Typed Fusion with Applications to Parallel and Sequential Code Generation*

Ken Kennedy
Department of Computer Science
Rice University, CITI
Houston, TX 77251-1892

Kathryn S. McKinley
Department of Computer Science
University of Massachusetts, LGRC
Amherst, MA 01003-4610

Abstract

Loop fusion is a program transformation that merges multiple loops into one and is an effective optimization both for increasing the granularity of parallel loops and for improving data locality. This paper introduces *typed fusion*, a formulation of loop fusion which captures the fusion and distribution problems encountered in sequential and parallel program optimization. Typed fusion is more general and applicable than previous work.

We present a fast algorithm for a typed fusion on a graph $G = (N, E)$, where nodes represent loops, edges represent dependence constraints between loops and each loop is assigned one of T distinct types. Only nodes of the same type may fuse. Only nodes of the same type may be fused. The asymptotic time bound for this algorithm is $O((N + E)T)$. The fastest previous algorithm considered only one or two types, but was still $O(NE)$ [KM93]. When $T > 2$ and there is no reason to prefer fusing one type over another, we prove the problem of finding a fusion with the fewest resultant loops to be NP-hard. Using typed fusion, we present fusion and distribution algorithms that improve data locality and a parallel code generation algorithm that incorporates compound transformations. We also give evidence of the effectiveness of this algorithm in practice.

1 Introduction

Loop fusion is useful because it can increase the granule size of parallel loops and expose opportunities to reuse variables in local storage. When it can be legally applied, loop fusion transforms multiple distinct compatible loop nests into a single nest. Loop nests can be incompatible for a variety of reasons. For example, the outer loop in one nest could be parallel while the other is sequential. We do not fuse these nests, because the resultant loop must be sequential.

One important source of incompatibility is *conformability*. Two loop headers are said to be *conformable* if they have the same number of iterations and are both either sequential or parallel loops. Note that they need not have the same index variable. Previous algorithms for performing loop fusion have only considered the case when all loop headers are conformable [Cal87, GOST92, KM93].

In *typed fusion*, we consider a wider class of problems where the loops being considered for fusion need not have conformable headers. The *type* of a loop or loop nest is determined by its header(s). Two loop nests with conformable headers have the same type. Typed fusion seeks to find the minimal number of loops resulting from a fusion in which nodes of a different type cannot be fused. Example 1 shows the application of typed fusion of sequential loops in order improve variable reuse on a uniprocessor.

*This research was supported in part by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center.

```
L1  DO J = 1, JMAXD
      DO I = 1, IMAXD
        F(I,J,1)=F(I,J,1)*B(1)
L2  DO K = 2, N-1
      DO J = 1, JMAXD
        DO I = 1, IMAXD
          F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1))*B(K)
L3  DO J = 1, JMAXD
      DO I=1, IMAXD
        TOT(I,J) = 0.0
L4  DO J = 1, JMAXD
      DO I = 1, IMAXD
        TOT(I,J)=TOT(I,J)+D(1)*F(I,J,1)
L5  DO K = 2, N-1
      DO J = 1, JMAXD
        DO I = 1, IMAXD
          TOT(I,J)=TOT(I,J)+D(K)*F(I,J,K)
```

(a) Original

```
G1  DO J = 1, JMAXD
      DO I = 1, IMAXD
        F(I,J,1)=F(I,J,1)*B(1)
        TOT(I,J) = 0.0
        TOT(I,J)=TOT(I,J)+D(1)*F(I,J,1)

G2  DO K = 2, N-1
      DO J = 1, JMAXD
        DO I = 1, IMAXD
          F(I,J,K)=(F(I,J,K)-A(K)*F(I,J,K-1))*B(K)
          TOT(I,J)=TOT(I,J)+D(K)*F(I,J,K)
```

(b) After Typed Fusion

Example 1: Subroutine *tridvpk* from ERLEBACHER.

Example 1(a) contains a fragment taken from the subroutine *tridvpk* in ERLEBACHER[1] where JMAXD, IMAXD, and N are compile time constants. If we only consider fusion problems where all the nests are conformable (as we did in our previous work [KM93]), we would only attempt to fuse L_3 and L_4.

Using typed fusion, we instead identify two types, t_0 and t_1 of conformable loop nests which may be fused, where L_1, L_3, and L_4 are of type t_0, and L_2 and L_5 are of type t_1. The typed fusion algorithm presented here fuses L_1, L_3 and L_4 into the fused nest G_1 and L_2 and L_5 into G_2 (see Example 1 (b)). Note that even though loops of type t_0 may not fuse with loops of type t_1, data dependences between loops of differing types prevent the problems from being considered in isolation. In our example, the data dependences require G_1 to precede G_2. In addition, data dependences represent opportunities to move references to the same memory location closer together in time, increasing the likelihood of reuse from cache or a register. For instance in G_1, the references to F(I,J,1) now occur on the same iteration rather than (N-1)$*$JMAXD$*$IMAXD + 2$*$JMAXD$*$IMAXD iterations apart and may therefore be placed in a register.

