Nested Circular Intervals: A Model for Barrier Placement in Single-Program, Multiple-Data Codes with Nested Loops.

Alain Darte, Robert Schreiber

Nested Circular Intervals: A Model for Barrier Placement in Single-Program, Multiple-Data Codes with Nested Loops

Alain Darte and Robert Schreiber

December 2004

Research Report N^o 2004-57

Nested Circular Intervals: A Model for Barrier Placement in Single-Program, Multiple-Data Codes with Nested Loops

Alain Darte and Robert Schreiber

December 2004

Abstract

We want to perform compile-time analysis of an SPMD program and place barriers in it to synchronize it correctly, minimizing the runtime cost of the synchronization. This is the barrier minimization problem. No full solution to the problem has been given previously.

Here we model the problem with a new combinatorial structure, a nested family of sets of circular intervals. We show that barrier minimization is equivalent to finding a hierarchy of minimum cardinality point sets that cut all intervals. For a single loop, modeled as a simple family of circular intervals, a linear-time algorithm is known. We extend this result, finding a linear-time solution for nested circular intervals families. This result solves the barrier minimization problem for general nested loops.

Keywords: Barrier synchronization, circular arc graph, nested circular interval graph, SPMD code, nested loops

Résumé

Le but de ce rapport est de montrer comment, après une analyse statique de code, on peut synchroniser, à l'aide de barrières, un programme de type SPMD tout en minimisant le temps de synchronisation à l'exécution. C'est le problème de minimisation des barrières. Aucune solution complète n'a été donnée à ce jour.

Nous modélisons le problème par une nouvelle structure qui généralise la notion de graphe d'arcs circulaires, une famille d'intervalles circulaires imbriqués. Nous montrons que le problème de minimisation de barrières revient à trouver une hiérarchie d'ensembles, de tailles minimales, de points du code (où placer les barrières) qui coupent Ż tous les intervalles. Pour une boucle simple, modélisée par un graphe d'arcs circulaires traditionnel, un algorithme linéaire est connu. Nous l'étendons en un algorithme linéaire pour une famille d'intervalles circulaires imbriqués. Ce résultat résout le problème de minimisation des barrières pour des boucles imbriquées.

Mots-clés: Barrière de synchronisation, graphe d'arcs circulaires, graphe d'intervalles circulaires imbriqués, code SPMD, boucles imbriquées

Contents

Nested Circular Intervals: A Model for Barrier Placement in Single-Program, Multiple-Data Codes with Nested Loops

Alain Darte
CNRS, LIP, ENS Lyon
46, Allée d'Italie,
69364 Lyon Cedex 07, France
Alain.Darte@ens-lyon.fr

Robert Schreiber
Hewlet Packard Laboratories,
1501 Page Mill Road,
Palo Alto USA
Rob.Schreiber@hp.com

16th December 2004

Abstract

We want to perform compile-time analysis of an SPMD program and place barriers in it to synchronize it correctly, minimizing the runtime cost of the synchronization. This is the barrier minimization problem. No full solution to the problem has been given previously.

Here we model the problem with a new combinatorial structure, a nested family of sets of circular intervals. We show that barrier minimization is equivalent to finding a hierarchy of minimum cardinality point sets that cut all intervals. For a single loop, modeled as a simple family of circular intervals, a linear-time algorithm is known. We extend this result, finding a linear-time solution for nested circular intervals families. This result solves the barrier minimization problem for general nested loops.

1 The problem of static optimization of barrier synchronization

A multithreaded program can exhibit interthread dependences. Synchronization statements must be used to ensure correct temporal ordering of accesses to shared data from different threads. Explicit synchronization is a feature of thread programming (Java, POSIX), parallel shared memory models (OpenMP), and global address space languages (UPC [15], Co-Array Fortran [7]). Programmers write explicit synchronization statements. Compilers, translators, and preprocessors generate them. In highly parallel machines, synchronization operations are time consuming [1]. It is therefore important that we understand the problem of minimizing the cost of such synchronization. This paper takes a definite step in that direction, beyond what is present in the literature. In particular, we give for the first time a fast compiler algorithm for the optimal *barrier* placement problem for a program with arbitrary loop structure.

