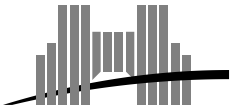
Laboratoire de l'Informatique du Parallélisme

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

On the complexity of loop fusion

Alain Darte October 1998

Research Report N° RR1998-50



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

On the complexity of loop fusion

Alain Darte

October 1998

Abstract

Loop fusion is a program transformation that combines several loops into one. It is used in parallelizing compilers mainly for increasing the granularity of loops and for improving data reuse. The goal of this report is to study, from a theoretical point of view, several variants of the loop fusion problem – identifying polynomially solvable cases and NP-complete cases – and to make the link between these problems and some scheduling problems that arise from completely different areas. We study, among others, the fusion of loops of different types, and the fusion of loops when combined with loop shifting.

Keywords: Parallelization, loop fusion, loop distribution, complexity

Résumé

La fusion de boucles est une transformation de programme qui combine plusieurs boucles en une seule. Elle est utilisée dans les compilateurs-parallèliseurs, principalement pour augmenter la granularité des boucles et pour améliorer la réutilisation des données. Le but de ce rapport est d'étudier d'un point de vue théorique plusieurs variantes du problème de fusion de boucles – en identifiant les cas solubles en temps polynomial et les cas NP-complets – et d'établir le lien entre ces problèmes et quelques problèmes d'ordonnancement provenant de domaines complètement différents. Nous étudions notamment le problème de la fusion de boucles typées ainsi que le problème de la fusion de boucles avec décalage.

Mots-clés: Parallélisation, fusion de boucles, distribution de boucles, complexité

On the complexity of loop fusion

Alain Darte
E-mail: Alain.Darte@ens-lyon.fr

October 1998

Abstract

Loop fusion is a program transformation that combines several loops into one. It is used in parallelizing compilers mainly for increasing the granularity of loops and for improving data reuse. The goal of this report is to study, from a theoretical point of view, several variants of the loop fusion problem – identifying polynomially solvable cases and NP-complete cases – and to make the link between these problems and some scheduling problems that arise from completely different areas. We study, among others, the fusion of loops of different types, and the fusion of loops when combined with loop shifting.

1 Introduction

Loop fusion is a program transformation that collapses several loops into one. The resulting program compaction and the corresponding increase in the size of the loop body has several well-known impacts on the performances of a program [15]. It was first used to reduce the cost of loop bound testing. It can also have a significant impact on memory performance (registers or cache) since it may put closer in time several variable reuses. Another interest is the reduction of synchronizations when loops are to be distributed among different computation units. Loop fusion has also an indirect impact on performance due to the fact that many useful optimizations are limited to basic blocks or perfectly nested loops: increasing the size of the loop body gives more chance for common subexpression elimination, instruction scheduling and software pipelining, nested loop optimizations, etc.

Loop fusion is not always legal since it may change the behavior of the program by inverting the execution order of dependent computations. The analysis of *data dependences* specifies when the fusion is allowed. Furthermore, even if loop fusion is legal from an execution point of view, some loops may not be fused because they have a different *type*, for example if they have different headers (lower and upper bounds, steps), or because they are going to be executed in different manner (sequential or parallel loops). The typed loop fusion problem, introduced by McKinley and Kennedy [7], is to fuse typed loops, while respecting dependences, so as to obtain a program with as few loops as possible, thereby achieving maximal code compaction. Other objectives and other frameworks have been studied, especially synchronization minimization [3] (it is not exactly a loop fusion problem but it is

closely related), (non-typed) loop fusion for array contraction [5], loop fusion for maximal reuse [10], loop fusion with loop shifting [2], etc.

The goal of this paper is to make a summary on the complexity of loop fusion problems, mainly by showing that it has a strong relationship with the shortest common supersequence (SCS) problem and with several scheduling problems. Thanks to this scheduling view on the loop fusion problem, we give algorithms for polynomial cases that are conceptually simpler than those proposed in the past, and we give an answer to open questions on the complexity of loop fusion, most of them having been already solved for the SCS problem, which is a very close problem. In particular, McKinley and Kennedy conjectured the typed fusion problem to be polynomially solvable for a fixed number of types, but it is actually NP-complete starting from two types if some dependences may prevent fusion, starting from three types otherwise, and polynomially solvable in all other cases.

The paper is organized as follows. In Section 2, we give some examples where the loop fusion problem arises and we state the loop fusion problem formally. In Section 3, we show in a unique framework how polynomially solvable cases can be solved by a simple graph traversal. In Section 4, we summarize NP-complete results for loop fusion alone. Section 5 is devoted to the problem of loop fusion combined with loop shifting. This simple association makes the problem immediately NP-complete. Conclusions are presented in Section 6.

2 Loop fusion: examples and problem definition

We first give some examples to introduce various loop fusion problems.

2.1 Partial loop distribution and maximal parallelism

Let us try to maximally parallelize the following code (Example 1), i.e. to place each statement in a parallel loop whenever this is possible, by performing loop distribution (the inverse of loop fusion).

Example 1

```
DO I=1,N
  A(I) = 2*A(I) + 1
  B(I) = C(I-1) + A(I)
  C(I) = C(I-1) + G(I)
  D(I) = D(I-1) + A(I) + C(I-1)
  E(I) = E(I-1) + B(I)
  F(I) = D(I) + B(I-1)
ENDDO
```

□

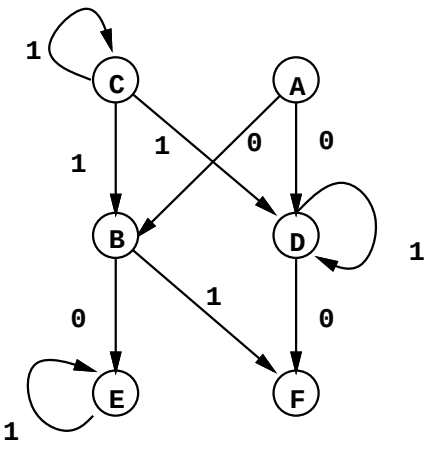


Figure 1: Dependence graph for Example 1.