This paper addresses the problem of producing the fewest number of resultant loop nests for the typed fusion problem without violating dependence constraints. We begin with a short technical background and formulate the typed fusion problem. We then contrast this work to previous work. In Section 5, we further motivate typed fusion by presenting several of its applications in parallel and sequential code generation. We next show typed fusion to be NP-hard if the number of types is not bounded.

In Section 7.1, we present an algorithm which fuses loops of the same type in a graph with typed nodes, dependence edges and *fusion-preventing* dependence edges. This algorithm requires $O(N + E)$ time and space. We employ this algorithm to produce an $O((N + E)T)$ time algorithm for *ordered typed fusion*, in which the T types can be strictly prioritized. This algorithm produces a fusion that is *incrementally optimal* in the sense that the fusion produces the minimum number of loops of each type, assuming that the fusions of all the higher-priority types are given and cannot be changed. Finally, Section 8 gives preliminary evidence of the effectiveness of typed fusion.

2 Technical Background

2.1 Dependence

We assume the reader is familiar with *data dependence* [Ber66, KKP+81] and the terms *true*, *anti*, *output* and *input* dependence, as well as the distinction between *loop-independent* and *loop-carried* dependences [AK87]. *Parallel*

[1] An ADI solver written by Thomas M. Eidson at ICASE

loops have no loop-carried dependences and *sequential loops* have at least one.

2.2 Safe Loop Fusion

Loop fusion is a *loop-reordering transformation*; it changes the order in which loop iterations are executed. A reordering transformation is said to be *safe* if it preserves all true, anti and output dependences in the original program. Input dependences need not be preserved.

Consider safe loop fusion between two loops (or loop nests) with conformable headers. Between two candidates the following cases may occur: (1) no dependence, (2) a loop-independent dependence, and (3) a dependence carried by an outer loop which encloses the candidates. Clearly, fusion is always safe, but maybe not desirable for case (1). Fusion is also safe in case (3); any loop-carried dependence between two loops *must* be on an outer loop which encloses them and fusing them does not change the carrier, thus preserving the dependence.

In the case of a loop-independent dependence, fusion is safe if the sense of the dependence is preserved, *i.e.*, if the dependence direction is not reversed. A simple test for this case performs dependence testing on the loop bodies as if they were in a single loop. After fusion, a loop-independent dependence between the original nests can (a) remain loop-independent, (b) become forward loop-carried or (c) become backward loop-carried. Since the direction of the dependence is preserved in the first two cases, fusion is legal. Fusion is illegal when a loop-independent dependence becomes a backward carried dependence after fusion. These dependences are called *fusion-preventing* dependences [AS79, War84].

Since a loop is parallel if it contains no loop-carried dependences and is sequential otherwise, fusion in case (b) is safe but prevents parallelization of the resultant loop. If either one or both of the loops were parallel, fusion would reduce loop parallelism. Consequently, when fusion is concerned with maximizing loop parallelism, these dependences are fusion-preventing. The algorithms below only perform safe fusions.

2.3 Loop Distribution

If loop distribution is applied to create the finest possible loop nests, then to exploit data locality or increase the granularity of parallel loops, loop fusion must be applied [KM93]. Two applications which use typed fusion to perform loop distribution is described in Section 5. However, all the fusion algorithms are applicable to loop distribution.

3 Typed Fusion

We represent the typed fusion problem with a graph in which each candidate loop nest is represented by a node. We say a pair of nests with depths d_1 and d_2 are conformable at level k (where $k \leq d_1$ and $k \leq d_2$) if loops 1 to k are perfectly nested and each pair of loop headers is conformable. A pair of conformable loop headers have the same number of iterations at each loop level and when generating parallel loops, they are either both parallel or sequential. If two nests have conformable headers at level k, they have the same type, otherwise their types are distinct and they cannot be fused. Each program fragment which falls between a pair of candidate nests is also assigned a node and a unique type, such that it may not fuse with any other node. Dependence and fusion-preventing edges from statements represented by one node to statements represented by a different node correspond to edges with the same orientation in the fusion graph. These edges further restrict the safe fusions which may be performed.

Given the definition of dependence on the original program ordering, the fusion graph is a directed acyclic graph or DAG. Typed fusion partitions nodes into fusion sets using the following rules and goal.

Rules:

1. *Separation constraint:* two nodes of different type cannot fuse.
2. *Fusion-preventing constraint:* two nodes connected by a fusion-preventing edge cannot fuse.

3. *Ordering constraint:* the relative ordering of two nodes connected by a dependence edge cannot change.