The *barrier* is the most common synchronization primitive. When any thread reaches a barrier, it waits there until all threads arrive, then all proceed. The barrier orders memory accesses: memory operations that precede the barrier must complete and be visible to all threads before those that follow. Even with the best of implementations, barrier synchronization is costly [11]. All threads wait for the slowest. Even if all arrive together, latency grows as $\log n$ with n threads. Finally, the semantics of a barrier generally must include a *memory fence*, which causes all memory operations that precede the barrier to be fully completed and globally visible before the start of any memory operation that follows the barrier.

Programmers and compilers add barriers to guarantee correctness. Experimental evidence [13] shows that programmers oversynchronize their codes. This is perhaps because it is hard to write correct parallel code, free of data races. We would therefore like to be able to minimize the cost of barriers through compiler optimization. A practical, automatic compiler barrier minimization algorithm would make it appreciably easier to write fast and correct parallel programs by hand and to implement other compiler code transformations, by allowing the programmer or other compiler phases to concentrate on correctness and rely on a later barrier minimization phase for reducing synchronization cost.

We call an algorithm *correct* if it places barriers so as to enforce all interthread dependences, and *optimal* if it is correct and among all correct barrier placements it places the fewest possible in the innermost loops, among such it places the fewest at the next higher level, etc. In their book on the implementation of data parallel languages, Quinn and Hatcher mention the barrier minimization problem [9]. They discuss algorithms for inner loops but not more complicated program regions. O'Boyle and Stöhr [13] make several interesting contributions. Extending the work of Quinn and Hatcher, they give an optimal algorithm for an inner loop with worst-case complexity $O(n^2)$, where n is the number of dependences, and an algorithm that finds an optimal solution for any *semiperfect* loop nest, i.e., a set of nested loops with no more than one loop nested inside any other. Its complexity is quadratic in the number of statements and exponential in the depth of the nest. Finally, they give a recursive, greedy algorithm for an arbitrarily nested loop, and finally for a whole program. This algorithm is correct, and it will place the fewest possible barriers into innermost loops. But it doesn't always minimize the number of barriers in any loop other than the innermost loops.

We describe (for the first time) and prove correct and optimal an algorithm for barrier minimization in a loop nest of arbitrary structure. The algorithm is fast enough to be used in any practical compiler: it runs in time linear in the size of the program and the number of dependence relations it exhibits.

Remarks Note that, in this report, we don't consider two important optimizations related to synchronization, a) statement reordering and b) the use of lighter weight synchronizations.

a) We have chosen to look at a model of the problem in which barriers must be placed without other changes to the program. In particular, we disallow reordering of statements and changes to the loop nesting structure, such as loop fusion and distribution might provide. We do not advocate this as a global program optimization strategy. Indeed, others have shown that such transformations may be beneficial. In the end, however, after such transformations have been applied by the optimizer, the problem that we address here remains: minimize the barriers without other code modifications. Barrier minimization with statement reordering is a scheduling problem. Barriers divide time into interbarrier epochs, and the problem is to schedule work into epochs such that the total length of the schedule is minimized. Callahan [5] and Allen, Callahan, and Kennedy [3] made basic contributions to the theory of program transformation to reduce schedule length. Note that statement reordering, even when legal, may not be advisable, principally because it can worsen memory performance, which is often a critical performance limiter. Thus, the problem at hand, nested loops with no reordering, is of considerable interest.

b) Some dependence relations do not require barriers: they are enforceable by lighter weight synchronization, such as event variable synchronization or point-to-point communication (see for example [14]). These dependences can be so enforced, and the code modified accordingly, before we consider the barrier minimization problem. It may be necessary, after this is done, to annotate the code so as to avoid the detection and enforcement (with a barrier) of a dependence that has been synchronized by event variables. We ignore this possibility for the remainder of the paper, and assume that barrier is the only primitive used for synchronization.

2 The program model and a statement of the problem

We assume a program with multiple threads that share variables. Each thread executes a separate copy of an identical program (single-program, multiple data, or SPMD). Threads know their own thread identifier (*mythread*) and the number of threads (*threads*). By branching on *mythread*, arbitrarily complicated MIMD behavior is possible. The threads call a barrier routine to synchronize. Barriers divide time into epochs. The effects of memory writes in one epoch are visible to all references, by all threads, in the following epochs.