The technique, proposed by Allen and Kennedy [1], is the following. First, data dependences between statements are computed: dependences are separated into dependences that occur between different iterations of the loop (called *loop carried* dependences) and dependences that occur inside the loop body (called *loop independent* dependences). The dependence graph for Example 1 is given in Figure 1, loop independent dependences are labeled 0, loop carried dependences are labeled 1.

Then, strongly connected components (SCCs) of the dependence graph are computed. Each SCC corresponds in the parallelized code to a separated loop and this loop is marked parallel if and only if the SCC does not contain any loop carried dependence. Finally, all SCCs are totally ordered following the dependences between SCCs and the parallelized code is generated following this total order. The code obtained by this procedure is given in Figure 2, for one particular total order. In this example, each vertex belongs to its own SCC, and loops that compute arrays A, B and F are parallel, other loops are sequential. But, how can we find the program in Figure 3 that exposes the same parallelism but uses the fewest number of loops? This problem is a particular case of the *typed fusion problem* that we address in Proposition 4. We call it the *partial distribution problem.*

```
DOPAR I=1,N
  A(I) = 2*A(I) + 1
ENDDOPAR
DOSEQ I=1,N
  C(I) = C(I-1) + G(I)
ENDDOSEQ
DOPAR I=1,N
  B(I) = C(I-1) + A(I)
ENDDOPAR
DOSEQ I=1,N
  E(I) = E(I-1) + B(I)
ENDDOSEQ
DOSEQ I=1,N
  D(I) = D(I-1) + A(I) + C(I-1)
ENDDOSEQ
DOPAR I=1,N
  F(I) = D(I) + B(I-1)
ENDDOPAR
```

Figure 2: Parallelized code with maximal loop distribution.

```
DOSEQ I=1,N
  C(I) = C(I-1) + G(I)
ENDDOSEQ
DOPAR I=1,N
  A(I) = 2*A(I) + 1
  B(I) = C(I-1) + A(I)
ENDDOPAR
DOSEQ I=1,N
  D(I) = D(I-1) + A(I) + C(I-1)
  E(I) = E(I-1) + B(I)
ENDDOSEQ
DOPAR I=1,N
  F(I) = D(I) + B(I-1)
ENDDOPAR
```

Figure 3: Parallelized code with partial loop distribution.

2.2 Minimization of synchronizations

Suppose now that the previous code obtained after maximal loop distribution is executed in such a way that all sequential loops are executed by the same processor and that parallel loops are cut among different processes at run-time. Where do we have to insert synchronization

barriers so that the resulting code is guaranteed to be correct at run-time? How many synchronization barriers do we have to insert? In the previous example, the best solution provided by Callahan's algorithm (that we recall in Section 3) is given in Figure 4. Only two barriers are needed. This is the *synchonization minimization problem*.

2.3 Loop fusion combined with loop shifting

Now suppose that we want to combine loop fusion and loop shifting (which consists in moving statements by a few iterations) so as to perform maximal fusion, while preserving parallel loops. Starting from the code of Figure 2, we can also derive the code of Figure 5. It is another solution with four loops (the code of Figure 3 is a first solution) but in which we shifted F by one iteration so that it can now fuse with B. (In this example, the shift transformed the loop carried dependence between B and F into a loop independent dependence). In general, we can fuse more loops with this combination, but how can we find the minimal number of loops? This problem is addressed in Section 5. It is NP-complete.

```
DOSEQ I=1,N
  C(I) = C(I-1) + G(I)
ENDDOSEQ
DOPAR I=1,N
  A(I) = 2*A(I) + 1
ENDDOPAR
   BARSYNC
DOPAR I=1,N
  B(I) = C(I-1) + A(I)
ENDDOPAR
DOSEQ I=1,N
  D(I) = D(I-1) + A(I) + C(I-1)
ENDDOSEQ
   BARSYNC
DOSEQ I=1,N
  E(I) = E(I-1) + B(I)
ENDDOSEQ
DOPAR I=1,N
  F(I) = D(I) + B(I-1)
ENDDOPAR
```

Figure 4: Code with synchronization barriers.

```
DOPAR I=1,N
  A(I) = 2*A(I) + 1
ENDDOPAR
DOSEQ I=1,N
  C(I) = C(I-1) + G(I)
  D(I) = D(I-1) + A(I) + C(I-1)
ENDDOSEQ
DOPAR I=0,N
  IF (I>0) B(I) = C(I-1) + A(I)
  IF (I<N) F(I+1) = D(I+1) + B(I)
ENDDOPAR
DOSEQ I=1,N
  E(I) = E(I-1) + B(I)
ENDDOSEQ
```

Figure 5: Code obtained by combination of loop fusion and loop shifting.

2.4 Problem formulation

We can now formulate the problem more formally. Once dependence analysis has been performed and potential fusions have been identified, the relations between loops are represented by a directed acyclic graph $G = (V, E = F \cup \overline{F}, T)$ in which each loop of the program

corresponds to a vertex $v \in V$ of the graph. The mapping T from V to a set of types $\mathcal{T}$ specifies the type $T(v)$ of a vertex v. Edges in E are classified in *precedence* edges (edges in F) and *fusion-preventing* edges (edges in $\overline{F}$). For simplifying the statements of the results presented hereafter, all fusion-preventing edges $e = (u, v)$ in $\overline{F}$ are supposed to be such that $T(u) = T(v)$.

A *fusion partition* is a partition of V into disjoint *clusters*: each cluster represents a set of loops to be fused. The fused graph G_f is the graph induced by the partition of V: there is an edge from a cluster c_1 to a cluster c_2 if there is an edge $e = (u, v) \in E$ such that $u \in c_1$ and $v \in c_2$. A fusion partition is *legal* if and only if the three following conditions are satisfied:

Type constraint Two vertices of different types can not belong to the same cluster.

Fusion-preventing constraint Two vertices connected by an edge in $\overline{F}$ can not belong to the same cluster.