Goal: without violating the constraints, fuse nodes of the same type so that resultant program has a minimal number of nodes.

4 Related Work

Typed fusion considers more general and complex problems and subsumes the problems that have previously been addressed in the fusion literature [Cal87, GOST92, KM93]. In addition to being more powerful, the typed fusion algorithm is $O((N + E)T)$ and is quicker than previous solutions when applied to the same problem domain. In Callahan's dissertation, a greedy loop fusion algorithm introduces both task and loop parallelism, but does not address improving data locality or granularity of loop parallelism [Cal87]. The greedy algorithm is $O(N^2 + E)$ space and time and is applicable only to a collection of conformable nests of a single type.

Gao *et al*. consider a weighted loop fusion problem for improving reuse on uniprocessors with and without pipelining [GOST92]. Their work is based on the maximum-flow/minimum-cut algorithm, but only considers sets of loops with conformable headers, *i.e.,* a single type $T = 1$. Their algorithm is limited because it does not reorder nests and it is not clear if its additional complexity results in better solutions. The algorithm is $O(knm\ log(n^2/m))$ time, where k is the number of fusion preventing edges, n is the number of nodes, and m is the number of edges.

For increasing the granularity of parallel loops, our previous work focused on a special case of typed fusion [KM93]. In this problem, all the nests have the same number of iterations and nodes are either marked parallel or sequential, *i.e.,* $T = 2$. The solution presented is $O(NE)$, but is only minimal in the number of parallel loops. We now have an algorithm for this problem which is minimal in the number of parallel loops and the total number of loops. In our previous work, we also considered the same weighted fusion problem as Gao *et al*. and proved it NP-hard [KM93]. We prove below that even the unweighted problem is NP-hard for $T > 2$. Typed fusion subsumes our previous work and fully characterizes general fusion problems.

5 Applications of Typed Fusion

This section describes three important applications of typed fusion.

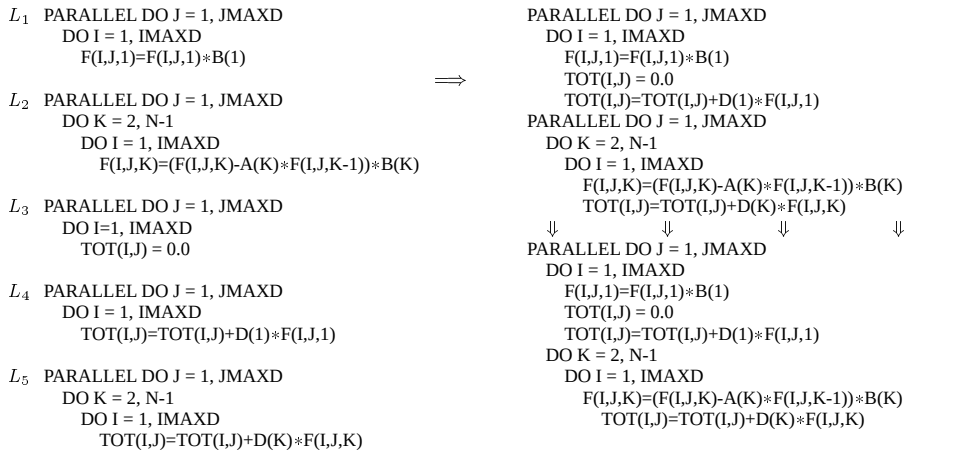
5.1 Sequential-Parallel Fusion or Loop Distribution for Parallelism

Suppose you create a collection of loops by distributing a single original loop around the statements in its body and then parallelize the loops wherever possible. The result is a collection of loops with conformable headers that come in two categories: parallel and sequential. A typed fusion algorithm with the two types, *parallel* and *sequential*, may be used to fuse these loops back together for maximum loop granularity.

5.2 Loop Fusion for Reuse

Given a graph which represents a collection of loop nests and code that is not in any loop, we would like to fuse conformable nests when they contain statements that access the same memory location, *i.e.,* introduce reuse. For obvious reasons, fusing to introduce reuse needs to consider input dependences in addition to the true, anti and output dependences required for correctness. We thus add any undirected input dependence edges to the typed fusion graph.[2] In fact, only true and input dependences indicate opportunities for reuse of a value in a register since a write kills the

[2] Input dependences are undirected because the references may occur in either order.



(a) Parallelized Nests

(b) Two iterations of Typed Fusion by *ParallelCodeGen*

Example 2: Subroutine *tridvpk* from ERLEBACHER.

live range of a value. On many modern uniprocessors, even the time to write a cache line does not differ appreciably whether the corresponding line is in cache or not. Accordingly, reuse may only be improved by fusion when loop nests are connected by a path of true and/or input dependences.