Clearly, if any thread hits a barrier then all threads must execute a barrier or there will be a deadlock, with some threads waiting forever. So in a correct program, all threads make the same number of barrier calls. We make a stronger assumption: following Aiken and Gay, we assume that the program is **structurally correct** [2], which means that all threads synchronize by calling the same barrier statement, at the same iteration of any containing loops. The simple way to understand this is as a prohibition on making a barrier control-dependent on any *mythread*-dependent condition. Structural correctness may be a language requirement as in the Titanium language [17]. Titanium uses the keyword *single* to allow a programmer to assert that a private variable takes only thread-independent values. We can also optimize programs in looser SPMD languages such as UPC [6] and Co-Array Fortran [12] if we discover at compile time that they have no structural correctness violations.

We don't view structural correctness as a significant restriction on the programmer's ability to express important and interesting parallel algorithms. Here's why. Aiken and Gay presented empirical evidence that actual shared-memory parallel applications rarely violate structural correctness, even in dialects that allow it. They implemented static single-valuedness analysis as well as the *single* keyword in an extension of the SPMD language Split-C [4]. In this dialect, they were able to implement and statically verify the structural correctness of a variety of typical parallel scientific benchmark codes (cholesky, fft, water, barnes, etc.) by making a small number of uses of *single* [2]. Also, structural correctness is a natural property of any SPMD implementation of a program written originally in a traditional fork/join model of parallelism such as OpenMP. Threads will synchronize with (one and the same) barrier at the end of each parallel construct.

We can analyze and optimize any program region consisting of a sequence of loops and statements, which we call a **properly nested region**. We can change any properly nested region into a single loop nest, by adding an artificial outer loop (with trip count one) around the region. We can, therefore, take the view from now on that the problem is to minimize barriers in some given loop nest. In a loop nest, the **depth** of any statement is the number of loops that contain it. The loop statement is itself a statement and has a depth: zero if it is the outermost loop. The nesting structure is a tree, with a node for each loop. The outermost loop is at the root, every other loop is a child of the loop that contains it. The **height** of a loop is zero if it is a leaf in the nesting tree, otherwise it is one greater than the height of its highest child.

We make the following assumptions:

- Loops have been normalized so that the loop counters are incremented by one. We don't really need this, but it allows us to simply write $i + 1$ when we mean the next value of the loop index i.

- Loops do not contain IF-THEN-ELSE statements. Otherwise we solve the barrier placement in each branch first (as O'Boyle and Stöhr do), before treating the rest of the loop nest. This is correct but sub-optimal. Therefore, our algorithm is optimal only for a loop nest with no dependences between statements in IF-THEN-ELSE.

- There are no zero-trip loops. This ensures that a barrier placed in the body of a loop L will enforce any dependence from a statement executed before L to another executed after L. Again, this assumption simplifies the discussion, but it is not really necessary for correctness. This because we can assume this property, solve the barrier placement problem, then re-analyze the program and determine those loops containing a barrier that enforces such a "long" dependence (from before the loop to after it) and that may possibly be zero-trip, and insert an alternative for the case where the loop does not execute, containing another barrier:

```
for (i = LB; i < UB; i++) { ... barrier; ...}
if (UB <= LB) {barrier;}
```

2.1 Barriers, temporal partial ordering, dependence relations, and correctly synchronized programs

Our problem is to place barriers to enforce interthread dependence relations. To reason about these, we need some preliminary notions. We denote by $S(\vec{i}_S)$ the **operation** that corresponds to the (static) statement S and the particular values of the loop counters, specified by the integer vector $\vec{i}_S$, for the loops, if any, in which S is nested. In an SPMD program, each operation $S(\vec{i}_S)$ has many **instances**, one for each thread that executes the portion of code that contains it. To distinguish between instances, we denote by $S(t_S, \vec{i}_S)$ the instance of $S(\vec{i}_S)$ executed by the thread whose number or identifier is t_S.

If statement instances s and t are executed by the same thread then we write $s \prec_{\text{seq}} t$ to indicate that s precedes t in sequential control flow. On the other hand, the barrier B synchronizes $S(t_S, \vec{i}_S)$ and $T(t_T, \vec{i}_T)$, instances from different threads, if there is an operation $B(\vec{i}_B)$ such that $S(t_S, \vec{i}_S) \prec_{\text{seq}} B(t_S, \vec{i}_B)$ and $B(t_T, \vec{i}_B) \prec_{\text{seq}} T(t_T, \vec{i}_T)$. The two individual barrier calls $B(t_S, \vec{i}_B)$ and $B(t_T, \vec{i}_B)$ are calls to the same operation $B(\vec{i}_B)$ of a single barrier B; because we target structurally correct programs, such calls always synchronize with one another.

