a coefficient of L^2 between those two problems: we consider all L^2 possible dummy intervals of $\mathbb{Z}/L\mathbb{Z}$ say I_g, evaluate the L^2 strings* CMetric-MH-Circuit($G_{\mathcal{I} \bigcup \{I_g\}}$)*;* Min-String($\mathcal{I}$) *is the minimal one.*

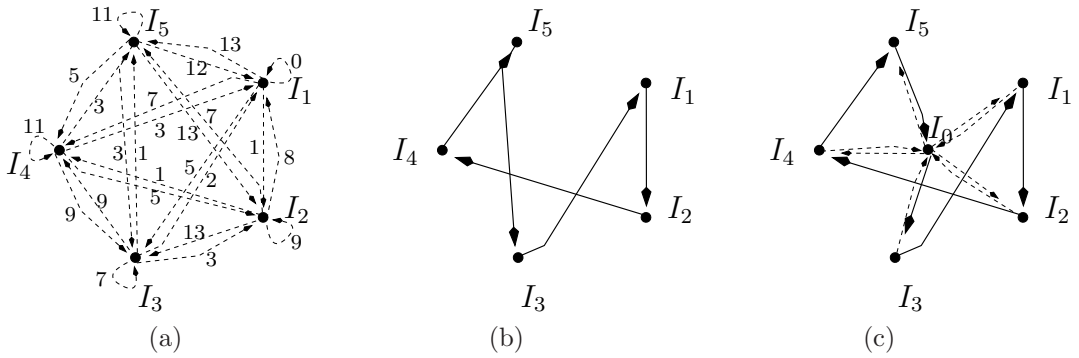


Figure 3: (a) The weighted oriented complete graph $G_{\mathcal{I}}$ corresponding to the example of Figure 2; (b) $(I_1, I_2, I_4, I_5, I_3)$ is a minimum Hamiltonian cycle for this graph ; (c) $(I_1, I_2, I_4, I_5, I_0, I_3)$ is a minimum Hamiltonian cycle for $G_{\mathcal{I}} \bigcup \{I_0\}$ so $I_3I_1I_2I_4I_5$ is a minimum string for $\mathcal{I}$.

4.1.2 Matching

Consider $G_{\mathcal{I}}$ and split each vertex I_i into two vertices $e(I_i)$ and $o(I_i)$; join each $e(I_i)$ to each $o(I_j)$ with an edge of weight $\widehat{c}(I_i, I_j)$. This is the corresponding split complete bipartite graph of $G_{\mathcal{I}}$, say $BG_{\mathcal{I}}$. To an Hamiltonian circuit of $G_{\mathcal{I}}$ corresponds a perfect matching of $BG_{\mathcal{I}}$. On the other hand, the set of edge of a perfect matching of $BG_{\mathcal{I}}$ does not provide a unique cycle in $G_{\mathcal{I}}$, but more generally a *set of disjoint cycles*.

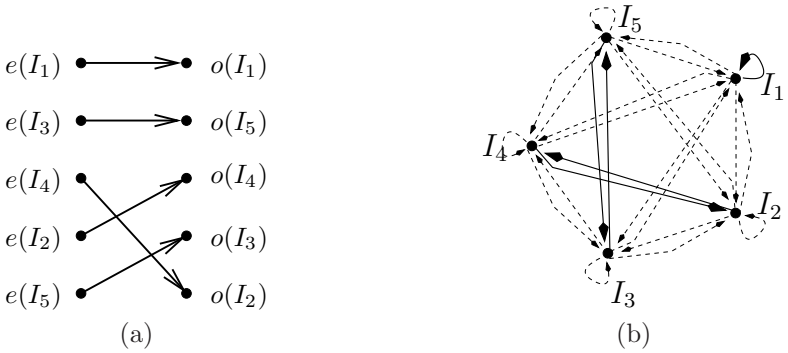


Figure 4: (a) A solution to the MIN-MATCHING problem for the example of Figure 3; (b) This solution does not provide an Hamiltonian circuit, but several disjoint cycles instead.

Definition 16 (Matching, cost of a matching) *Let $BG = (V, U, V \times U)$ be a weighted bipartite complete graph with weights $\widehat{c}(v, u) \geq 0$. A matching is a bijective assignment σ which connects every $v \in V$ to one $\sigma(v) \in U$. The cost of a matching is $\widehat{C}(\sigma) = \sum_{v \in V} \widehat{c}(v, \sigma(v))$.*

The weighted matching problem of a bipartite graph is a known polynomial problem that can be solved in $O(n^3)$ using the Hungarian method [6]:

Definition 17 (MIN-MATCHING**)** *Let $BG = (V, U, V \times U)$ be a weighted bipartite complete graph with weights $\widehat{c}(v, u) \geq 0$. Find a matching σ of minimum cost.*

The restriction of the matching problem to $BG_{\mathcal{I}}$ can be stated as follow: find a bijective assignment σ which connects every I_i to one $\sigma(I_i)$ such that the cost $\widehat{C}(\sigma) = \sum_{i=1}^{n} \widehat{c}(I_i, \sigma(I_i))$ is minimized.

In the general case, a solution to the MIN-MATCHING problem does not provide a solution to the MIN-HAMILTONIAN-CIRCUIT problem. But *in our particular case*, from a solution to MIN-MATCHING($\mathcal{I}$), it is possible to merge the obtained disjoint cycles into a unique cycle that would be solution to CMETRIC-MH-CIRCUIT.

4.1.3 From MIN-MATCHING to MIN-STRING

To summarize, to a set of n cache-placed procedures represented by the cyclic intervals $\mathcal{I} = \{I_1, \ldots, I_n\}$, our code expansion minimization problem relies on MIN-STRING($\mathcal{I}$). In the general case, MIN-STRING is NP-complete slightly similar to the traveling salesman problem. An Hamiltonian circuit in an oriented graph, provides a perfect matching. In the general case, the minimum perfect matching problem which is polynomial is not very useful to solve the minimum Hamiltonian circuit problem. Our problem is slightly different in the way that our initial graph has the particular property that we call cyclic-metric. Under these conditions, MIN-MATCHING can be solved in $O(n \log^*(L))$; then the obtained disjoint cycles can be merged into a minimum Hamiltonian circuit solution of CMETRIC-MH-CIRCUIT in $O(n \log^*(n))$; finally, solving MIN-STRING can be done by solving CMETRIC-MH-CIRCUIT on different sets of intervals $\mathcal{I} \bigcup \{I_g\}$ where I_g takes all L^2 possible cyclic interval values $\mathbb{Z}/L\mathbb{Z}$. This provides an algorithm in $O(nL^2 \log^*(L+n)$. Note that only $2L$ intervals have to be considered which reduces the complexity to $O(nL \log^*(L+n))$.

4.2 The STRING-MERGE algorithm

This section presents our solution to the MIN-STRING problem: the MATCHING-GREEDY algorithm given in Paragraph 4.2.1 solves the MIN-MATCHING problem in $O(n \log^*(L))$; then, Paragraph 4.2.2 describes the CYCLE-MERGE algorithm that merges the obtained disjoint cycles into a minimum Hamiltonian circuit in $O(n \log^*(n))$; finally, our STRING-MERGE algorithm that solves the MIN-STRING problem in $O(nL \log^*(L+n))$ is given in Paragraph 4.2.3.

4.2.1 Optimal matching with MATCHING-GREEDY

Following is the formal definition of the MATCHING-GREEDY algorithm.

Definition 18 (MATCHING-GREEDY)
Let π be a permutation of $\{1, 2, \ldots, n\}$.
MATCHING-GREEDY(π)
  not_taken $= \{I_1, \ldots, I_n\}$
  do i=1,n
    let $I_k \in$ not_taken **such that**
      $\widehat{c}_\epsilon(I_{\pi(i)}, I_k)$ minimum
    not_taken = not_taken - $\{I_k\}$
    $\sigma(I_{\pi(i)}) = I_k$
  return σ
This greedy algorithm returns a matching where the couples are determined in the order of the permutation π. The order corresponding to the permutation π is $<_\pi$ such that $I_{\pi(1)} <_\pi I_{\pi(2)} <_\pi \cdots <_\pi I_{\pi(n)}$. Assignments are made following the rule "the nearest, the best"[2].