The typed fusion graph $G = (N, E)$ for reuse is thus constructed by adding input edges and typing the nests as follows. Two nodes have the same type if their nests are conformable and there exits a path in G between them consisting only of true and input edges. Returning again to Example 1, L_1 and L_4 are compatible and connected by input dependence, and L_3 and L_4 are connected by a true dependence. By transitivity, L_1, L_3, and L_4 are assigned the same type. This model may be further refined to keep register pressure down by assigning the same type only when the references may share a register after fusion.

We can also weight the edges to further differentiate between fusion choices. This problem is NP-hard even for one type [KM93]. Extending this algorithm to the weighted problem for reuse is left for future work.

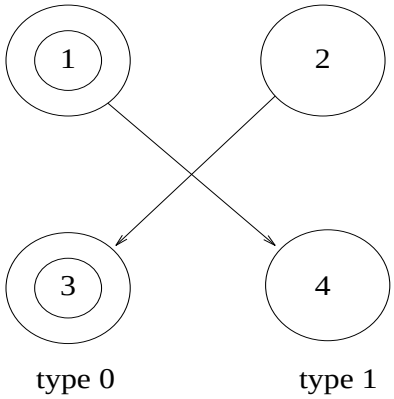
5.3 Parallel Code Generation

Now we examine the problem of performing loop fusion as part of a general automatic parallel code generator. Given $L = \{l_1, l_2, ..., l_n\}$ a collection of nests and code that is not in any nest, the body of each nest l_i may also be a collection of nests and statements. This view inspires a hierarchical code generation scheme we call *ParallelCodeGen*. The basic strategy is to parallelize each nest l_i independently and then fuse the results together. If l_i cannot be parallelized, it is decomposed into the finest possible subnest using maximal distribution on which *ParallelCodeGen* is applied recursively. In a little more detail, the following steps make up *ParallelCodeGen*.

1. For $i = 1$ to n attempt to parallelize l_i. We use techniques described elsewhere [KM92] which perform loop interchange and other transformations to introduce outer loop parallelism and effectively exploit data locality. A parallelization strategy appropriate to the target architecture should be selected. If l_i cannot be parallelized, maximally distribute l_i into subnests $SN = \{s_1, s_2, ..., s_m\}$ and invoke *ParallelCodeGen* recursively on SN.

2. After all l_i have been considered, type the nests as deeply as possible, *i.e.*, if two or more nests of depth d are conformable to depth d, assign them a unique type. A worst case $O(N^2)$ time algorithm to create these types places l_1 into an equivalence class and considers adding l_2. If l_2 and l_1 are conformable at any level k, add l_2 to the class and mark it with k, otherwise create another class. Now consider adding l_j to a class with 2 or more

Figure 1: A Typed Fusion Conflict



members. If l_j is conformable with a member at a deeper level than k, then the members conformable to l_j are split into a new class. Each equivalence class is then assigned a unique type.

3. Apply typed fusion. If fusion is applicable, repeat step 2 but find equivalence classes only for depth $k-1$ or shallower, where k is the depth of the deepest nests just fused and $k > 1$.[3]

As an example, we return to ERLEBACHER. Assume a parallelization strategy that simply finds a parallel loop and moves it to its outermost legal position. In Example 1, all the J loops would be made parallel and in L_2 and L_4 the J loop would be shifted into the outermost position, completing step 1 in *ParallelCodeGen* as illustrated in Example 2(a). The first iteration of steps 2 and 3 then assign the classes {L_1,L_3,L_4}and {L_2,L_5}distinct types and fuses each at depth 2 and 3 respectively, resulting in the two nests in the top right corner of Example 2(b). The second iteration of steps 2, builds an equivalence class for depth 2 or shallower. The two nests are conformable at level 1 and are safely fused in step 3, resulting in the single parallel nest in Example 2(b).

This hierarchy prefers fusions of deeper loop nests which creates the most opportunities for reuse. An alternative strategy which prefers larger granularity of parallelism fuses beginning with single outermost loops, recursing on inner loops inside any fused region. In Example 2, since all the fusions of conformable headers are legal, fusing innermost first or outermost first makes no difference in the result.

6 Unordered Typed Fusion

The *unordered typed fusion* problem assumes that there is no reason to prefer fusion of one type of node over another. Consider the fusion graph depicted in Figure 1. In this graph, if we fuse the two nodes of type 0, we cannot fuse the nodes of type 1 because that would introduce a cycle in the fused graph. Such a cycle is illegal because it is impossible to chose an order for the resulting nodes. How difficult is it to find the optimal solution for the unordered typed fusion problem with the constraints enumerated in Section 3, where optimal means fusing the nodes into the fewest number of resultant loops?