For operations, let us write $S(\vec{i}_S) \prec_{\text{seq}} T(\vec{i}_T)$ if sequential control flow orders their instances on each individual thread. We say that the barrier B synchronizes operations $S(\vec{i}_S)$ and $T(\vec{i}_T)$ if there is an operation $B(\vec{i}_B)$ such that $S(\vec{i}_S) \prec_{\text{seq}} B(\vec{i}_B) \prec_{\text{seq}} T(\vec{i}_T)$. Formally, $S(\vec{i}_S) \prec_{\text{seq}} T(\vec{i}_T)$ is defined as follows. Let c be the number of loops that surround both of S and T. Then $S(\vec{i}_S) \prec_{\text{seq}} T(\vec{i}_T)$ if either $\vec{i}_S$ is lexicographically smaller than $\vec{i}_T$ in their first c components (those that refer to their common containing loops) or the two index vectors are equal in their first c components and S precedes T in the program text.

An interthread dependence relation R_{ST} between statements S and T is a set of pairs of operations. At least one of S or T is a write to a shared variable. For each pair $(S(\vec{i}_S), T(\vec{i}_T)) \in R_{ST}$, there is some barrier in the source code that synchronizes them. And finally, there are instances of $S(\vec{i}_S)$ and $T(\vec{i}_T)$, not both on the same thread, that reference the same shared variable, or at least we cannot determine at compile time that they do not, so they must be correctly ordered in time. From now on, when we talk of **dependences** we shall mean these interthread dependences. A barrier B **enforces** a dependence R if it synchronizes every pair of operations in the relation. In this case, we have $S(\vec{i}_S) \prec_{\text{seq}} B(\vec{i}_B) \prec_{\text{seq}} T(\vec{i}_T)$, in other words, the barriers in the given SPMD program define a temporal partial order (sub-order of the order $\prec_{\text{seq}}$) on operations, which determines the dependence relations.

2.2 Barriers, dependence level, and NCIF

For our purposes, it is enough to analyze dependence, find the instance relations, ignore the intrathread pairs, project each instance relation (that has interthread pairs) into a relation on operations, and determine the loop, if any, that carries it. We informally introduce these ideas here, and define things carefully later. For now, consider the SPMD program fragment:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
       b[i][j + m*mythread] = f(c[i][j + m*mythread]);
       if (i > 0) a[i][j + m*mythread] = b[i-1][g(j + m*mythread)];
    }
    barrier;
}
```

Consider the write of `b[i][j+m*mythread]` and the read of `b[i-1][g(j+m*mythread)]`. If the compiler cannot analyze the behavior of the indexing function `g`, it must assume that the thread that writes an element of `b` is different from the thread that reads this element – so this is an interthread (flow) dependence. The compiler can know, however, that the dependence relation consists of instances (s, t) for which the iteration vector if s is (i, j) and that of t is $(i + 1, j')$. Because the i loop is the outermost loop for which the dependent pairs occur in different loop iterations, we say that this loop carries the dependence and that the dependence is *loop-carried.*

The barrier in the example code enforces this dependence. There are other places where a barrier could be placed to do this. It could occur before the inner loop:

```
for (i = 0; i < n; i++) {
    barrier;
    for (j = 0; j < m; j++) {
       b[i][j + m*mythread] = f(c[i][j + m*mythread]);
       if (i > 0) a[i][j + m*mythread] = b[i-1][g(j + m*mythread)];
    }
}
```

It would also suffice to have a barrier in the inner loop:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
       b[i][j + m*mythread] = f(c[i][j + m*mythread]);
       if (i > 0) a[i][j + m*mythread] = b[i-1][g(j + m*mythread)];
       barrier;
    }
}
```