This algorithm provides an optimal solution to the MIN-MATCHING problem. The proof is given in Appendix A. The complexity of this algorithm corresponds to n iterations of finding a minimum over n elements. Naively, it would cost $O(n \log(n))$. In fact, the elements are bounded by 0 and L and the complexity can be reduced to $O(n \log^*(L))$ thanks to the implementation in algorithm 2. In the following pseudo-code, *DSF* is for Disjoint-Sets-Forest and *RDL* is for Ranked-Disjoint-Lists ie an array of lists. The procedures used in these pseudo-code are described in Appendix C.

[2] $\widehat{c}_\epsilon$ provides a deterministic algorithm: if there is more than one possibility, the bigger interval (in terms of length) is chosen. If there is still more than one interval, the one with the smallest index is chosen.

Implementation and complexity of MATCHING-GREEDY

Algorithm 2

procedure MATCHING-GREEDY($\mathcal{I}, \sigma$)
  $\Lambda_o \leftarrow$DSF-NEW(L)
  BUILD-NEAREST-ORIGINS-DSF($\mathcal{I}, \Lambda_o$)
  $\mathcal{L}_o \leftarrow$ARRAY-NEW(L)
  BUILD-INTERVAL-ORIGINS-RDL($\mathcal{I}, \mathcal{L}_o$)
  foreach $i \leftarrow (1, \ldots, n)$ **do**
    $\lambda_o \leftarrow$DSF-ROOT($\Lambda_o, e(\mathcal{I}[i])$)
    $j \leftarrow$RDL-POP($\mathcal{L}_o, \lambda_o$)
    $\sigma[i] \leftarrow j$
    if RDL-IS-EMPTY($\mathcal{L}_o, \lambda_o$) **then**
      $k \leftarrow (\lambda_o + 1) \mod L$
      $\lambda'_o \leftarrow$DSF-ROOT(Λ_o, k)
      if $\lambda'_o \neq \lambda_o$ **then**
        DSF-UNION($\Lambda_o, \lambda_o, \lambda'_o$)
  return

The first step of the MATCHING-GREEDY procedure finds all interval origin locations and put each cache location in the set of the nearest origin location. The returned disjoint set of locations Λ_o give for any cache location the nearest origin location. This step is done by the BUILD-NEAREST-ORIGINS-DSF procedure in $O(n)$. The second step builds for each interval origin location the list of interval starting at this location. This step is done by the BUILD-INTERVAL-ORIGINS-RDL procedure in $O(n)$. The main loop of the MATCHING-GREEDY procedure iterates over the set of intervals. For each interval, given the location in the cache $e(I_i)$, the nearest list of interval origin is returned by the procedure DSF-ROOT. From this list, an interval is taken and the matching is updated. When the list of intervals at this origin is empty, the set of locations is merged with the nearest set of locations. The number of DSF operations is the number of initial interval origin locations bounded by $\min(n, L)$. Thus the complexity of MATCHING-GREEDY is $O(n \log^*(L))$.

4.2.2 Optimal circuit with CYCLE-MERGE

In this paragraph, we describe the *simple merge* and *final merge* processes that are used to build a minimum Hamiltonian circuit from an optimal solution to MIN-MATCHING. The notion of merging uses the notion of gaps:

Definition 19 (gap) *Let σ be a matching of $\mathcal{I}$, then the gap $[e(I_i), o(\sigma(I_i)))$ that follows the interval I_i is denoted by $(\vdash_\sigma .I_i) = [o((\vdash_\sigma .I_i)), e((\vdash_\sigma .I_i)))$ and by G_i when the notation may be confusing. If $C = \circlearrowleft(I_{i_1}, \ldots, I_{i_m})$ is a cycle of size m, we define the set of gaps of C by the set of m gaps $\{G_{i_1}, \ldots, G_{i_m}\}$. Finally, the topological closure of $G_i = [o(G_i)), e(G_i))$ is denoted by $\overline{G_i} = [o(G_i)), e(G_i)]$.*

Definition 20 (mergeable) *Let σ be a matching of $\mathcal{I}$, and $C = \circlearrowleft(I_{i_1}, I_{i_1}, \ldots))$, $C' = \circlearrowleft(I_{j_1}, I_{j_2}, \ldots)$ be two disjoint cycles of σ. C and C' are said to be mergeable if there exists two gaps G_i and G_j respectively from C and C' such that $\overline{G_i} \bigcap \overline{G_j} \neq \emptyset$.*

Definition 21 (simple merge) *Let σ be a matching of $\mathcal{I}$, and C, C' two mergeable cycles with intersecting gaps $\overline{G_i}$ and $\overline{G_j}$. The corresponding* simple merge *transposes the successors of I_i and I_j, $\sigma' = \sigma \circ \tau_{I_i I_j}$. The simple merge process is illustrated by Figure 5.*

SIMPLE-MERGE($\mathcal{I}, \sigma$)
  $\sigma' = \sigma$
  let $\mathcal{C}$ the set of cycles of σ
  let $\mathcal{G}$ the set of gaps of σ
  forall $(\vdash_\sigma .I_i) \neq (\vdash_\sigma .I_j)$ in $\mathcal{G}$
    let C, C' **such that** $I_i \in C$, $I_i \in C'$
    if $C \neq C'$
      $\sigma'(I_i, I_j) \leftarrow \sigma'(I_j, I_i)$
  return σ'

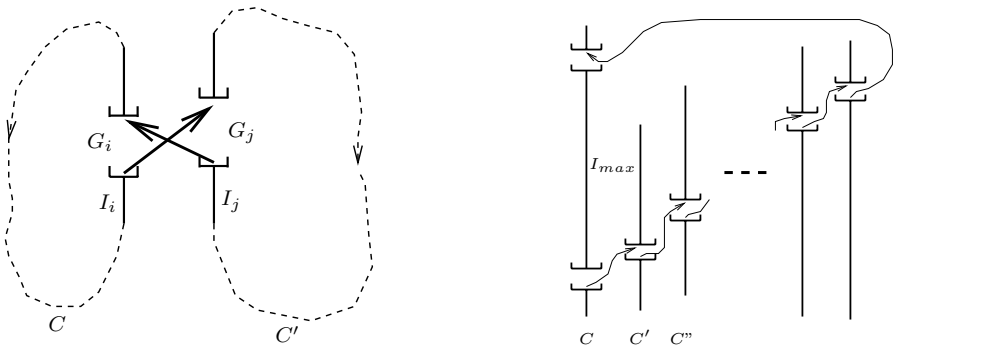
This algorithm merges all mergeable cycles. The implementation given in algorithm 3 has a complexity of $O(n \log^*(n))$.

Implementation and complexity of SIMPLE-MERGE

Algorithm 3

procedure SIMPLE-MERGE($\mathcal{I}, \sigma$)
  $\mathcal{C} \leftarrow$DSF-NEW(n)
  BUILD-CYCLES-DSF($\sigma, \mathcal{C}$)
  if DSF-ROOT-COUNT($\mathcal{C}$) $= 1$ **then return** 1
  $\Lambda_e \leftarrow$DSF-NEW(L)
  BUILD-NEAREST-ENDS-DSF($\mathcal{I}, \Lambda_e$)
  $\mathcal{L}_e \leftarrow$RDL-NEW(n, L)
  BUILD-INTERVAL-ENDS-RDL($\mathcal{I}, \mathcal{L}_e$)
  $\lambda_{max} \leftarrow -1$
  foreach $i \leftarrow (0, \ldots, n)$ **do**
    if $o(\mathcal{I}[\sigma[i]]) < e(\mathcal{I}[i])$ **then**
      $\lambda \leftarrow o(\mathcal{I}[\sigma[i]])$
      if $\lambda > \lambda_{max}$ **then**
        $\lambda_{max} \leftarrow \lambda$
        $i_{max} \leftarrow i$
        $C_{max} \leftarrow$DSF-ROOT($\mathcal{C}, i_{max}$)
  $k \leftarrow 0, \lambda_e \leftarrow$DSF-ROOT($\Lambda_e, 0$)
  while $k <$ DSF-ROOT-COUNT(Λ_e) **do**
    if $\lambda_{max} < k$ **then**
      $i_{max} \leftarrow$RDL-POP($\mathcal{L}_e, k$)
      $\lambda_{max} \leftarrow o(\mathcal{I}[\sigma[i_{max}]])$
      if $\lambda_{max} < k$ **then** $\lambda_{max} \leftarrow \lambda_{max} + L$
      $C_{max} \leftarrow$DSF-ROOT($\mathcal{C}, i_{max}$)
    while *not* RDL-IS-EMPTY($\mathcal{L}_e, k$) **then**
      $i \leftarrow$RDL-POP($\mathcal{L}_e, k$)
      $C \leftarrow$DSF-ROOT($\mathcal{C}, i$)
      if $C \neq C_{max}$ **then**
        $(\sigma[i], \sigma[i_{max}]) \leftarrow (\sigma[i_{max}], \sigma[i])$
        $C_{max} \leftarrow$DSF-UNION($\mathcal{C}, C_{max}, C$)
        if DSF-ROOT-COUNT($\mathcal{C}$) $= 1$ **then**
          return 1
        $\lambda \leftarrow o(\mathcal{I}[\sigma[i]])$
        if $\lambda < k$ **then** $\lambda \leftarrow \lambda + L$
        $\lambda_{max} \leftarrow o(\mathcal{I}[\sigma[i_{max}]])$
        if $\lambda_{max} < k$ **then** $\lambda_{max} \leftarrow \lambda_{max} + L$
        if $\lambda > \lambda_{max}$ **then**
          $i_{max} \leftarrow i$
    $k \leftarrow k + 1, \lambda_e \leftarrow$DSF-ROOT($\Lambda_e, (\lambda_e + 1) \mod L$)
  return DSF-ROOT-COUNT($\mathcal{C}$)