Theorem 1: *Unordered Typed Fusion is NP-hard.*

Proof. To establish the result, we present a polynomial-time algorithm for reducing Vertex Cover to Unordered Typed Fusion. Vertex Cover is the problem of choosing a minimal set *S* of vertices in an undirected graph $G = (V, E)$ such that every edge in V is incident on at least one vertex in *S*. Construct a fusion graph $G_F = (V_F, E_F)$ as follows.

[3] Two nests conformable at level k are also clearly conformable at level $k-1$.

1. Represent each vertex $v \in V$ from Vertex Cover by two nodes h_v and t_v with the same unique type in V_F.
2. For each edge $(v, w) \in E$ add two edges (h_v, t_w) and (h_w, t_v) to E_F.

This construction clearly takes time proportional to the size of the original Vertex Cover graph.

Claim: There is a solution S of size k for the Vertex Cover problem $G = (V, E)$ if and only if there is a solution of size $V + k$ to the Unordered Typed Fusion problem $G_F = (V_F, E_F)$.

If. Assume there is a solution of size k to the Vertex Cover problem, *i.e.,* there exists a set S of size k which is an edge cover for $G = (V, E)$. For each node $v \in S$, place a fusion-preventing dependence (h_v, t_v) in G_F. If w is not in S, no fusion-preventing edge is needed between h_w and t_w in G_F. The only way an edge might be required is if there is another pair h_x and t_x such that (h_w, t_x) and (h_x, t_w) are both in E_F and there is no fusion-preventing edge (h_x,t_x). But the corresponding edge (w, x) is in E and neither w nor x is in S, implying S is not an edge cover, a contradiction. Every h_w and t_w, where w is not in S, can therefore be fused. After fusion, there are k types with two vertices, $V - k$ types with only one vertex, and the size of the solution graph is $V + k$.

Only If. Assume that there exists a set F of types in which the h_v and t_v node are fused and the size of this set is $V - k$. Since the remaining k types have two vertices each, the total number of vertices in G_F is $V + k$. Let S be the set of vertices in V corresponding to the types that are not in F. S must be an edge cover. If it were not, there would exist at least one edge (v, w) where neither endpoint is in S, but then there would be edges (h_v, t_w) and (h_w, t_v) and the two pairs h_v and t_v, and h_w and t_w would be fused F. Those fusions cannot be in F because it would cause a cycle in the graph after fusion. □

Note that the number of *types* in this reduction corresponds to the number of vertices in the Vertex Cover problem, which suggests that there might exist polynomial solutions for a fixed number of types. Callahan proved that if a greedy algorithm is employed on a DAG with a single type, the resulting fusion has the smallest possible number of partitions [Cal87]. We also believe there exists a polynomial-time algorithm to find an optimal solution to the Unordered Typed fusion problem for two types. This proof is left for future work.

7 Ordered Typed Fusion

In some applications of typed fusion, there is an implied preference that can help resolve conflicts in the fusion graph. For example in the simple case of fusing parallel and sequential loops where reuse is not a consideration (the application from Section 5.1), the parallel loop is always preferred for fusion, because larger granularity is critical for parallelism, while fusing two sequential loops can only save a little loop overhead.

The *ordered typed fusion* is a typed fusion problem in which there is an absolute order of the types $\{t_1, t_2, \ldots, t_k\}$ such that if there is a conflict between types t_i and t_j where $i < j$ then it is always preferable to fuse the nodes of type t_i, no matter how many fusions are prevented in types $t_{i+1}, t_{i+2}, \ldots, t_k$ as a result. An ordered typed fusion is *incrementally optimal* if for each type t, given a fusion of types with higher priority, the fusion for t has a minimal number of resultant loops. In the next section, we introduce an algorithm that produces an incrementally optimal solution for the ordered typed fusion problem.

7.1 A Typed Fusion Algorithm

We now explain the algorithm *TypedFusion* from Figure 2 for typed fusion of a single selected type t_0 in a graph with multiple types. In essence, this algorithm carries out greedy fusion for the single selected type. We define a *bad path* for type t to be a path that begins with a node of type t and either contains a fusion-preventing edge between two nodes of type t, or a node of type different from t. The algorithm treats a bad path as a fusion-preventing edge. Only nodes of the same type are considered for fusion.

TypedFusion achieves an optimal $O(N + E)$ time bound by choosing the correct node with which to fuse in constant time as each new node n of the selected type is visited. Intuitively, n can fuse with all its ancestors from

which there is not a bad path, but it cannot by pass its predecessors to fuse with an ancestor. It can also fuse with nodes which are neither ancestors nor descendants. *MaxBadPrev*(n) determines the first node m of type t_0 from which there is no bad path to n. It contains the maximum number of a node of type t_0 from which there exists a bad path to the node n being visited. We show that n cannot fuse with that node or any node with lower number, where nodes are numbered breadth-first in their fusion group.