This solution might be overkill, however. Clearly there are more barriers executed (assuming $m > 1$) than for the other solutions. On the other hand, if there were some other dependence, carried by the j loop or not carried by any loop, that required a barrier inside the j loop, then this might be the best way to also enforce to dependence involving the array `b`. This is the case, for example, in the code hereafter. Note that the inner-loop barrier enforces a flow *loop independent* (i.e., not carried by any loop) dependence involving the array `a`, an antidependence on the array `c`, and also the flow dependence on the array `b`, by virtue of our certainty that the inner loop executes at least once for every iteration of the outer loop.

```
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
       b[i][j + m*mythread] = f(c[i][j + m*mythread]);
       if (i > 0) a[i][j + m*mythread] = b[i-1][g(j + m*mythread)];
       barrier;
       if (mythread > 0) c[i][j + m*mythread] = 2 * a[i][j + m*(mythread-1)]
    }
}
```

We now define more formally the relations between barrier placements and loop-carried/loop-independent dependences. We consider a properly nested region, which is turned into a single loop nest, as above. A set of dependences between statement instances is found by analysis of the given loop nest. [1] The dependence relations between statement instances are projected into a set of relations between operations, each of which is either loop-independent or is carried at some loop level, as described next.

Consider a dependence from operation $S(\vec{i})$ to $T(\vec{j})$: we know that $S(\vec{i}) \prec_{\text{seq}} T(\vec{j})$. Let c be the number of loops that surround both S and T; $\vec{i}$ and $\vec{j}$ have at least c components. We use the standard notion of dependence **level** [16]: if the first c components of $\vec{i}$ and $\vec{j}$ are equal, the dependence is **loop-independent** at level c, otherwise it is **loop-carried** at level k where $k \leq c$ is the largest integer such that the first $k-1$ components of $\vec{i}$ and $\vec{j}$ are equal. We view the statements of the program as laid out from the earliest (in program text order) on the left to the last on the right. Thus, "to the left of" and "leftmost" mean earlier and earliest (with respect to program text order). We describe the dependences as **circular intervals**, which we define below.

First consider the case of a loop-independent dependence. An example is depicted in Figure 1 from S to T, at level $c = 1$: a white box represents a DO, a grey box an ENDDO, the arrow from S to T represents the control flow. The dependence is represented by an open interval $]S, T[$ (see the

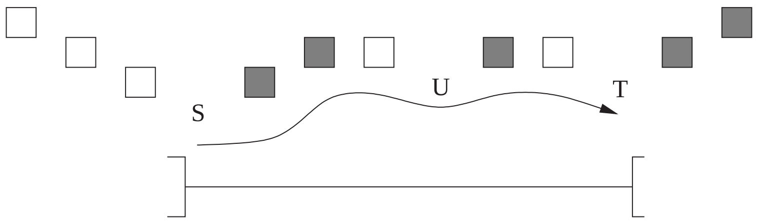

Figure 1: Interval for a loop-independent dependence (basic case).

bottom of Figure 1), and any barrier placed inside this interval enforces the dependence. All cases of loop-independent dependences can be represented by such an interval. For example, if we know that a loop containing S at depth $\geq c$ (i.e., not around T) executes at least once before the control flow goes to T, we represent the dependence with a larger interval from the DO of this loop to T (see Figure 2). If, likewise, a loop surrounding T iterates at least once before reaching T, then the interval is extended on the right to the appropriate ENDDO.

Now consider a loop-carried dependence. An example from S to T, of level $k = 2$, is depicted in Figure 3 where T strictly precedes S in the program text and $j_k = i_k + 1$. The control points where a barrier needs to be inserted (and any such control point is fine) can be represented by a **circular**

[1]The mechanism and the precision of dependence analysis is not the subject of this paper, so we will not go into any detail as to how the dependence relations are determined. We specify here how the dependences are represented, and analyze where barriers can be placed to enforce dependences.

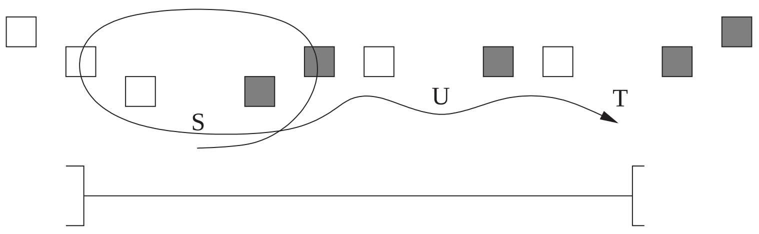

Figure 2: Case of an interval, for a loop-independent dependence, left-extended to a DO.

interval from S to T through the ENDDO and DO of the loop at depth $k-1$ shared by S and T. In the example, this means that any barrier insertion between S and the ENDDO of the second loop, or between the DO of the second loop and T enforces this dependence. If, on the other hand, k is 1, the interval would be extended through the ENDDO and DO of the first loop. Again, if we know more about additional iterations of a loop deeper than k surrounding either S or T, we may be able to use a wider circular interval, whose endpoints may be a DO earlier than S (the fourth DO in the example) or an ENDDO after T (the ENDDO of the third loop in the example).