The first step of the SIMPLE-MERGE procedure builds the initial set of cycles from the input matching in $O(n)$. If there is only one cycle the matching is a minimal circuit and we are done. The second step builds the list of interval ends in $O(n)$. The third step is a loop that finds in $O(n)$ the largest gap that contains the location 0. This gap if it exist is the starting point for the next step. The last step is constituted of two nested loops that perform at most n iterations. This loop finds all gaps that intersect the current largest gap and merges the corresponding cycles. Merging cycles is done with a DSF operation which give a complexity for the loop of $O(n \log^*(n))$. Thus the complexity of SIMPLE-MERGE is $O(n \log^*(n))$.



Simple merge — Final merge, when all cycles are unmergeable

Figure 5: Simple and final merge phases of the CYCLE-MERGE algorithm

Definition 22 (final merge) *Let σ be a matching of $\mathcal{I}$ such that there remains no mergeable cycles. The final merge, as described by Figure 5, consists of replacing the largest interval I_{max} by a gap G_{max} such that $G_{max} = [o(I_{max}), e(I_{max}))$:*

FINAL-MERGE$(\mathcal{I}, \sigma)$
  $\{\sigma$ is s.t. there remains no mergeable cycle$\}$
  let I_{max} **such that** $\widehat{c}(I_{max}) = \max_{I_i \in \mathcal{I}}(\widehat{c}(I_i))$
  $o(\sigma(I_{max})) = e(I_{max})$
  $e(I_{max}) = o(I_{max})$
  $\sigma' =$ SIMPLE-MERGE$(\mathcal{I}, \sigma)$
  return σ'

The CYCLE-MERGE algorithm is then a simple succession of simple merges until it converges, followed by a final merge:

Definition 23 (CYCLE-MERGE**)**
CYCLE-MERGE$(\mathcal{I}, \sigma)$
  $\{\sigma$ is an optimal matching$\}$
  $\sigma =$ SIMPLE-MERGE$(\mathcal{I}, \sigma)$
  if σ contains several disjoint cycles
    $\sigma =$ FINAL-MERGE$(\mathcal{I}, \sigma)$
  return σ

Whenever there is no final merge, because the obtained circuit is also an optimal matching it provides an optimal circuit. On the other hand, the final merge adds L to the cost of the optimal matching: fortunately if the obtained circuit is not an optimal matching it is an optimal circuit. Hence the CYCLE-MERGE algorithm provides an optimal solution to the CMETRIC-MH-CIRCUIT problem. Its complexity corresponds to the complexity of the SIMPLE-MERGE algorithm i.e. $O(n \log^*(n))$.

4.2.3 Optimal string with STRING-MERGE

In this paragraph, we will show how CYCLE-MERGE can be used to find an optimal string. Let us consider the set of intervals $\mathcal{I} = \{I_1, \ldots, I_n\}$ and an optimal string $S_{opt} = I_{i_1} I_{i_2} \ldots I_{i_n}$ of length $\widehat{C}(S_{opt}) = k_{opt} L - l_0$. Then let us consider the interval $I_g = [e(I_{i_n}), o(I_{i_1}))$ of length l_0 that corresponds to the gap between the end and the beginning of S_{opt}. Let us add this interval to $\mathcal{I}$ and consider $\sigma =$ CYCLE-MERGE$(\mathcal{I} \bigcup \{I_g\})$ of corresponding cycle $C_{opt} = \circlearrowleft(I_g, \sigma(I_g), \sigma^{(2)}(I_g), \ldots, \sigma^{(n)}(I_g))$ where $\sigma^{(k)}$ represents $\sigma \circ \sigma \circ \cdots \circ \sigma$ composed k times. Because C_{opt} is an optimal cycle for the set $\mathcal{I} \bigcup \{I_g\}$, we have $\widehat{C}(C_{opt}) \leq \widehat{C}\left(\circlearrowleft(I_g, I_{i_1}, \ldots, I_{i_n})\right)$. Hence,

$$\begin{array}{rcccl} \widehat{C}(\sigma(I_g)\sigma^{(2)}(I_g)\ldots\sigma^{(n)}(I_g)) & \leq & \widehat{C}(\circlearrowleft(I_g, \sigma(I_g), \sigma^{(2)}(I_g), \ldots, \sigma^{(n)}(I_g))) - l_0 & = & \widehat{C}(C_{opt}) - l_0 \\ & \leq & \widehat{C}\left(\circlearrowleft(I_g, I_{i_1}, \ldots, I_{i_n})\right) - l_0 & = & \widehat{C}(I_{i_1} I_{i_2} \ldots I_{i_n}) \end{array}$$

In other words, $\sigma(I_g)\sigma^{(2)}(I_g)\ldots\sigma^{(n)}(I_g)$ is an optimal string. Consequently, for all set of cyclic intervals $\mathcal{I}$, there exists a cyclic interval I_g such that for $\sigma =$ CYCLE-MERGE$(\mathcal{I} \bigcup \{I_g\})$, the string $\sigma(I_g)\sigma^{(2)}(I_g)\ldots\sigma^{(n)}(I_g)$ is optimal.

Definition 24 (STRING-MERGE**)**
STRING-MERGE$(\mathcal{I})$
  let $\Lambda_e = \{e(I_i)\}_{I_i \in \mathcal{I}}$ the set of interval end locations
  let $\Lambda_o = \{o(I_i)\}_{I_i \in \mathcal{I}}$ the set of interval origin locations
  let $\mathcal{G} = \{[\lambda_e, \lambda_o)\}_{(\lambda_e, \lambda_o) \in (\Lambda_e, \Lambda_o)}$ the set of possible gaps
  forall $I_g \in \mathcal{G}$
    $\sigma =$ CYCLE-MERGE$(\mathcal{I} \bigcup \{I_g\})$
    $S = \sigma(I_g)\sigma^{(2)}(I_g)\ldots\sigma^{(n)}(I_g)$
    if $\widehat{C}(S) < \widehat{C}(S_{opt})$ **then**
      $S_{opt} = S$
  return S_{opt}

A straightline implementation would test all $|\mathcal{G}| = |\Lambda_e|.|\Lambda_o| \leq L^2$ possible I_g intervals and select among all the shortest string. Fortunately, at most $\Lambda_e + \Lambda_o <= 2L$ intervals have to be tested, and the algorithm 4 runs in $O(L)$ calls to the CYCLE-MERGE procedure.