Precedence constraint The fused graph is acyclic.

Remarks:

- the first constraint explains why we chose to call fusion-preventing edges only edges that link two vertices of the same type: anyhow two vertices of different types can not be fused.
- the third constraint guarantees that the fused loops can be executed in some order and that code generation is feasible. Figure 6(d) illustrates what may happen without this constraint. No execution of loops is feasible with the last partition.

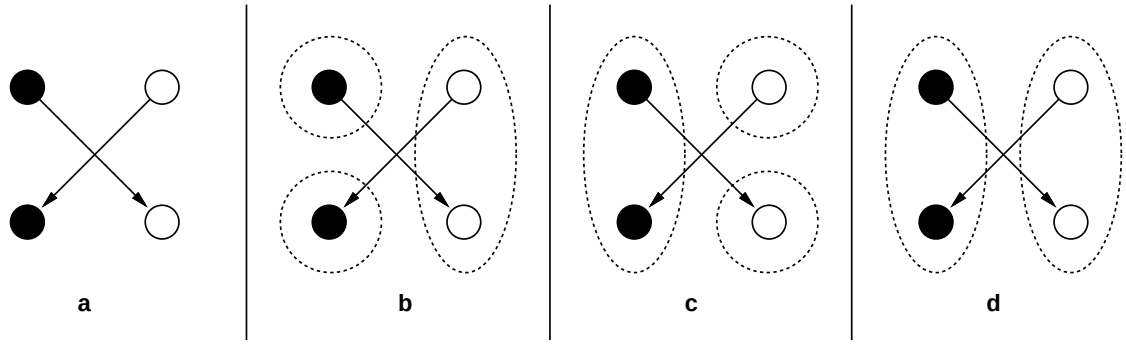


Figure 6: (a) original graph, (b) and (c) two legal partitions, (d) a cyclic partition (illegal).

All fusion problems consist in finding a legal fusion partition that optimizes some given criterion: the main problem addressed in this paper is the *typed fusion problem* in which the objective is to minimize the number of clusters, thereby achieving maximal fusion.

Loop fusion can also be formulated as a *scheduling* problem. Indeed, since we are looking for clusters that induce an acyclic graph, we can directly search for a totally ordered set of

clusters. We are thus looking for a mapping C from V to the non negative integers $\mathbb{N}$ such that the following conditions are satisfied:

Type constraint For all $v \in V$ and $u \in V$, $T(v) \neq T(u) \Rightarrow C(v) \neq C(u)$ (1)

Precedence constraint For each edge $e = (u, v) \in F$, $C(v) \geq C(u)$ (2)

Fusion-preventing constraint For each edge $e = (u, v) \in \overline{F}$, $C(v) \geq C(u) + 1$ (3)

The typed fusion problem is to find a mapping C for which $\max\{C(v) \mid v \in V\}$ is minimal.

When there is no fusion-preventing edges, this formulation is nothing but a single machine problem: some tasks (the vertices $v \in V$) have to be performed on a single machine in some configuration (the type of v): the goal is to minimize the number of switches from one configuration to another one. Such a problem was addressed in [8]. For example, vertices can be communicated processes that run on one machine, the goal is to minimize the number of context-switches. Another example is a traveling salesman who has to perform different operations in some partial order (the graph G) in different towns (the types), and who wants to minimize the number of moves between towns.

When all edges between two vertices of same type are fusion-preventing, and when the graph is a set of chains, the typed fusion problem is nothing but the shortest common supersequence (SCS) problem [6] whose complexity has been widely studied, each chain being interpreted as a string from the alphabet of types.

The typed fusion problem is an intermediate problem: some edges may or may not be fusion-preventing.

3 Simple loop fusion: polynomially solvable cases

All polynomially solvable loop fusion subproblems can be solved in the same framework: with one traversal of a graph whose edges e have a weight $w(e)$, we can compute for each vertex v the maximal weight $W(v)$ of a path directed to v. Depending on the problem, we will give a different meaning to the value $W(v)$. We will also define in different ways the weights of edges and the weights of paths of length 0 (and sometimes the graph itself), but in all cases, we will then use an algorithm of the form:

```
TRAVERSAL(G = (V,E,w)) {
  1. Initialize W(v) for vertices v with no predecessor.
  2. For all vertices v in topological order do:
        W(v) = Max{W(u) + w(e) where e = (u,v)}
}
```

which can be implemented in $O(|V| + |E|)$ steps.

3.1 Loop fusion with a single type

Here, the graph G is a graph $G = (V, E = F \cup \overline{F}, T)$ where T is a singleton. Therefore, there is no type constraint. Only the precedence and the fusion-preventing constraints have to be considered (Equations (2) and (3)), which makes the problem obvious. Initialize $W(v)$ to 0

if v has no predecessor in G and let $w(e) = 1$ if $e \in \overline{F}$, and $w(e) = 0$ otherwise. The value $W(v)$ computed by the algorithm TRAVERSAL gives the first cluster in which the vertex v can be placed. In other words, loops can always been maximally fused in a greedy way, i.e. as soon as possible with a scheduling terminology.

3.2 Synchronization minimization

The synchronization minimization problem is not exactly a fusion problem, but it is so similar to the loop fusion problem with a single type that we mention it here. A greedy approach is also clearly optimal, as it was observed by Callahan [3]. For this problem, the graph $G = (V, E, T)$ is a DAG with two types S (for sequential) and P (for parallel) but the goal is not to fuse loops (so, there is actually no type constraint as Equation (1)), but the goal is to place synchronization barriers between loops, assuming that parallel loops are going to be distributed into different processes while all sequential loops are going to be executed by the same processor.