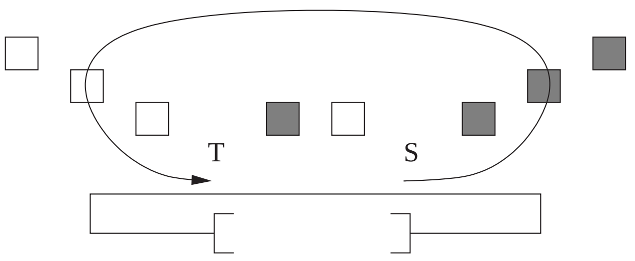

Figure 3: Circular interval for loop-carried dependence (basic case).

A **wrap-around** dependence, which spans more than one full iteration of loop k, where k is the level of the dependence, can also be represented by an open interval from the DO at depth $k-1$ to its ENDDO. Such a dependence can also simply be ignored if we know that the loop contains at least another dependence that will require a barrier anyway.

To summarize, we distinguish two types of dependence. A dependence can be:

Type A a loop-independent dependence at level k represented by an interval $]x, y[$ where x (resp. y) is a statement or a DO (resp. ENDDO), x is textually before y, and x and y are surrounded by exactly k common loops: a barrier needs to be inserted textually after x and before y, and any such barrier does the job.

Type B a loop-carried dependence represented by an interval $]x, y[$ and an integer k, where x (resp. y) is a statement or a DO (resp. ENDDO), x is textually after y, and they have at least k common loops: a barrier needs to be inserted textually after x and before the common surrounding ENDDO whose depth is $k-1$, or after the common surrounding DO whose depth is $k-1$ and before y, and any such placement is fine. (A wrap-around dependence is represented as a particular Type B dependence, from a DO to the corresponding ENDDO.)

Thus, our model of the barrier placement problem is a linear arrangement of control points and a set of circular intervals. We refer to such a model as a **nested circular interval family** (NCIF). A barrier placement is equivalent to a set of points (at which to insert barrier statements) between the control points of the NCIF. It is **correct** if each interval in the NCIF is "cut" by (i.e., contains) one or more barriers.

2.3 When is one solution better than another?

We represent the cost of a barrier placement P for a loop nest by a vertex-weighted tree $T = \text{cost}(P)$, whose structure is that of the nesting structure of the loop nest. Each vertex v (interior or leaf) has a weight $b(v)$ given by the number of barriers in the strict body of the loop (i.e., not in a deeper loop) to which v corresponds. Define a partial order $\preceq$ among tree costs as follows:

Definition 1 *Let T and U be the tree costs of two barrier placements for a loop nest. Let t and u be the roots of T and U, and $(T_i)_{1\leq i\leq n}$ and $(U_i)_{1\leq i\leq n}$ be the subtrees (rooted at the children of t and u) of T and U. We say that T is less than or equal to U (denoted $T \preceq U$) if*

- *$T_i \preceq U_i$, for each i, $1 \leq i \leq n$,*
- *if, for each i, $1 \leq i \leq n$, $T_i = U_i$, then $b(t) \leq b(u)$,*

If $T \preceq U$ and $T \neq U$, we say that T is less than U (denoted $T \prec U$).

Now we can compare barrier placements: P is better than Q if $\text{cost}(P) \prec \text{cost}(Q)$. We say that a barrier placement P is **optimal** if it is correct and is as good or better than every other correct barrier placement. This definition of optimality is not the same thing as saying "there is no placement better than this one." It asserts that an optimum cannot be incomparable with any other placement, but must be as good as or better than all others. Observe that the existence of optimal placements is not immediate, since the relation $\preceq$ is only a partial order. The next lemma shows that optimal placements always exist. Moreover, the recursive definition of $\preceq$ implies that, for a given loop L, all optimal placements have the same tree cost and that the restriction of any optimal placement for L to any loop L' contained in L is optimal for L'. We can therefore talk about *the* cost of a loop nest, defined to be the tree-cost of any optimal placement.