Algorithm 4

procedure STRING-MERGE($\mathcal{I}$)
  $\sigma \leftarrow$CYCLE-MERGE($\mathcal{I}$)
  $i_{start} \leftarrow$MAX-GAP($\mathcal{I}, \sigma$)
  $S_{opt} \leftarrow \sigma(I_{start})\sigma^2(I_{start}) \dots \sigma^{n-1}(I_{start})I_{start}$
  $G_{opt} \leftarrow [e(\mathcal{I}[i_{start}]), o(\mathcal{I}[\sigma(i_{start})])[$
  $\widehat{C}_{opt} \leftarrow \widehat{C}(\sigma)$
  BUILD-NEAREST-ORIGINS-DSF($\mathcal{I}, \Lambda_o$)
  BUILD-NEAREST-ENDS-DSF($\mathcal{I}, \Lambda_e$)
  $\lambda_e \leftarrow$DSF-ROOT($\Lambda_e, 0$)
  $\lambda_o \leftarrow$DSF-ROOT(Λ_o, λ_e)
  $i \leftarrow 0$
  while $i <$ DSF-ROOT-COUNT(Λ_e) **do**
    $inc_i \leftarrow 0, inc_j \leftarrow 0$
    $I_g \leftarrow [\lambda_e, \lambda_o[$
    if $\widehat{c}(I_g) \leq \widehat{c}(G_{opt})$ **then**
      $inc_j \leftarrow 1$
    else
      $\mathcal{I}' \leftarrow \mathcal{I} \bigcup I_g$
      $\sigma' \leftarrow$CYCLE-MERGE($\mathcal{I}'$)
      if $\widehat{C}(\sigma') = \widehat{C}_{opt}$ **then**
        $S_{opt} \leftarrow \sigma'(I_g)\sigma'^{(2)}(I_g) \dots \sigma'^{(n)}(I_g)$
        $G_{opt} \leftarrow I_g$
        $inc_j \leftarrow 1$
      else
        $inc_i \leftarrow 1$
    if $inc_j = 1$ **then**
      $\lambda'_o \leftarrow$DSF-ROOT($\Lambda_o, (\lambda_o + 1) \mod L$)
      if $\widehat{c}([\lambda_e, \lambda'_o[) \leq \widehat{c}([\lambda_e, \lambda_o[)$ **then**
        $inc_i \leftarrow 1$
      else
        $\lambda_o \leftarrow \lambda'_o$
    if $inc_i = 1$ **then**
      $i \leftarrow i + 1$
      $\lambda_e \leftarrow$DSF-ROOT($\Lambda_e, (\lambda_e + 1) \mod L$)
return S_{opt}

Complexity of STRING-MERGE

The first step of the STRING-MERGE procedure is to find from a CYCLE-MERGE run a first G_{opt} such that $\widehat{C}(S_{opt}) = \widehat{C}(\sigma) - \widehat{c}(G_{opt})$. The main loop iterates over the gap candidates $G = [\lambda_e, \lambda_o)$, for each gap candidates there are three possibilities. If $\widehat{c}(I_g) < \widehat{c}(G_{opt})$ then the string will not be optimal thus we take the next λ_o for testing a larger gap. Otherwise, we run CYCLE-MERGE and if the resulting circuit is still optimal, we've found a better string and we take the next λ_o for testing a larger gap. Otherwise the resulting circuit is not optimal, thus we take the next λ_e. The loop ends when we have exhausted all possible gap start λ_e. The number of iterations of the loop is bounded by $|\Lambda_e| + |\Lambda_o| \leq 2L$. Thus the complexity of STRING-MERGE is $O(L)$ calls to CYCLE-MERGE which gives a global complexity of $O(nLlog^*(n + L))$. Note that for our application, where L the number of cache lines is a constant, the complexity of STRING-MERGE is $O(nlog^*(n))$ thus nearly linear in the number of functions to place.

5 Experimental results

In this section we compare the Pettis and Hansen procedure placement algorithm with the Gloy and Smith algorithm and our solution. We give figures for code size expansion and cache miss rate reduction.

These algorithms have been implemented in the commercial CMG ST200 Compiler based on the OPEN64 compiler. The ST220, an implementation of LX [1] VLIW processor technology designed by Hewlett-Packard and STMicroelectronics, is targeted at system on chip (SoC) solutions. These processors have been developed primarily for compute intensive audio/video applications. Such applications include embedded systems in consumer, digital TV and telecoms markets where there is a specific need for high performance low cost audio/video/data processing. The ST220 is fitted with a 32 Kbyte, direct-mapped, single-level Instruction Cache, that has a large miss penalty. On some applications where no instruction cache optimization have been performed, more than 50% of the cycles are spent waiting for instruction cache refills. This means that the performance of an application can be improved significantly by applying code reordering techniques that minimize instruction cache conflicts. Instruction cache optimizations are not properly part of the compiler, but have been integrated in the toolchain and are accessible in a transparent way for the user via the compiler driver. The algorithms can be directed by profiling information or by a static estimation of the compiler. In this article, experiments were performed using execution trace. The set of benchmarks is first run to generate the execution traces. Then the benchmarks are optimized with the profile informations. The code size figures are computed on the optimized versions. The miss rate figures are given after execution of the optimized programs on a cycle accurate simulator for the ST220 processor. We focus mainly here on the behavior of the algorithm with accurate profile information, thus the second run is done with the same data sets as the run that generated the profile information.

We provide experimental results for the code size expansion of our procedure placement compared to the Pettis and Hansen (PH) algorithm in Figure 6. The code size for PH is normalized to 1, the three other sets are size expansion factor over PH with different values for α as defined in Definition 11. The set labelled $\alpha = 1$ gives the code size expansion when only cache miss cost is minimized, which is the result of the original Gloy and Smith (GS) algorithm. The two other sets are results for two different values of α.

On average the code size expansion of GS is 2.77. With a value of $\alpha = 0.999995$ the code size expansion of our algorithm is 1.08. Obviously adding code expansion in the cost estimate greatly reduce code size. The 1.08 expansion factor compared to PH is mainly due to the fact that we place all procedures at a line cache boundary. This insert empty space between procedures even if our algorithm made a memory placement with no gap.

We also provide results for the cache miss rate compared to PH in Figure 7. The cache miss rate for PH is normalized to 1, the three other sets are the results for our placement, values lower than 1 are better.

On average the miss rate improvement of GS compared to PH is 37%. With a value of $\alpha = 0.999$ our algorithm gives an improvement of 30%. With a value of $\alpha = 0.999995$ our algorithm gives an improvement of 35%. Thus with a value of α close to 1, we mostly have the performance of GS with a minimal code size expansion.

For most of the benchmarks, PH is worse than GS and our algorithm. Some benchmarks though have a better behavior with PH compared to GS (mkinraw, mp2audio, pcmencdec, radial33, testmong, trilinear33). For these benchmarks, the large size expansion of GS compared to PH has a negative impact on performance. Indeed, while GS reduce conflict-misses, code size expansion creates more cold misses and capacity misses as the gaps between procedures cause useless code to be fetched into the cache. Thus it is important to take code size expansion into account.

This also explain why our algorithm is also better than GS on the majority of the benchmarks (13 over 20). Thus we take advantage of both the locality of the placement and an accurate cache conflict metric.

For two benchmarks, mkinraw and mp2audio, our solution with $\alpha = 0.999995$ is worse than PH. Even if our size expansion is low compared to GS, we pay the same negative effect on these