Denote by $W(v)$ the minimal number of synchronization barriers that are required before the execution of the loop corresponding to v. If v has no predecessor, then $W(v) = 0$. Now consider an edge $e = (u, v)$. If $T(u) = T(v) = S$, then no barrier is required between u and v since v will be anyhow executed serially after u: we simply have to ensure that $W(v) \geq W(u)$, thus we let $w(e) = 0$. In all other cases, an additional barrier is required, $W(v) \geq W(u) + 1$ and we let $w(e) = 1$. Computing W for each vertex is done using one graph traversal as before. Loops are generated by increasing value of W and one barrier is generated each time W is incremented. The overall complexity is $O(|V| + |E|)$.

Back to Example 1

For deriving the code of Figure 4, we consider the graph of Figure 7, parallel vertices are in black, sequential vertices are in white, integers close to edges are the weights w. $W(C)$ and $W(A)$ are first set to 0, then $W(B) = W(D) = 1$, and finally $W(E) = W(F) = 2$. □

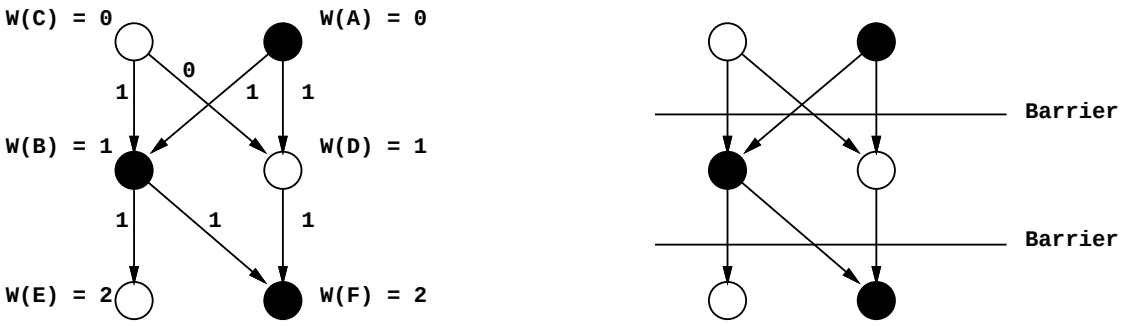


Figure 7: Graph for synchronization minimization.

Other execution models are possible: for example, if sequential loops are not guaranteed to be executed by the same processor, then we have to let $w(e) = 1$ for $e = (u, v)$ even if $T(u) = T(v) = S$. On the contrary, if we can guarantee that each pair of dependent

iterations of two consecutive parallel loops will be executed by the same processor, then we can let $w(e) = 0$ (loops could be fused). This situation can appear in some cases when generating code with the owner-computes rule [12].

3.3 Loop fusion with two types and no fusion-preventing edges

Suppose that $G = (V, E = F \cup \overline{F}, T)$ is such that $T = \{S, P\}$ (only two types) and $\overline{F}$ is empty (no fusion-preventing edges). Then, any valid optimal fusion partition leads to a total order on clusters of the form $SPSPSP...$ or $PSPSPS...$. The best solution is thus either the best solution that starts with an S, or the best solution that starts with a P.

Let us try to find the best solution that starts with an S. We first find a lower bound for the cluster number $W(v)$ in which v can be placed. If v has no predecessor, $W(v) = 0$ if $T(v) = S$ and $W(v) = 1$ if $T(v) = P$ since we look for a solution that starts with an S. For an edge $e = (u, v)$ with $T(u) \neq T(v)$, the type constraint imposes $W(v) \geq W(u) + 1$, we thus let $w(e) = 1$. Otherwise, since there are no fusion-preventing edges, we just let $w(e) = 0$. The algorithm TRAVERSAL computes W in $O(|V| + |E|)$ steps. Furthermore, $W(v)$ is even when $T(v) = S$, and $W(v)$ is odd when $T(v) = P$. Thus W is a valid total order on clusters since it satisfies both the type constraint (1) (thanks to the parity property) and the precedence constraint (2): the lower bound is therefore a valid solution.

Finding the best solution that starts with a P is done symmetrically. The best solution (with no restriction on the starting type) is thus found by two calls of complexity $O(|V|+|E|)$.

Example 2

All examples given in [10] and [7] are examples with two types and no fusion-preventing edges, they can thus be optimized in $O(|V| + |E|)$ operations. The main example in [10] is depicted and solved in Figure 8, numbers close to vertices (resp. edges) are the value of W (resp. w), clusters are in dashed curves. The optimal solution that starts with an S (white vertices) has 4 clusters, the optimal solution that starts with a P (black vertices) has 5 clusters. □

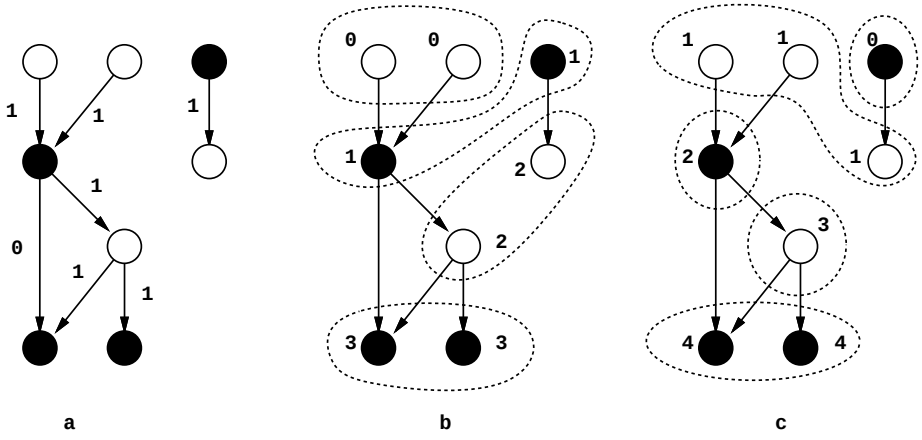


Figure 8: (a) original graph for Example 2, (b) starting with white, (c) starting with black.

3.4 Ordered typed fusion

In order to approximate the general typed fusion problem, McKinley and Kennedy proposed a heuristic called the *ordered typed fusion*. The principle is to first maximally fuse loops of a given type, then to maximally fuse loops of a second type taking into account what has been done for the first type, etc. This can also be done using the algorithm TRAVERSAL as we now explain.