The algorithm also computes $maxPred(n)$, the highest number of a direct predecessor of n with which n can fuse. We show that n cannot fuse with any node numbered lower than that predecessor. The algorithm therefore computes the fusion node by taking the maximum of the next higher node than *maxBadPrev*(n) and of *maxPred*(n). If the result is 0, then n receives the next higher new number. Consequently, n fuses with a direct predecessor, or a node from which there is no path to n. The algorithm produces a greedy fusion for the selected type, because it finds the lowest-numbered node with which n can fuse.

We note that if there is only one type, this algorithm is an $O(N + E)$ algorithm for untyped fusion, the fastest such algorithm of which we are aware.

Correctness. To establish correctness, we must show that the constraints in the problem definition are observed. First, the separation constraint is clearly satisfied, because the algorithm fuses only nodes of the selected type. To show that it obeys the fusion-preventing constraint, we must establish that it never fuses two nodes joined by a fusion-preventing edge. To show that it satisfies the ordering constraint, we must show that it never fuses two nodes joined by a path that passes through a node of different type. Since all original edges are left in the graph, this ensures that fusion does not create a cycle in the output DAG. Both of these hypotheses are established by the following Lemma, which relies on the greedy structure of the algorithm.

Lemma 1. *For each n taken from the worklist W in TypedFusion, n cannot fuse with a previously visited node x if (a) there exists a previously visited node $y \neq x$ such that $type(y) = type(n) = t_0$, $num(y) > num(x)$ and y cannot legally be fused with n or (b) there exists a node m such that $(m, n) \in E$, m can be fused with n, and $num(m) > num(x)$.*

Proof. Suppose there is an element for which neither (a) nor (b) hold, then there must exist some first element n_1 for which they do not hold. Since the algorithm always picks the first node of type t_0 in visit order, assume that previous steps of the algorithm always correctly fused with the earliest possible node of the same type. Recall that a *bad path* for type t_0 is one that begins at a node of type t_0 and either contains a fusion-preventing edge between two nodes of that type or passes through a node of a different type. We can fuse n_1 with any node of type t_0 as long as there is no bad path from that node to n_1.

1. To establish condition (a), assume that n_1 can fuse with x even though there exists a y of the same type and larger number that cannot fuse with n_1. Since previous steps of the algorithm correctly fuse with the lowest-numbered node of compatible type, we must assume that y did not fuse with x because it would have been incorrect to do so. Consequently, there must be a bad path for $type(n_1)$ from x to y. But since y cannot fuse with n_1, there must also be a bad path from y to n_1. But that means that there must be a bad path from x to n_1, so the nodes cannot be fused.

2. Assume there exists a direct predecessor m of n_1 such that $type(m) = type(n_1) = t_0$ and $num(m) > num(x)$. Why did this node not fuse with x? Since previous steps of the algorithm correctly fuse with the earliest possible node, there must be a bad path for $type(n_1)$ from x to m. Hence, there is a bad path from x to n_1 and these nodes cannot be fused.

TypedFusion computes for each node m the quantity *maxBadPrev*(m), which is the maximum number for nodes of type t_0 visited by the time m is reached from which there is a bad path for type t_0 to m. A node n may

Figure 2: Algorithm for Fusion of a Specified Type in a Graph

TypedFusion (G, T, t_0)

INPUT: $G = (N, E)$ the original typed graph
T is a set of types
t_0 is a specific type for which we will find a minimal fusion
$type(n)$ is a builtin function that returns the type of a node

OUTPUT: L a linear list of fused loops consistent with the constraints

INTERMEDIATE: $num(n)$ is the number of the first visit to node n of type t_0.
$lastnum$ is the most recently assigned number.
$maxBadPrev(n)$ is $max\{num(x)|type(x) = t_0$ and $\exists$ a bad path for t_0 from x to $n\}$.
$MaxPred(n)$ is $max\{num(x)|(x, n) \in E$ & x has been visited$\}$.
$node(i)$ is an array that maps numbers to nodes such that $node(num(x) = x)$.
$visited$ is the number of the first node of type t_0 in the graph.
$next(i)$ maps the ith node to the number of the next node of the same type.
W is a working set of nodes ready to be visited

ALGORITHM:

$lastnum$:= 0; $count(*)$:= 0; $visited$:= 0; $node(*)$:= 0; **/* Initialization */**
for each edge $e = (m, n) \in E$ **do** $count(n) := count(n) + 1$;
for each node $n \in N$ **do** {
  $maxPred(n)$:= 0; $maxBadPrev(n) = 0$; $num(n)$:= 0; $next(n)$:= 0;
  if $count(n)$ = 0 **then** $W := W \cup \{n\}$ **end if**; }
while $W \neq \emptyset$ **do begin** **/* Iterate over working set, visiting nodes, and fusing nodes of type** t_0 ***/**
  let n be an arbitrary element in W; $W := W - \{n\}$; $t := type(n)$;
l1: **if** $t = t_0$ **then** /* A node of the type being worked on */
    /* Compute node to fuse with. If none, assign a number and add to $visited$**.*/**
    if $maxBadPrev(n) = 0$ **then** $afterNoFuse$:=$visited$
    else $afterNoFuse := next(maxBadPrev(n))$ **end if**;
    $p := max(maxPred, afterNoFuse)$;
    if $p \neq 0$ **then**
      $x := node(p)$;
      $num(n) := num(x)$; $maxBadPrev(n) := max(maxBadPrev(n), maxBadPrev(x))$;
      fuse x and n and call the result n, making all edges out of either be out of n;
    else /* a new node */
      $lastnum := lastnum + 1$; $num(n) := lastnum$; $node(num(n)) := n$;
      /* append node n **to the end of** $visited$ ***/**
      if $lastnum = 1$ **then** $visited := lastnum$
      else $next(lastnum - 1) := num(n)$;**end if**;
  end if; **end if**;
  /* Update $maxBadPrev$ **and** $maxPred$ **for successors and add to** W **as appropriate */**
l2: **for each** node m such that $(n, m) \in E$ **do** {
    $count(m) := count(m) - 1$; **if** $count(m)$ = 0 **then** $W := W \cup \{m\}$; **end if**;
    if $t \neq t_0$ **then**
      $maxBadPrev(m) := max(maxBadPrev(m), maxBadPrev(n))$
    else /* $t = t_0$ ***/**
      if $type(m) \neq t_0$ **or** (n, m) is fusion-preventing **then**
        $maxBadPrev(m) := max(maxBadPrev(m), num(n))$
      else /* equal types and not fusion preventing */
        $maxPred(m) := max(maxPred(m), num(n))$;
        $maxBadPrev(m) := max(maxBadPrev(m), maxBadPrev(n))$
    end if; **end if**; }
end for; end while;

Figure 3: Algorithm for Ordered Typed Fusion.

OrderedTypedFusion (G, T)
INPUT: $G = (N, E)$ the original typed graph
T is a set of types, ordered by priority
ALGORITHM:
for all $t \in T$ in order of decreasing type priority **do TypedFusion**(G, T, t);
topologically sort G to produce final output
end OrderedTypedFusion

not fuse with a node numbered *maxBadPrev*(n) or lower, by the lemma above. It could, however fuse with *afterNoFuse*=*next*(*maxBadPrev*(n)).

The algorithm also computes for each node n of type t_0 the quantity *maxPred*(n), the maximum number of a predecessor of n that *can* be fused with n. By Lemma 1, n cannot fuse with any node having a number less than *maxPred*(n). Thus, if *max(afterNoFuse,maxPred*(n)) exists, it is exactly the first node with which n may legally fuse. The algorithm is greedy, because it always chooses to fuse with this node, if it exists.

Since all edges in the original graph are left in the final graph (adjusted to reflect fusion) and we cannot fuse nodes joined by a bad path, the output graph after fusion is acyclic. □

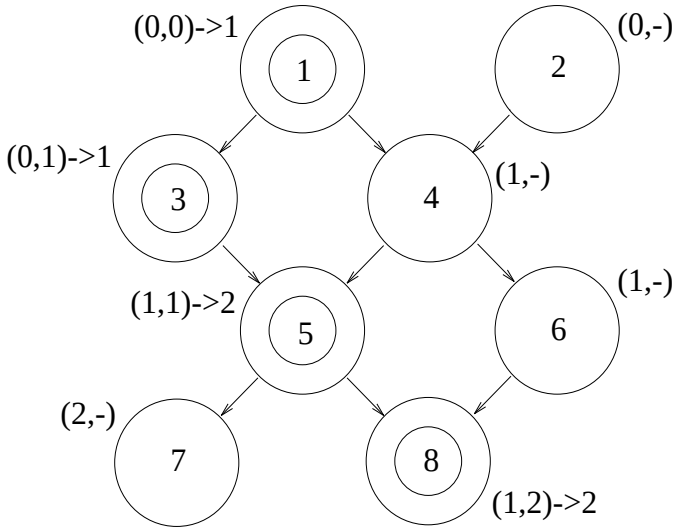
Optimality. A greedy algorithm for fusing nodes of the same type repeatedly selects loops from a set for which all predecessors have been assigned to a partition and adds them to the current partition until no more can be added. At that point, it starts a new current partition. In his dissertation, Callahan proved that if a greedy algorithm is employed on a DAG where all the nodes are of the same type, the resulting fusion is optimal, *i.e.*, it has the smallest possible number of partitions [Cal87]. The proof here is similar. *TypedFusion* carries out the greedy algorithm for a selected type because, for each node n, it finds the lowest-numbered node in *visited*($type(n)$) that can be fused with n. □

Complexity. There are two major phases in this algorithm: initialization and worklist iteration. Clearly, initialization takes no more than $O(N + E)$ time. Consider the worklist iteration. Each node is put on and extracted from the worklist W exactly once, so the total time taken by worklist extraction is $O(N)$. For each node extracted, two major phases are executed. In the code beginning at label *l1*, which selects the node for fusion, no operation takes more than constant time. The loop beginning at label *l2* examines each successor of the node being visited exactly once, so the total number of iterations of this loop is $O(N + E)$. Within a single iteration, all of the operations can be implemented in constant time. Hence, the total time for the loop is $O(N + E)$.