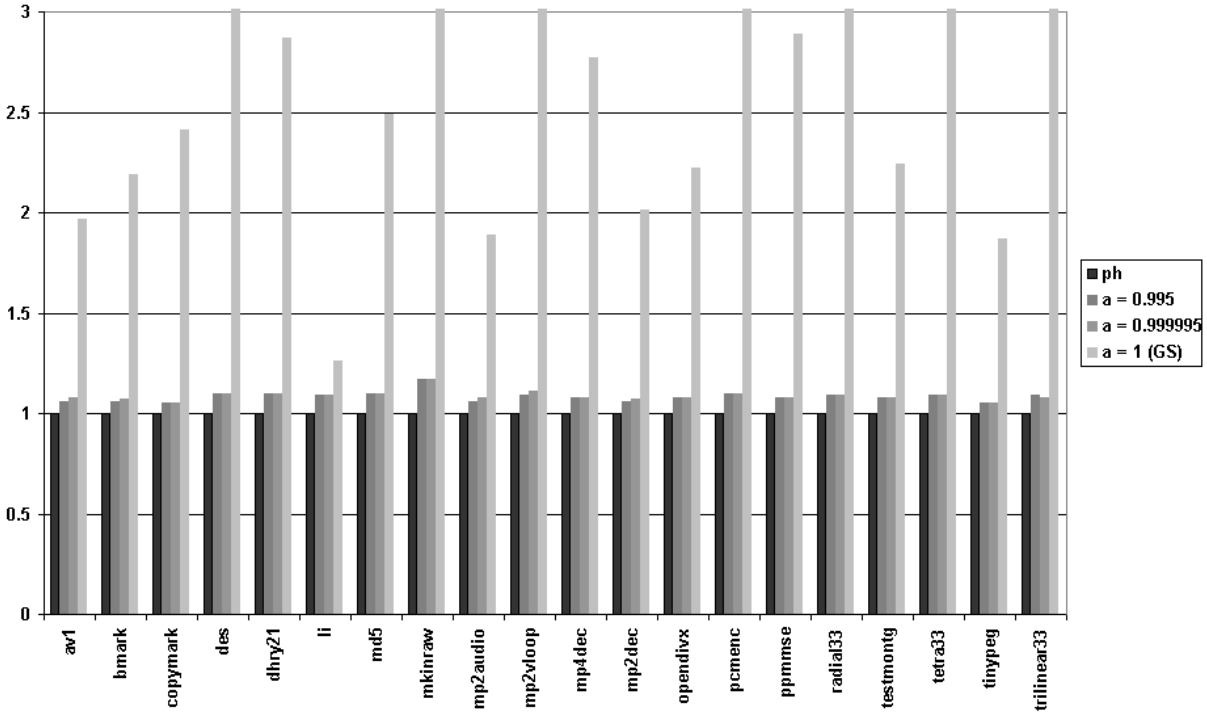


Figure 6: Code size expansion compared to PH with different values of α

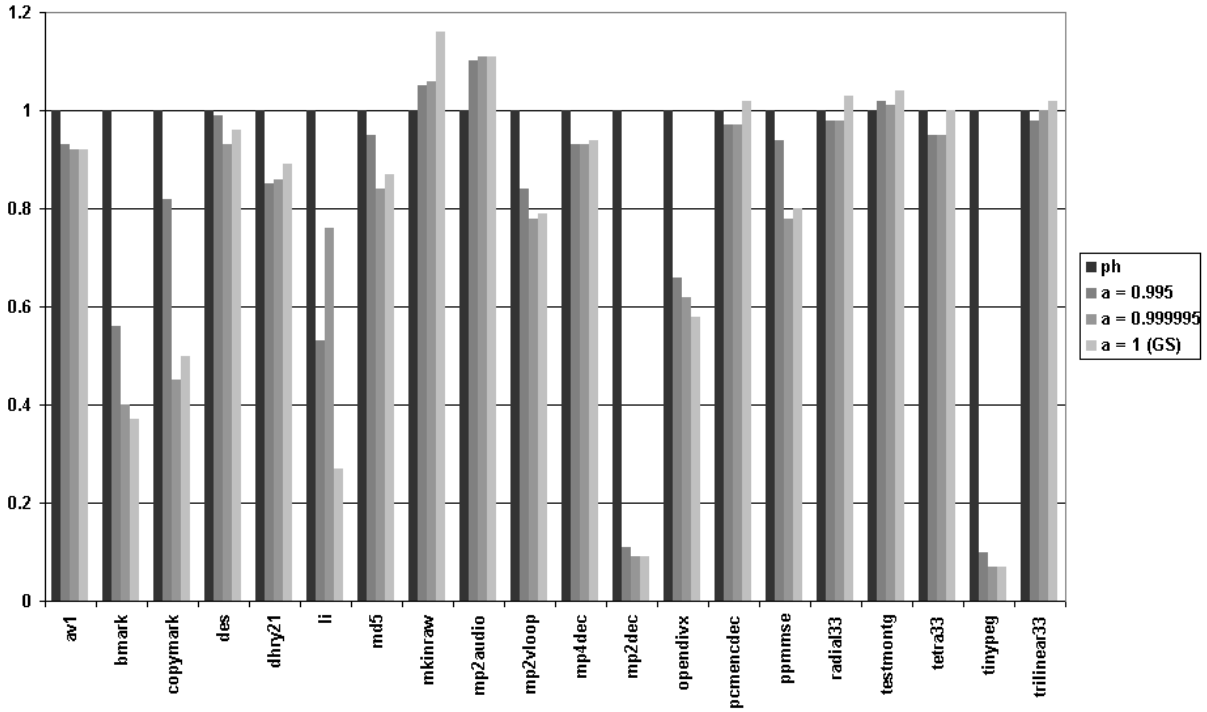


Figure 7: Miss rate compared to PH with different values of α

cases. For the case of mkinraw the program size is only 1.5 times the cache size and most cache misses are cold misses. It explains that we do not improve over the PH algorithm and thus we pay the alignment constraints we have on procedure placement. For the case of mp2audio, we have more conflict misses than PH, this is due to the fact that the processor as an instruction buffer which fetches a number of instructions ahead. Thus when a procedure returns, a number of instructions after the procedure have been fetched into the cache. This behavior is modeled in our cache-placement implementation, however we have only a static estimate of the effective number of instruction fetched ahead and thus our conflict miss metric is not fully accurate.

6 Conclusion

The cache-placement problem as it has been described in this article is a difficult one since it is NP-complete and not even $\alpha(n)$-approximable whatever $\alpha(n)$ is. But there exists efficient algorithms which give satisfactory solutions. The work we have done in this article shows how to reduce code expansion, a major disadvantage of re-allocating the addresses of procedures in the cache-placement stage with only cache miss metrics. First, our algorithm takes into account code size expansion when placing the procedures in the cache-placement phase. Second, our algorithm finds the best allocation for the memory-placement problem: the one which leaves as little empty space as possible between procedures. The results show a code size expansion of 8% over the original program with an improvement of miss rate of 35% over the Pettis and Hansen algorithm, very close to the 37% improvement obtained by the Gloy and Smith algorithm. However, the code size expansion still has a negative impact on performance on some cases. This could be reduced by using an hybrid algorithm that first creates aggregates of functions like the Pettis and Hansen algorithm up to the cache size and then merge the aggregates with our solution.

References

[1] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *The 27th Annual International Symposium on Computer architecture 2000*, pages 203–213, New York, NY, USA, 2000. ACM Press.

[2] Michael R. Garey and Davis S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.

[3] Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):977–1027, 1999.

[4] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–182, 1997.

[5] John Kalamatianos and David R. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 244–253, Las Vegas, Nevada, January 31–February 4, 1998. IEEE Computer Society TCCA.

[6] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimisation, Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, 1982.

[7] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6):16–27, June 1990.

Appendix A: optimality of the String-Merge algorithm

This part is devoted to the proofs of optimality of the String-Merge algorithm. Because our solution to the Min-String problem uses the solution to a particular Hamiltonian-Circuit problem that itself uses the solution to a particular Min-Matching problem, the remaining of this part is decomposed as follow: Section 6.1 exposes our solution to the particular Min-Matching problem; Section 6.2 exposes our solution to the particular Hamiltonian-Circuit problem based on fusion of cycles;

6.1 Optimal matching with Matching-Greedy

The goal of this section is to show that Matching-Greedy gives an optimal solution to our matching problem. The main ideas of the proof are roughly the following: we define a distance between matchings; we consider a matching, obtained using the Matching-Greedy algorithm, and consider for our distance the closest optimal matching; we show that we can modify this optimal matchings without increasing its cost, thus keeping the optimality, such that the resulting matching becomes strictly closest; because the distance is an interger, this proves the optimality of our "greedy" matching.

6.1.1 Distance, pre-order and precedence-choice graph

The goal of this paragraph is to introduce preliminary notions and results useful for the proof of theorems 1 and 2 given further. In particular, Definition 27 sets the definition of a precedence-choice graph for a matching and Lemma 2 enlarges this definition to a precedence-choice graph for a permutation.

Definition 25 (pre-order) *A pre-order p is a non complete order. If a has a successor b, it is written $a \prec_p b$. If $P = (V, E)$ is an oriented acyclic graph, there is a unique pre-order associated to P: $a \prec_P b \iff (a, b) \in E$. To simplify, we mingle a pre-order with its given corresponding acyclic graph.*

Definition 26 (linear extension) *A linear extension $\overline{p}$ of a pre-order p is an order which verifies $a \prec_p b \implies a <_{\overline{p}} b$.*

Definition 27 (distance) *Let π and π' be two orderings. If $\pi = \pi'$ then the distance* from π to π' *is null, $d(\pi, \pi) = 0$. Otherwise, if $\pi \neq \pi'$ let us decompose π and order' into $\pi = (UaV)$ and $\pi' = (UWbaZ)$ such that a and b are intervals and U,V,W, and Z are strings of intervals possibly empty. Denote by $|S| = n$ the length of the string $S = s_1 s_2 \dots s_n$. Then the distance* from π to π' *is $|V|n + |Wb|$.*

Definition 28 (Precedence-choice graph) *Let σ be a matching. The precedence-choice graph $P_\sigma = (V, E)$ is the following oriented graph:*

- *V is the set of intervals*
- *$(a, b) \in E \iff \widehat{c}_\epsilon(b, \sigma(a)) < \widehat{c}_\epsilon(b, \sigma(b))$. In other words, if $\pi = \overline{P_\sigma}$, considering a greedy matching* Matching-Greedy(π)*, $(a, b) \in E$ relates the fact that $a <_\pi b$ is a necessarily condition to get the couples $(b, \sigma(b))$ and $(a, \sigma(a))$ (see Figure 8).*

The two following lemmas state that the precedence-choice graph of an optimal matching and the precedence-choice graph obtained from Matching-Greedy are both acyclic.