Lemma 1 *For any two solutions P and Q, there is a solution as good or better than both P and Q. Consequently, optimal solutions exist.*

Proof. The proof is by induction on the height of the loop, i.e., the number of nested loops it contains.

For a loop L of height 0, i.e., for an innermost loop, P is as good or better than Q if P places no more barriers in L than Q. Thus, any two solution costs are comparable, and either P is better than Q (so use P), or the converse (use Q), or they are equally good (use either).

For a loop L of height $h > 0$, containing the loops $(L_i)_{1\leq i\leq n}$, consider two solutions P and Q such that P is not as good or better than Q and Q is not as good or better than P (otherwise, there is nothing to prove), i.e., two solutions whose tree costs are not comparable by $\preceq$. Let T and U be their respective tree costs, and P_i and Q_i be the restrictions of P and Q to L_i, with tree costs T_i and U_i. By definition of $\preceq$, there exist j and k, perhaps equal, such that $T_j \not\preceq U_j$ and $U_k \not\preceq T_k$. By the induction hypothesis, there exist solutions R_i for every subtree L_i, as good or better than both P_i and Q_i. In particular, each R_i is a correct placement for L_i, therefore they enforce all dependences not carried by L and not lying in the body of L. We can extend the local solutions R_i to a solution R for L by placing a barrier after each statement in the body of L (this is brute force, but enough for what we want to prove). We have $\text{cost}(R_i) \preceq T_i$ for all i, and $\text{cost}(R_j) \prec T_j$ (indeed, $\text{cost}(R_j) = T_j$ is not possible since this would imply $T_j \preceq U_j$). Thus R is better than P. Similarly R is better than Q.

What we just proved is correct even if we restrict to the finite set of solutions that place in each loop at most as many barriers as statements plus one (i.e., one barrier between any two statements). Therefore, the fact that any two solutions have a common as good or better solution implies that there are optimal solutions. ■

Note that two placements with the same tree cost (even if they differ in the exact position of barriers inside the loops) lead to the same dynamic barrier count. The key point is that to get an optimal placement for a nest, one must select the right set of optimal placements for the contained loops. Consider the example in Figure 4 with dependences from G to A (carried by the outer

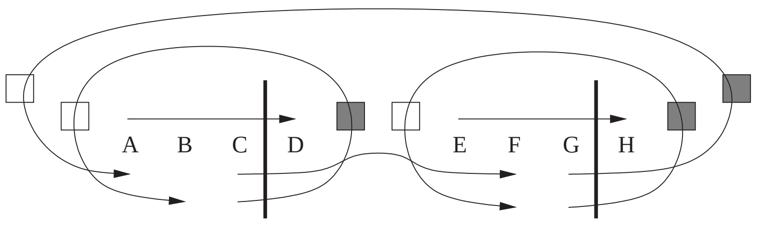

Figure 4: A 2D example and its (unique here) optimal placement.

loop, i.e., with $k = 1$) and from C to F (loop-independent at level 1). The dependences internal to the inner loops are (A, D) and (C, B), as well as (E, H) and (G, F). These allow for two local optima for each of the inner loops: a barrier may be placed just before B or just before D, and just before F or just before H. Clearly, there are four possible combinations of two local optima, but only the choice of barriers just before D and just before H leads to a global optimal, because with this choice (uniquely) of local optima, no barriers are needed at depth 1.

For completeness, let us point out that if every loop iterates at least twice whenever encountered, an optimal placement executes the smallest possible number of barriers among all correct placements.

Lemma 2 *If each loop internal to the nest iterates at least twice for each iteration of the surrounding loop, then an optimal solution minimizes, among all correct placements, the number of barrier calls that occur at runtime.*

Proof. It suffices to show that if Q, with tree cost U, is not optimal (in terms of $\preceq$), then there exists a better solution P, with tree cost $T \prec U$, such that P does not induce more dynamic barriers than Q.