This time, let us try to find the smallest possible number of clusters for the type T. As before, we look for a lower bound for the cluster number $W(v)$ in which a vertex v of type T can be placed. For a vertex u of type $T' \neq T$, the meaning of $W(u)$ is the minimal number of clusters of type T that have to be placed *before* u.

If v has no predecessor, we let $W(v) = 0$: indeed, either $T(v) = T$ and v can be placed in the first cluster of type T, or $T(v) \neq T$ and no cluster of type T is required before v. For an edge $e = (u, v)$ with $T(u) = T$, we let $w(e) = 1$ if $e \in \overline{F}$ (fusion-preventing constraint), or if $T(v) \neq T$ (type constraint). In all other cases, we let $w(e) = 0$ since we count only the number of clusters of type T. The algorithm TRAVERSAL computes W in $O(|V| + |E|)$ steps. Then, we fuse all vertices v of type T with the same value $W(v)$. This can also be done in $O(|V| + |E|)$. The resulting graph is acyclic since all edges are directed in increasing values of W: it is thus a valid solution and it requires the minimal number of clusters of type T since it is equal to the lower bound W.

Then, the same technique is applied for the second type and so on. The resulting overall complexity is $O(|\mathcal{T}|(|V| + |E|))$ where $|\mathcal{T}|$ is the number of types.

Back to Example 1

Consider Example 1 again. It has two types, only one edge is fusion-preventing (marked with a slash), see Figure 9(a). The solution in Figure 9(b) (resp. Figure 9(c)) is the solution when black (resp. white) is first maximally fused. Figure 10 (resp. Figure 11) depicts the two steps when fusing black (resp. white) vertices first. The solution obtained in Figure 10 corresponds to the code of Figure 3. □

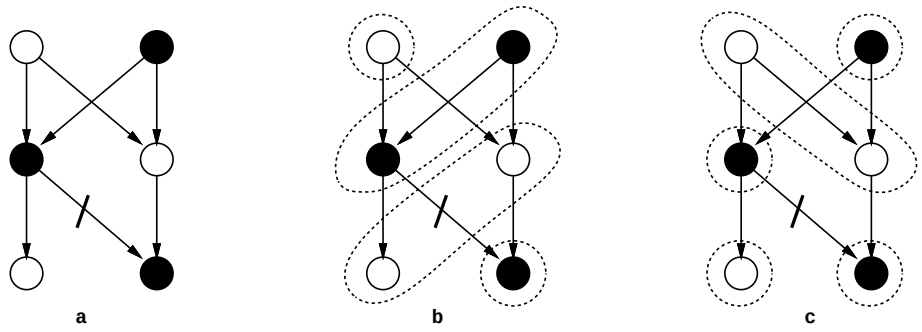


Figure 9: (a) original graph, (b) solution when fusing black then white, (c) solution when fusing white then black.

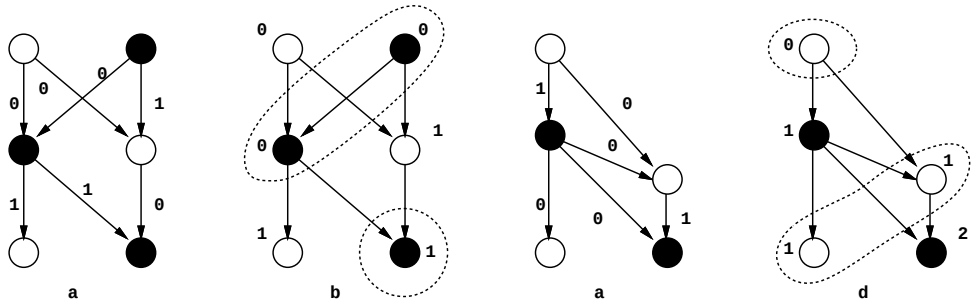


Figure 10: (a) weighted graph for fusing black, (b) fusion of black, (c) weighted graph for fusing white after black, (d) fusion of white.

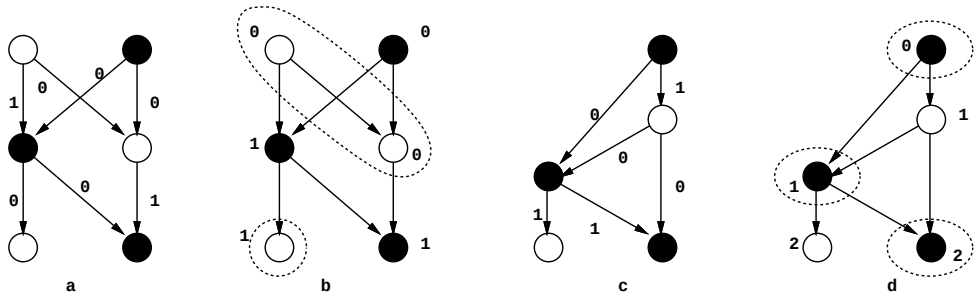


Figure 11: (a) weighted graph for fusing white, (b) fusion of white, (c) weighted graph for fusing black after white, (d) fusion of black.

Note that another simple technique can be used to find an approximation to the general typed fusion problem. Let $w(e) = 1$ for each $e = (u, v)$ such that $T(u) \neq T(v)$ or $e \in \overline{F}$, and $w(e) = 0$ otherwise, initialize W to 1 for vertices with no predecessor, and compute W by the algorithm TRAVERSAL. We find $c = \max\{W(v) \mid v \in V\}$ clusters, in each cluster all vertices of different types are not related, and all vertices of the same type can fuse. This leads to a legal partition with at most $c|\mathcal{T}|$ clusters. Furthermore, c is a lower bound for the total number of clusters. Finding a heuristic with performance ratio $|\mathcal{T}|$ is thus straightforward.