Lemma 1 *Let σ be an optimal matching, then the corresponding precedence-choice graph P_σ is acyclic.*

Lemma 2 *Let π be an order and $\sigma_\pi =$* Matching-Greedy(π)*. Then the precedence-choice graph P_{σ_π} is acyclic. For simplicity we mingle the notion of precedence-choice graph of a permutation π with the precedence-choice graph of its corresponding matching σ_π*

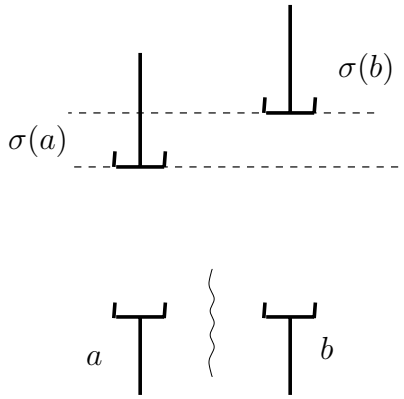


Figure 8: The matching of a and b depends on the ordering of a and b. Here $a <_\pi b$.

Proof of Lemma 1 Let us prove the lemma by contradiction: let us consider a cycle $(a_1, a_2, \dots a_k = a_1)$ of P_σ, then the matching σ' where every couple $(a_i, \sigma(a_i))$ is replaced by $(a_i, \sigma'(a_i) = \sigma(a_{i-1}))$ has a strictly smaller cost than σ. This contradicts the hypothesis that σ is optimal. ■

Proof of Lemma 2 To prove the lemma, we simply prove that

$$(a, b) \in E \Longrightarrow a <_\pi b$$

Indeed, by contradiction $b <_\pi a$ and $\widehat{c}_\epsilon(b, \sigma_\pi(a)) < \widehat{c}_\epsilon(b, \sigma_\pi(b))$ is absurd because MATCHING-GREEDY cannot allocate b to $\sigma_\pi(b)$ if $\sigma_\pi(a)$ is not yet allocated to a i.e. available. ■

The following lemma provides a transformation on permutations that can be performed without modifying its cost.

Lemma 3 *Let π be a permutation and P_π the precedence-choice graph of π. Let a and b such that $a \prec_{P_\pi} b$. Consider the modified permutation $\pi' = \tau_{ab} \circ \pi$ where a and b have been simply transposed*[3]*. Then,*

$$\sigma_{\pi'} = \sigma_\pi \circ \tau_{ab} \tag{1}$$
$$\widehat{C}(\pi') = \widehat{C}(\pi) \tag{2}$$

Proof of Lemma 3 Let us denote U, V and W the strings of intervals (possibly empty) respectively before, between and after (a, b) in π. Hence, $\pi = (UaVbW)$ and $\pi' = (UbVaW)$. Let us also denote the string $\sigma(a_1)\sigma(a_2)\dots\sigma(a_m)$ by $\sigma(a_1 a_2 \dots a_m)$.

By applying MATCHING-GREEDY on both permutations, we trivially obtain $\sigma_\pi(U) = \sigma_{\pi'}(U)$, then $\sigma_\pi(Ua) = \sigma_{\pi'}(Ub)$, then $\sigma_\pi(UaV) = \sigma_{\pi'}(UbV)$, $\sigma_\pi(UaVb) = \sigma_{\pi'}(UbVa)$ and eventually $\sigma_\pi(\pi) = \sigma_{\pi'}(\pi')$. Point (1) follows.

Now, let us figure out the equality (2) $\widehat{C}(\pi) - \widehat{C}(\pi') = 0$. First, from $\widehat{c}_\epsilon(a, \sigma_\pi(a)) < \widehat{c}_\epsilon(a, \sigma_\pi(b))$ we get $o(\sigma_\pi(a)) \in [e(a), o(\sigma_\pi(b))] = G_{ab}$ and from $\widehat{c}_\epsilon(b, \sigma_\pi(a)) < \widehat{c}_\epsilon(b, \sigma_\pi(b))$ we get $o(\sigma_\pi(a)) \in [e(b), o(\sigma_\pi(b))] = G_{bb}$. Now denote by $\perp$ the bottom of $G_{ab} \cup G_{bb} = G_{\perp b}$ i.e. $e(a)$ if $G_{bb} \subset G_{ab}$ and $e(b)$ otherwise. Then, $\forall x \in \{e(a), e(b), o(\sigma_\pi(a)), o(\sigma_\pi(b)))\}$, $x \in G_{\perp b}$.

Hence, for all $s \in \{a, b\}$ and $t \in \{\sigma_\pi(a), \sigma_\pi(b)\}$,

$$\begin{aligned} \widehat{c}(s,t) = & \quad (o(t) - \perp) \mod L \\ - & \quad (e(s) - \perp) \mod L \end{aligned} \tag{3}$$

Finally, from (1), we get $\widehat{C}(\pi) - \widehat{C}(\pi') = \widehat{c}(a, \sigma_\pi(a)) + \widehat{c}(b, \sigma_\pi(b)) - \widehat{c}(a, \sigma_\pi(b)) - \widehat{c}(b, \sigma_\pi(a))$ which simplifies to 0 thanks to equation (3). ■

[3] τ_{ab} is the permutation that transposes a and b. $\circ$ is the composition operator, $g \circ f(x) = g(f(x))$.

6.1.2 Optimality of MATCHING-GREEDY

Theorems given in this section show a bijection between the set of precedence-choice graph for all permutations and the set of all optimal matching. In particular it states the optimality of the MATCHING-GREEDY algorithm.

Theorem 1 *For each ordering π, then $\sigma =$ MATCHING-GREEDY(π) is an optimal matching.*

The following lemma is necessary for the proof of Theorem 1. It states that each matching obtained from the MATCHING-GREEDY algorithm is characterized by its precedence-choice graph .

Theorem 2 *Let σ be an optimal matching, p_σ its corresponding precedence-choice graph pre-order. Then, for each linear extension $\overline{p_\sigma}$ of p_σ,* MATCHING-GREEDY$(\overline{p_\sigma}) = \sigma$.

Lemma 4 *Let π be a permutation, and $\sigma_\pi =$* MATCHING-GREEDY(π). *Let p be the pre-order corresponding to the precedence-choice graph P_{σ_π}.*

Then for each linear extension $\overline{p}$ of p, MATCHING-GREEDY$(\overline{p}) = \sigma_\pi$.

Proof of Theorem 2 Let σ be an optimal matching, $p_\sigma = (V, E)$ the corresponding precedence-choice graph pre-order, $\pi = \overline{p_\sigma}$ a linear extension of p_σ and $\sigma' =$ MATCHING-GREEDY(π).

By construction, there exists no interval I such that $I \prec_{p_\sigma} I_{\pi(1)}$. Hence, $I_k = \sigma(I_{\pi(1)})$ minimizes the value $\widehat{c}_\epsilon(I_{\pi(1)}, I_k)$ and $\sigma(I_{\pi(1)}) = \sigma'(I_{\pi(1)})$. Then, by recurrence on π, suppose that $\forall i \in \{1, 2, \ldots, m-1\}$, $\sigma(I_{\pi(i)}) = \sigma'(I_{\pi(i)})$. Consider, $I_k = \sigma(I_{\pi(j)})$ such that $\widehat{c}_\epsilon(I_{\pi(m)}, I_k) < \widehat{c}_\epsilon(I_{\pi(m)}, \sigma(I_{\pi(m)}))$, then by definition of p_σ, $I_{\pi(j)} \prec_{p_\sigma} I_{\pi(m)}$ and $j < m$. Hence $\sigma(I_{\pi(m)})$ minimizes $\widehat{c}_\epsilon(I_{\pi(m)}, I_k)$ over all the remaining intervals. In other words, $\sigma(I_{\pi(m)}) = \sigma'(I_{\pi(m)})$.