Procedure *OrderedTypedFusion* can be used to solve the ordered typed fusion problem by calling it for each type in order of reducing priority, as shown in Figure 3. Proof that this algorithm is correct and produces an incrementally optimal schedule is straightforward. Since algorithm *TypedFusion* is called once for each type, the entire process takes $O((N + E)T)$, where T is the total number of types in the graph.

Discussion. To adapt this algorithm for untyped fusion, some order must be selected in which to consider the types. This order should be dependent on the application. When fusing for reuse for example, the types should probably be ranked based on their amount of potential reuse.

Figure 4: Parallel-sequential Fusion Example with Label Annotations by *TypedFusion*.



Example. As an example, we demonstrate the behavior of *OrderedTypedFusion* on the problem of fusing parallel and sequential loops from Section 5.1. The input graph is shown in Figure 4, in which parallel loops are indicated by a double circle and sequential loops by a single circle. The label within each node indicates the order in which that node is visited.

The labels adjacent to each node illustrate the behavior of the first pass of *TypedFusion* which fuses the parallel nodes. The parallel nodes are annotated with the following labels:

$$\textit{(maxBadPrev(parallel), maxPred)} \rightarrow \textit{final node number},$$

and sequential nodes are annotated with a pair:

$$\textit{(maxBadPrev(parallel), -)}.$$

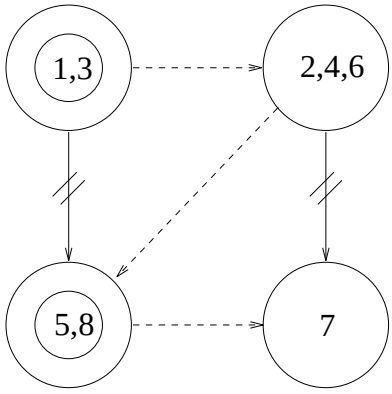
Using these labels in Figure 4, we can see that when node 5 is visited it cannot fuse with node 1 because it inherits a *maxBadPrev(parallel)* of 1 from the node visited on step 4. The rest of the example is straightforward from examination of the *TypedFusion* algorithm. The final graph after fusion is shown in Figure 5.

8 Experimental Results and Discussion

In our previous work, improvements in execution times due to untyped fusion ranged from 4 up to 32 percent [KM93]. Every time fusion was applied it was profitable. For example, untyped fusion of ERLEBACHER on uniprocessors improved performance between 4 and 17 percent. However, typed fusion on the nests in Example 1 fuses nests of depth 2 and 3 that provide significant locality and could not be fused in untyped fusion. We expect further improvements for both parallel and sequential execution.

The fusion problems we have encountered are fairly simple (7 or fewer initial loop nests) and are handled optimally using these algorithms. This preliminary evidence indicates that a more complex algorithm is probably unnecessary in practice. However, more complex problems may be presented by Fortran 90 programs. Fusion is especially important for Fortran 90 because of the array language constructs. To compile Fortran 90 programs, the array notation must be expanded into loop nests which contain a single statement, providing significantly more opportunities for fusion.

Figure 5: Final Graph after Fusion, but Before Topological Ordering



9 Summary

This paper discusses the problem of typed fusion, which seeks to fuse nodes of the same type in a DAG that contains nodes of several different types. In such a graph two nodes of the same type may be fused if there is no fusion-preventing edge between them and they are not joined by a path that passes through a node of a different type or a node of the same type that cannot be joined to either of the endpoints. Finding an optimal solution for the problem of unordered typed fusion is shown to be NP-hard.

The paper presents an incrementally optimal algorithm for ordered typed fusion of loops that maximizes the granularity of loop parallelism, therefore minimizing synchronization. The algorithm runs in $O((N + E)T)$ time, where N is the number of nodes in the input graph, E is the number of edges and T is the number of types. This algorithm is applied to several problems in parallel and sequential code generation. One of these applications is a general framework for parallel code generation using loop fusion and loop distribution.

References

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AS79] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, October 1966.

[Cal87] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[GOST92] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[KKP+81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[KM92] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[KM93] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[War84] J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, UT, January 1984.