3.5 Typed fusion for a bounded number of chains

When the graph $G = (V, E = F \cup \overline{F}, T)$ is a set of d chains, then for each chain, we can first fuse all successive vertices of same type that are linked by a precedence edge (i.e. not fusion-preventing). We can thus assume, without loss of generality, that all edges between vertices of same type are fusion-preventing. In this case, the typed fusion problem is now exactly the shortest common supersequence (SCS) problem. The SCS gives the totally ordered set of clusters and the correspondence between each chain and the SCS specifies in which cluster a vertex can take place.

It is well-known that the SCS problem in this case can be solved in $O(2^d \prod_{i=1}^{d} r_i)$ steps by dynamic programming, where r_i is the length of the i-th chain and d is the number of chains. To say it differently, we can describe all possible solutions by a graph in a d-dimensional space where each vector $(x_1, \ldots, x_d)$ with $0 \leq x_i \leq r_i$ corresponds to the fact that the first x_i

vertices of the i-th chain have found their place in the SCS, and each edge (at most 2^d edges leave each vertex) specifies how we can progress along chains. Then, the maximal fusion (or the shortest common supersequence) is obtained by computing the shortest path from the vector $(0, \ldots, 0)$ to the vector $(r_1, \ldots, r_d)$, once again by one graph traversal. Figure 12 illustrates this technique for two chains. In this example, the shortest path is unique (using three diagonal edges), so is the maximal fusion. The search can also be restricted to solutions that progress maximally on each chain. In this case, some edges are superfluous and the complexity reduces to $O(d \prod_{i=1}^{d} r_i)$ (at most d edges leave each vertex).

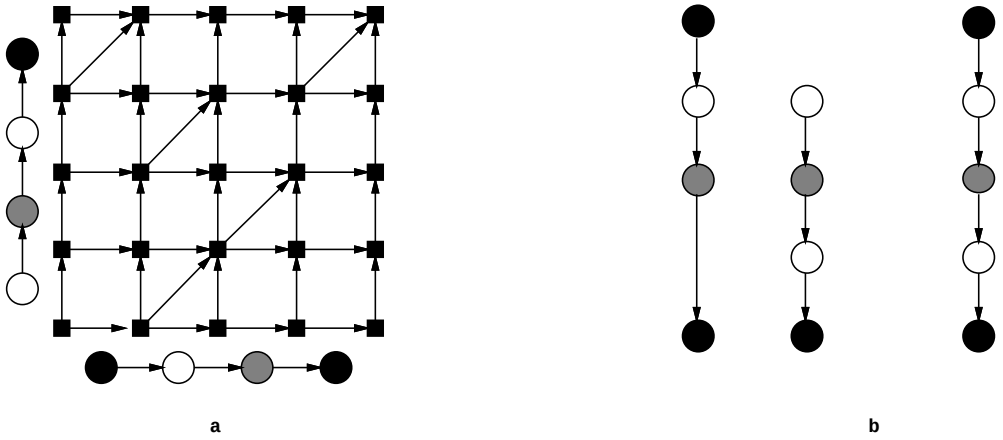


Figure 12: (a) the 2-dimensional graph, (b) the shortest common supersequence.

4 Typed loop fusion: NP-complete results

In [10], McKinley and Kennedy considered the problem of parallel and sequential code generation, i.e. a typed fusion problem with two types. To solve this problem, they proposed the ordered typed fusion (that we addressed in Section 3.4), but they pointed out it was not an optimal approach in general. In [7], they generalized the typed fusion problem to an arbitrary number of types, which finds its application in the fusion of loops with different headers. By a reduction from the Vertex Cover problem, they proved the following:

Proposition 1 *The typed fusion problem, i.e. the problem of maximal fusion with typed loops, is NP-complete if the number of types is not fixed.*

As they noticed, the number of types required by their reduction is equal to the number of vertices in the Vertex Cover problem. Thus, Proposition 1 does not answer the initial fusion problem of parallel and sequential loops. They conjectured the existence of a polynomial-time algorithm for two types. We now prove that this is unfortunately not true.

Proposition 2 *The problem of maximal fusion, in the presence of fusion-preventing edges, is NP-complete starting from two types (and even for chains).*

Proof Suppose that T is a set with two types $\{S, P\}$ and that the graph $G = (V, E = F \cup \overline{F}, T)$ is a set of chains such that all edges between two vertices of same type are fusion-preventing. Then the typed fusion problem is exactly the SCS problem on a binary alphabet, which has been proved NP-complete by Räihä and Ukkonen [14]. Each chain can indeed be considered as a string on the binary alphabet $\{0 = S, 1 = P\}$. ■

The proof in [14] is also a reduction from the Vertex Cover problem. It requires blocks of 0 and blocks on 1 on each string. Therefore, this result does not address the typed fusion problem when there are no fusion-preventing edges. As we explained in Section 3.3, when there are no fusion-preventing edges and only two types, the problem is trivially solvable in polynomial time. In [8], the problem is a scheduling problem that corresponds to the typed fusion problem with no fusion-preventing edges. By replacing each letter x by the substring $x\overline{x}$ where $\overline{x}$ is a new letter, Lofgren and al. easily showed that the problem is NP-complete for $2n$ types if the SCS problem is NP-complete for n letters. At this time, they use the reduction of Maier [9] for which $n \geq 5$, concluding that the problem is NP-complete for 10 types. With the result of Räihä and Ukkonen, the same technique shows that the problem is NP-complete for 4 types. The following proposition addresses the problem with 3 types. (In [8], Lofgren and al. mentioned also that the problem was NP-complete for 3 types but no proof was provided.)

Proposition 3 *The problem of maximal fusion, with no fusion-preventing edges, is NP-complete starting from three types (even for chains).*

Proof The proof in [14] is 10 pages long. We just give here, for the curious reader, the modifications that are made compared to the original proof. (The proof presented here can not be understood without the original one.) What we have to prove is that all modified strings required in [14] for the reduction from Vertex Cover are strings, from an alphabet with three letters, for which consecutive letters are different (so that no fusion-preventing edge is required). In [14], t (resp. r) is the number of vertices (resp. edges) in the Vertex Cover instance.