Eventually, $\sigma = \sigma'$. ■

Proof of Lemma 4 By contradiction: so let $\sigma' =$ MATCHING-GREEDY$(\overline{p})$ with $\overline{p}$ a linear extension of p and let us suppose that $\exists a \mid \sigma(a) \neq \sigma'(a)$. We take a as small as possible for $\overline{p}$. Then, from the definition of $\widehat{c}_\epsilon$, $\widehat{c}_\epsilon(a, \sigma(a)) \neq \widehat{c}_\epsilon(a, \sigma'(a))$.

If $\widehat{c}_\epsilon(a, \sigma'(a)) > \widehat{c}_\epsilon(a, \sigma(a))$ then it means that there exists b such that $\sigma'(b) = \sigma(a)$, and $b <_{\overline{p}} a$. As $\sigma(b) \neq \sigma(a)$, $\sigma(b) \neq \sigma'(b)$, which is contradictory to the fact that a was the smallest.

If $\widehat{c}_\epsilon(a, \sigma'(a)) < \widehat{c}_\epsilon(a, \sigma(a))$. As a didn't choose $\sigma'(a)$ with π, it means that there exists b such that $b <_\pi a$ and $\sigma(b) = \sigma'(a)$. Hence $\widehat{c}_\epsilon(a, \sigma(b)) < \widehat{c}_\epsilon(a, \sigma(a))$ so $b \prec_p a$. But $a <_{\overline{p}} b$ and $\overline{p}$ is not a linear extension of p. ■

Proof of Theorem 1 Let π be a permutation, $\sigma_\pi =$ MATCHING-GREEDY(π). Consider also an optimal coupling σ_{opt} and π_{opt} its corresponding ordering π_{opt} such that MATCHING-GREEDY$(\pi_{opt}) = \sigma_{opt}$ (Theorem 2). Among all possible optimal matchings, let us consider one that minimizes $d(\pi, \pi_{opt})$. Let us show by contradiction that $d(\pi, \pi_{opt}) = 0$. So, suppose that $\pi \neq \pi_{opt}$. We can write $\pi = (UaV)$ and $\pi_{opt} = (UWbaZ)$. Consider $\pi'_{opt} = (UWabZ) = \tau_{ab} \circ \pi$ where a and b have been swapped. From Lemma 3, $\widehat{C}(\pi_{opt}) = \widehat{C}(\pi'_{opt})$, so MATCHING-GREEDY$(\pi'_{opt}) = \sigma'_{opt}$ is still optimal. But, $d(\pi, \pi'_{opt}) = d(\pi, \pi_{opt}) - 1$ which contradicts the minimality of the distance to π. Hence, $\pi_{opt} = \pi$ and $\sigma = \sigma_{opt}$ is an optimal matching. ■

6.2 Optimal circuit with CYCLE-MERGE

The goal of this section is to prove the optimality of the algorithm CYCLE-MERGE. Theorem 3 states that the simple merge process does not degrade the matching. Theorem 4 states that when no more simple merge can be performed, remaining cycles can be merged using a final merge process that degrades the solution by an additional constant L. Theorem 5 states that whatever ordering has been used by the MATCHING-GREEDY algorithm it leads to the same connected components thus to a non (simple) mergeable solution. This proves the optimality of the obtained circuit.

Theorem 3 (simple merge) *Let σ be a matching of $\mathcal{I}$, σ' be the matching after a simple merge. Then $\widehat{C}(\sigma') \leq \widehat{C}(\sigma)$. In particular if σ is an optimal matching then σ' is also optimal.*

Proof of Theorem 3 Because $\overline{G_i} \bigcap \overline{G_j} \neq \emptyset$ either $e(I_i) \subset \overline{G_j}$ or $e(I_j) \subset \overline{G_i}$. Suppose for convenience that $e(I_j) \in \overline{G_i}$. Then, there are three possibilities as described in Figure 9. Let us denote by $G'_i = (\vdash_{\sigma'} .I_i) = [e(I_i), o(\sigma(I_j)))$ and $G'_j = (\vdash_{\sigma'} .I_j) = [e(I_j), o(\sigma(I_i)))$. The reader can easily check that for each cases, $\widehat{c}(G'_i) + \widehat{c}(G'_j) \leq \widehat{c}(G_i) + \widehat{c}(G_j)$ which proves the theorem. ■

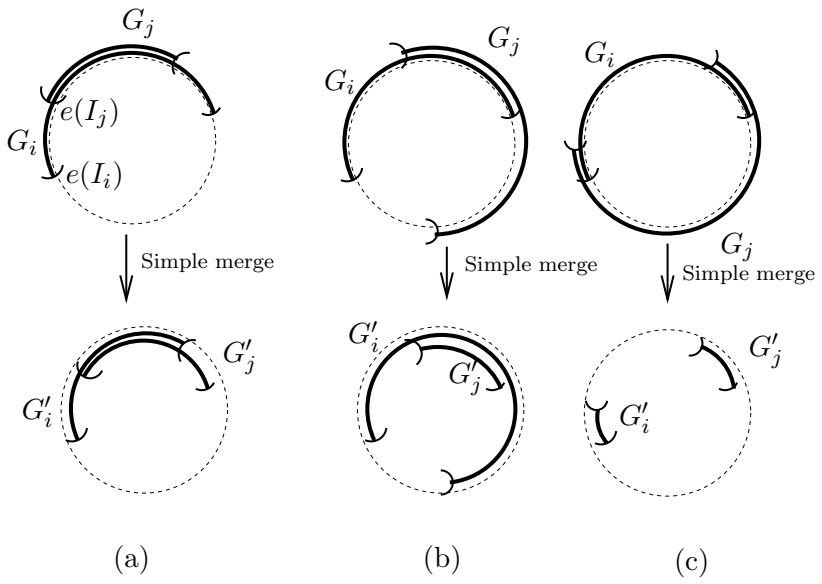


Figure 9: Three different schemes when $\overline{G_i} = [[e(I_i), o(\sigma(I_i))]$ and $\overline{G_j} = [[e(I_j), o(\sigma(I_j))]$ intersect. In cases (a) and (b), $\widehat{c}(G'_i) + \widehat{c}(G'_j) = \widehat{c}(G_i) + \widehat{c}(G_j)$; in case (c), $\widehat{c}(G'_i) + \widehat{c}(G'_j) = \widehat{c}(G_i) + \widehat{c}(G_j) - L$.

Theorem 4 (final merge) *Let σ be a matching of $\mathcal{I}$ such that there remains no mergeable cycles. Let I_{max} be the largest interval of $\mathcal{I}$: $\widehat{c}(I_{max}) = \max_{I_i \in \mathcal{I}}(\widehat{c}(I_i))$.*
Let $C = \circlearrowleft(I_{max}, \sigma(I_{max}), \dots, \sigma^{(m-1)}(I_{max}))$ with $\sigma^{(m)}(I_{max}) = I_{max}$. Suppose $m \neq n$ i.e. there remains other disjoint cycles. Then, all other cycle C' of σ contains a gap included in I_{max} and the final merge merges all cycles into a unique cycle of cost $\widehat{C}(\sigma) + L$.

Proof of Theorem 4 Let $C' = \circlearrowleft(I_{i_1}, I_{i_2}, \dots)$ be a cycle of σ disjoint to C. Because C and C' are unmergeable, all gaps from C' are either contained in I_{max} or in $[0, L)_{\setminus I_{max}}$. Suppose all gaps of C' are outside of I_{max}, then C' would contain an interval that includes I_{max} which contradicts its maximality. Hence, C' contains a gap included in I_{max}.

The fact that, once I_{max} has been replaced by G_{max}, the simple merge process converges to a unique cycle is due to the two following facts:

- C now contains a large gap G_{max} that intersect at least one gap from each other cycle.
- Whenever σ is an optimal coupling, the union of the union of gaps of two cycles is equal to the union of the gaps of the simple merged cycle of those two cycles. Hence, at each step, all remaining disjoint cycle can be simple merged to C because the union of the gaps from C still contains the interval I_{max}.

■

Theorem 5 *Let σ be an optimal matching of $\mathcal{I}$, then $\sigma' = \textsc{Cycle-Merge}(\mathcal{I}, \sigma)$ is an optimal Hamiltonian circuit.*

Proof of Theorem 5 The case where no final merge has been performed is straightforward: every circuit is a particular matching so our obtained circuit which is an optimal matching is also an optimal circuit. The hard point is when a final merge has been performed. The idea is to prove that in that case all optimal matching necessarily contains several disjoint cycles. The consequence is that there exists no Hamiltonian circuit of cost $\widehat{C}(\sigma)$. Then because the cost of a matching is a multiple of L, $\widehat{C}(\sigma') = \widehat{C}(\sigma) + L$ is minimal which will proves the theorem.

Hence, let us consider that σ contains at least two disjoint cycles say C and C' and prove that for all optimal matching σ_{opt} we have $[\forall I \in C,\, \sigma_{opt}(I) \notin C']$. A nice consequence is that all optimal matching have the same connected components. To prove this, let us consider $I \in C$ and σ_{opt}. Because C and C' are not mergeable, there exists $I' \in C'$ such that $e(I) \in I'$. Now consider the set of intervals from $\mathcal{I}_{\setminus C'}$ that ends (respectively begins) in I': $\mathcal{I}_e = \{J \in \mathcal{I}_{\setminus C'}, e(J) \in I'\}$ (resp. $\mathcal{I}_o = \{J \in \mathcal{I}_{\setminus C'}, o(J) \in I'\}$). Because C' is not mergeable with $\mathcal{I}_{\setminus C'}$, all gaps that begin in I' end in I' and $\sigma(\mathcal{I}_e) = \mathcal{I}_o$. So, $\mathbf{card}\mathcal{I}_e = \mathbf{card}\mathcal{I}_o$. Now to conclude, remind Theorem 2 that states that all optimal matching is a "greedy" matching, and clearly in a greedy process $\sigma_{opt}(\mathcal{I}_e) = \mathcal{I}_o$. In particular $\sigma_{opt}(I) \in \mathcal{I}_{\setminus C'}$. ■

Appendix B: NP-completeness of the cache-placement problem

The goal of this appendix is to prove Theorem 6.

Theorem 6 MIN-CACHE-MISS *is NP-complete.*

Proof of Theorem 6

1. The problem is in NP : given a solution, the number of cache misses is calculated in $O(T)$ with T the size of the trace.
2. The reduction is from GRAPH K-COLORABILITY [2].

Hence, let us consider a graph $G = (V, E)$ with $V = \{x_1, x_2, \dots x_n\}$.
Also, let $G' = (V \bigcup \{p\}, E \bigcup (\bigcup_{i=1}^{n} \{v, x_i\}))$.

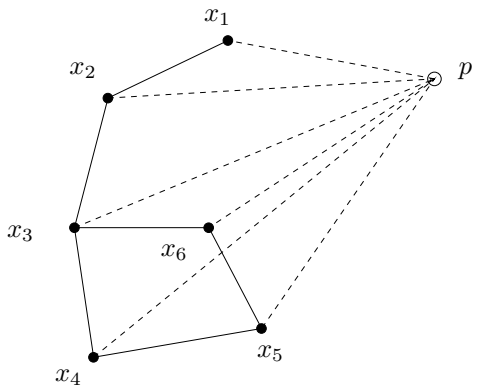


Figure 10: A example of graph G and the corresponding graph G'

Now, let P be the following program :

```
Program P
  do i = 1, n − 1
    do j = i + 1, n
      if (x_i, x_j) ∈ E then
        do tmp = 1, cardE + 2
          call X_i
          call X_j
```

Where $X_1, \cdots, X_n$ are n functions of size $l(P)$. Let us take $l(P) = 1$ for convenience.

The trace of P (unique in this case) is

$$\mathcal{T} = P \prod_{(x_i,x_j)\in E} (X_i P X_j P)^{\mathbf{card}E+2}$$

Also, G' is k-colorable if and only if G is $(k-1)$-colorable.

Finally, because the MIN-CACHE-MISSinstance for P is linear with $\mathbf{card}E$ (no exponential growth...), Lemma 5 proves the theorem. ■

Lemma 5 *G' is k-colorable if and only if the minimum number of cache misses for P with a cache of size k is less than $2\mathbf{card}E$.*

Proof of Lemma 5

First, we prove that if G' is k-colorable then we can build an allocation o of cost $\widehat{c}(o)$ less than $2\mathbf{card}E$.

Let $c : V \to \{0, \ldots, k-1\}$ be a coloration of G'. Our allocation is a direct mapping of the coloration: $o(P) = c(p)$ and for each i, $o(X_i) = c(x_i)$.

Now,

- because p is connected to every vertex, no other procedure than P has the 0 offset, so there is no cache miss with P,
- because two neighbor vertices have a different color, there is also no cache misses in the do loop indexed by *tmp*.

Hence, the only possible cache misses are when there is a transition of edge in the program. For each transition point, the maximum number of cache misses is two: when the two newer procedures are not loaded.

Consequently, there is less than $2\mathbf{card}E$ cache misses.

Secondly, we prove by contradiction that if the minimum allocation cost is less than $2\mathbf{card}E$ then G' is k-colorable. So suppose that G' is not k-colorable. Let o be an optimal allocation of cost $\widehat{\mathrm{Miss}}_{\mathcal{T}}(o) \leq 2\mathbf{card}E$. Let $c : v \to o(P); x_i \to o(X_i)$. Then there exists $(x_i, x_j) \in E$ so that $c(x_i) = c(x_j)$. Then $o(X_i) = o(X_j)$: it means that the trace $(X_i P X_j P)^{\mathbf{card}E+2}$ will create $2(\mathbf{card}E + 2) - 2$ conflict-cache misses , which is greater than $2\mathbf{card}E$. ■

Appendix C: Algorithmic support

Algorithm 5
procedure BUILD-INTERVAL-ORIGINS-RDL($\mathcal{I}, \mathcal{L}$)
  foreach $i \leftarrow (1, \ldots, n)$ **do**
    $\lambda_o \leftarrow o(\mathcal{I}[i])$
    RDL-INSERT($\mathcal{L}, \lambda_o, i$)
  return

Algorithm 6
procedure BUILD-INTERVAL-ENDS-RDL($\mathcal{I}, \mathcal{L}$)
  foreach $i \leftarrow (1, \ldots, n)$ **do**
    $\lambda_e \leftarrow e(\mathcal{I}[i])$
    RDL-INSERT($\mathcal{L}, \lambda_e, i$)
  return

Algorithm 7
procedure BUILD-NEAREST-ORIGINS-DSF($\mathcal{I}, \Lambda$)
  foreach $i \leftarrow (1, \ldots, n)$ **do**
    $\lambda \leftarrow o(\mathcal{I}[i])$
    DSF-INSERT(Λ, λ)
  $\lambda_n \leftarrow \lambda$
  $k \leftarrow (\lambda_n - 1) \mod L$
  while $k \neq \lambda_n$ **do**
    if *not* DSF-CONTAINS(Λ, k) **then**
      DSF-INSERT(Λ, k)
      $\lambda \leftarrow$DSF-UNION(Λ, λ, k)
    else
      $\lambda \leftarrow k$
    $k \leftarrow (k - 1) \mod L$
  return

Algorithm 8
procedure BUILD-CYCLES-DSF($\sigma, \mathcal{C}$)
  foreach $i \leftarrow (1, \ldots, n)$ **do**
  if *not* DSF-CONTAINS($\mathcal{C}, i$) **then**
    DSF-INSERT($\mathcal{C}, i$)
    $C \leftarrow i$
    $j \leftarrow \sigma[i]$
    while *not* DSF-CONTAINS($\mathcal{C}, j$) **then**
      DSF-INSERT($\mathcal{C}, j$)
      $C \leftarrow$DSF-UNION($\mathcal{C}, C, j$)
      $j \leftarrow \sigma[j]$
  return

Algorithm 9
procedure FINAL-MERGE($\mathcal{I}, \sigma$)
  $\widehat{c}_{max} \leftarrow -1$
  for $i \leftarrow (0, \ldots, n)$ **then**
    if $\widehat{c}(\mathcal{I}[i]) > \widehat{c}_{max}$ **then**
      $\widehat{c}_{max} \leftarrow \widehat{c}(\mathcal{I}[i])$
      $i_{max} \leftarrow i$
  $\mathcal{I}' \leftarrow \mathcal{I}$
  $\mathcal{I}'[i_{max}] \leftarrow [o(\mathcal{I}[i_{max}]), o(\mathcal{I}[i_{max}])[$
  SIMPLE-MERGE($\mathcal{I}', \sigma$)

Algorithm 10

procedure CYCLE-MERGE($\mathcal{I}$)
  $\sigma \leftarrow$ ARRAY-NEW(n)
  MATCHING-GREEDY($\mathcal{I}, \sigma$)
  $N_{cycles} \leftarrow$ SIMPLE-MERGE($\mathcal{I}, \sigma$)
  if $N_{cycles} > 1$ **then**
    FINAL-MERGE($\mathcal{I}, \sigma$)
  return σ