In [14], each string is a concatenation of substrings labeled E (for edge) and N (for node). Each substring labeled N corresponds to $(t + 1)$ blocks of 1, sometimes separated by one 0, and each substring labeled E corresponds to $(r + 1)$ blocks of 0, sometimes separated by two 1. The key point is to notice that the separation by only one 1 is sufficient. We can then modify the substrings in the following way, using a new letter $\overline{a}$:

- each block of 1 is replaced by a block of $1\overline{a}$.
- each block of 0 is replaced by a block of $0\overline{a}$.

Doing so, no string can contain two identical consecutive letters. Now, let us check that the arguments of the proof are exactly the same.

To make the proof checking simpler, we assume that in each substring (labeled E or N), the number of blocks is the same, equal to $c + 1$ where $c = \max(t, r)$. Furthermore, the number of pairs ($1\overline{a}$ or $0\overline{a}$) in each block is Kc (instead of $7c$ in [14], we will then choose $K = 4$). Using the notations of [14], q is now equal to $2Kc(4c + 3c + 7) + (2c + 2c)$ and

if there is a vertex cover of size k, there is a SCS of size $q + r + k$. We now consider the modifications that have to be made on the seven successive lemmas in [14]. Using Räihä and Ukkonen terminology, a solution has less than $q + r + k$ *threads*, i.e. less than $r + k < 2c$ *extra threads* (it is assumed $k < t$).

Lemma 1 The block $\overline{N}[j]\overline{N}[m]$ contains $2Kc(c+1)$ ones, and there are only $Kc(c+1)+2c$ ones between $\overline{N}_s^L$ and $\overline{N}_s^R$, thus at least $Kc(c+1) - 2c \geq 2c$ extra threads, a contradiction if $K \geq 2$ and $c \geq 1$.

Lemma 2 First, if the one in $\overline{E}[i]^L$ is not to the left of $\overline{N}_s^L$, then there remains at least one block of $0\overline{a}$ of $\overline{E}[i]^L$ to schedule "during" $\overline{N}_s^L$, thus at least $Kc - c \geq 2c$ extra threads for scheduling the zeros of this block, a contradiction if $K \geq 3$.

Then, if $\overline{E}[i]^L$ has zeros and $\overline{a}$ that are not to the left of $\overline{N}_s^L$, then an extra thread is used to schedule the one in $\overline{E}[i]^L$ and establish a correspondence between $\overline{E}[i]^L$ and $\overline{E}^L$. It remains to check what happens if we remove the thread that previously scheduled the one. We now count the number of zeros until the end of $\overline{E}[i]^L$: using the notations of [14], there are $Kc(c + 1 - i)$ zeros in S_i after the one, and Kch in S_h before the one, to compare with the $Kc(c+1)+c$ ones in $\overline{E}^L\overline{N}_s^L$: if $h > i$, then $Kc(h-i)-c \geq 2c$, a contradiction if $K \geq 3$.

Lemma 4 To prove (i), when the zero is to the right of $\overline{N}$, we count the number of ones to the right of the thread θ: $Kc(c + 1) + c$ ones in T and at least $Kc(c + 1) + Kc + 1$ ones in S_i, thus at least $Kc + 1 - c \geq 2c$ extra threads, a contradiction if $K \geq 3$.

To prove (ii), we first check that $\overline{N}[j']$ is the rightmost N-block of $S_{i'}$. Otherwise, the number of ones between θ and θ_3 is at least $Kc(c+1)+Kc$ for $S_{i'}$ and at most $Kc(c+1)+c$ for T, thus there are at least $Kc - c \geq 2c$ extra threads, a contradiction if $K \geq 3$. Then, if $j < j'$, there are $Kc(c + 1 + j')$ ones between θ_1 and θ for $S_{i'}$, and $Kc(c + 1 + c + 1 - j)$ ones between θ and θ_4 for S_i, thus at least $3Kc(c + 1) + Kc$ ones, while there are at most $3Kc(c + 1) + 2c$ for T, a contradiction when $K \geq 4$. If $j > j'$, then there are at least $Kc(c + 1) + Kc$ ones between θ_2 and θ_3 for S_i and $S_{i'}$, but only $Kc(c + 1) + 2c$ for T, same conclusion. This study was when $\overline{E}[i']^R$ is not to the right of E_s^R. Otherwise, counting the ones between the first of θ_1 and θ_2 to the last of θ_3 and θ_4 leads to $2Kc(c + 1) + Kc$ for the two strings S_i and $S_{i'}$, but only $2Kc(c + 1) + 2c$ for T.

Lemma 6 If the extra thread for the one in $\overline{E}[i]^L$ is shared, then there is at least one block of zeros scheduled by extra threads to the left of $\overline{N}_s^L$, thus at least $Kc \geq 2c$ extra threads, a contradiction if $K \geq 2$.

Lemma 7 Only one extra thread is needed to build Θ_4 from Θ_3.

Lemmas 3 **and** 5 No change. ■

Propositions 1, 2, and 3 still do not provide an answer to one of the most important problems in practice: the partial distribution problem presented in Section 2.1. Indeed,

when parallel loops and sequential loops come from the distribution of a single loop, then sequential loops can always be fused back, just by retrieving the original semantics of the program. Therefore, the problem of generating a code with as few loops as possible, and in which each statement is in a parallel loop when possible, is a typed fusion problem, with two types, but for which fusion-preventing edges can occur only between vertices of one the two types (namely the parallel type). We thus need to refine again the result of Proposition 2 since the proof in [14] requires consecutive letters for both types. For that, we use a more recent proof of the NP-completeness of the SCS problem proposed by Middendorf [11].

Proposition 4 *The problem of maximal fusion, in the presence of fusion-preventing edges, is NP-complete starting from two types even if only one type is concerned with fusion-preventing edges (even for chains).*

Proof The reader can check that the proof of Theorem 3.6 in [11] requires only strings with consecutive zeros and three non consecutive ones. For the partial distribution problem, the zeros correspond thus to parallel loops with fusion-preventing edges between them, and the ones correspond to sequential loops. ■

5 Loop fusion combined with loop shifting

We now recall the complexity of the fusion of parallel loops with uniform dependences when combined with loop shifting, problem which was introduced in [2]. Loop shifting is a program transformation that consists in moving a statement along the iterations of a loop that surrounds it. This combination finds its applications in parallelizing algorithms that use "shifted-linear" schedules [4]. For example, the code in Figure 5 was obtained by shifting (backwards) the statement that computes the array F.

The problem is stated as follows. We are given a directed graph $G = (V, E, w)$, where each vertex $v \in V$ corresponds to a parallel loop, and where each edge e has a weight $w(e)$ (the dependence distance). An edge e is fusion-preventing if the dependence is not loop independent (i.e. $w(e) \neq 0$). If loop shifting is not considered, the problem of maximal fusion is trivially polynomially solvable as explained in Section 3.1: define $\overline{G} = (V, E, \overline{w})$ where $\overline{w}(e) = 0$ if $w(e) = 0$ and $\overline{w}(e) = 1$ otherwise, then the minimal number of loops after fusion is one plus the maximal weight of a path in $\overline{G}$. When loop shifting is considered, we are allowed to shift each vertex v by $\rho(v)$ iterations so as to modify the dependence distance and hope to fuse more loops. After the shift ρ, an edge $e = (u, v)$ has a weight $w_\rho(e) = w(e) + \rho(v) - \rho(u)$: if $w_\rho(e)$ is now 0, the edge is not fusion-preventing anymore. The problem is thus to find a shift ρ from V to $\mathbb{Z}$ such that the maximal weight of a path in the graph $\overline{G_\rho}$ is minimized. This problem has been proved (strongly) NP-complete in [2].

Proposition 5 *The problem of maximal fusion of parallel loops with uniform dependences is strongly NP-complete when combined with loop shifting.*

Proof The proof is by reduction of the fusion problem from the UET-UCT scheduling problem (Unitary Execution Time - Unitary Communication Time) that was proved NP-complete in [13]. ■

An integer linear programming (ILP) formulation has also been proposed in [2] to solve exactly the problem. The technique is to decouple the shift problem and the fusion problem: the fusion problem is solved by ILP with additional constraints that guarantee a feasible shift. Then, the shift itself is found by a simple graph traversal.

6 Conclusion

The first motivation of this paper was to characterize the complexity of the typed fusion problem with a fixed number of types, problem that was conjectured polynomially solvable for two types by McKinley and Kennedy [7]. We have shown that this was unfortunately not the case. We have extended this result to a subcase that occurs very frequently in practice, the problem of partial distribution, which is a typed fusion problem with two types (parallel and sequential) where the sequential type can always fuse. This subproblem is also NP-complete. Loop fusion is thus a difficult problem even for simple objectives such as the minimization of the the total number of loops or the maximization of data reuse [10].

Variants of polynomially solvable cases seem also difficult, as we illustrate with the problem of maximal fusion of a single type combined with loop shifting. Other extensions that have a practical interest remain to be considered: for example, minimizing the number of synchronization barriers is polynomially solvable, but what is the complexity of the typed fusion problem when restricting to solutions that require the minimal number of synchronization barriers? Another example is an optimized variant of the ordered typed fusion problem: finding the minimal number of loops for a given type is polynomially solvable, but how can we pick, among all solutions that minimize the number of loops for this type, a solution for which a second type will be minimized? This is not exactly the (unordered) typed fusion problem since there are examples for which the total number of loops is minimized by a solution for which none of the types is independently minimized.

The consequence of this study is that loop fusion is hard (at least in theory). From a practical point of view, this justifies the use of an exhaustive evaluation of solutions when dealing with small code portions: in this case, more accurate performance models can then be used for picking the right solution. However, when a compiler has to perform such an optimization on larger codes, or even on small portions but too often, then heuristics have to be used: a reasonable one seems to be the ordered typed fusion if code compaction is the first goal. McKinley and Kennedy also proposed a very simple heuristic for maximal reuse [10] which has been shown useful on real code examples.

Acknowledgments

I would like to give special thanks to Lucian Finta and Francis Sourd for helpful comments and references on the shortest common supersequence problem and on related scheduling problems.

References

[1] John R. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Journal of Parallel Computing*, 24(3), 1998. Special issue on Languages and Compilers for Parallel Computers.

[3] David Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[4] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–497, 1997.

[5] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *The 5th Workshop on Languages and Compiler for Parallelism*, number 757 in Lecture Notes in Computer Science, pages 281–295. Springer-Verlag, 1992.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability, a guide to the theory of NP-completeness*. W. H. Freeman and Company, 1979.

[7] Ken Kennedy and Kathryn S. McKinley. Typed Fusion with Applications to Parallel and Sequential Code Generation. Technical Report CRPC-TR94646, Center for Research on Parallel Computation, Rice University, 1994.

[8] C. B. Lofgren, L. F. McGinnis, and C. A. Tovey. Routing printed circuit cards through an assembly cell. *Operations Research*, 39(6):992–1004, November-December 1991.

[9] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[10] Kathryn S. McKinley and Ken Kennedy. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *The Sixth Annual Languages and Compiler for Parallelism Workshop*, number 768 in Lecture Notes in Computer Science, pages 301–320. Springer-Verlag, 1993.

[11] M. Middendorf. More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science*, 125:205–228, 1994.

[12] M. F. P O'Boyle, A. P. Nisbet, and R. W. Ford. A compiler algorithm to reduce invalidation latency in virtual shared memory systems. In *Proceedings of PACT'96*, Boston, MA, October 1996. IEEE Computer Society Press.

[13] C. Picouleau. Two new NP-complete scheduling problems with communication delays and unlimited number of processors. Technical Report 91-24, IBP, Université Pierre et Marie Curie, France, April 1991.

[14] K.-J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16:187–198, 1981.

[15] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.