Consider Q a solution for L with tree cost U, not optimal with respect to $\preceq$. Let L' be a loop of minimal height such that the restriction of Q to L', with tree cost U', is not optimal. By construction, U' is a subtree of U and all subtrees of U' are optimal tree costs for their corresponding subloops. Furthermore, for the solution Q, the number of barriers in the strict body of L' (i.e., $b(u)$ where u is the root of U') is strictly larger than in any optimal solution for L'. Replace in Q the barriers in L' (i.e., in L' and deeper) by the barriers of any optimal solution for L'. This gives a partially correct solution: all dependences are enforced except maybe some dependences that enter L' or leave L'. To enforce them, add a barrier just before L' and a barrier just after L', so as to get a new correct solution P. The tree cost T of P is obtained by replacing in U the subtree U' by the optimal subtree T' of L'. The root t of T' is such that $b(t) \leq b(u) - 1$.

By construction, P is better than Q in terms of $\preceq$ since $b(t) < b(u)$. It remains to count the number of dynamic barriers induced by P and Q. There is no difference between P and Q, in terms of tree cost, for loops inside L'. So they have the same dynamic cost. This is the same for all other loops, except for the strict body of L' and for the loop strictly above L'. Consider any iteration of this loop: the difference between the number of dynamic barriers for Q and the number of dynamic barriers for P is $N(b(u) - b(t)) - 2$ where N is the number of iterations of L' for this particular iteration of the surrounding loop. Since $N \geq 2$ and $b(t) - b(u) \geq 1$, P does not induce more dynamic barriers than Q. ■

3 Inner-loop barrier minimization

In this section, we recall results for one-dimensional cases: the case of a straight-line code and the case of a single innermost loop.

For a straight-line code (i.e., no loop), a simple greedy linear-time algorithm does the job: Find the first (leftmost) right endpoint of any interval, and cut with a barrier just to the left of this endpoint. Repeat while any uncut intervals remain. This technique was used by Quinn and Hatcher [9] and by O'Boyle and Stöhr [13]. Next, these authors leverage this process to get a quadratic-time algorithm for simple loops: Try each position in the loop body for a first barrier, which cuts the circle making it a line; next apply the linear-time algorithm above to get the remaining barriers; and finally choose the solution with the fewest barriers.

Surprisingly, it seems that none of the previous work on barrier placement recognized that the problem for a straight-line code is nothing but the problem of finding a minimum clique cover in an interval graph. (The algorithm of [9, 13] is exactly the well-known greedy algorithm for this problem [8]). Generalized to an inner loop, the problem is to find a minimum *linear* clique in a circular interval family, for which there exists a very simple linear-time (thus better than quadratic) algorithm [10]. Our technique to solve optimally the case of general loops and to reduce the complexity of the barrier placement algorithm (even in cases for which an optimal algorithm has already been given) is based on this linear-time algorithm for finding a minimum linear clique cover in a circular interval family. We introduce these concepts and explain the corresponding algorithms next.

3.1 Straight-line code and minimum clique cover of an interval graph

In a straight-line code, only loop-independent dependences exist (dependences of type A). They correspond to intervals $I_i =]h_i, t_i[$ where h_i and t_i are integers such that $h_i < t_i$, i.e., intervals on a line. The classical graph associated to intervals on a line is the so-called interval graph, an undirected graph with a vertex per interval and an edge between two intervals that intersect.

An *independent set* is a set of intervals, such that no two of them intersect. A *clique* is a set of intervals that defines a complete subgraph in the corresponding interval graph, i.e., a set of intervals, each pair of which intersect. In such a clique, consider an interval I_i with largest head (i.e., largest h_i) and an interval I_j with least tail (i.e., least t_j). By definition, the interval $]h_i, t_j[$ is not empty (since I_i and I_j intersect) and is contained in each interval of the clique. Any point z in this interval belongs to all intervals in the clique; if a barrier is placed at z, it enforces (or "cuts") all intervals of the clique. Conversely, any barrier defines a clique, which is the set of intervals enforced by this barrier (they all intersect since they all contain the point where the barrier is placed). Such a clique is called a *linear* clique.

We just showed that any clique in an interval graph is a linear clique and that any barrier corresponds to a linear clique. Thus, finding an optimal barrier placement amounts to find a minimum (linear) clique cover, i.e., a set of cliques, of smallest cardinality, such that each interval belongs to at least one of these cliques. Consider an optimal barrier placement, i.e., a minimum clique cover and modify it as follows. Move the first (i.e., the leftmost) barrier as much as possible to the right, while keeping correctness, i.e., place it just before the first tail of any interval. Do the same for the second barrier, move it as much as possible to the right, i.e., place it just before the first tail of any interval not already enforced by the first barrier. Do the same for all remaining barriers, one after the other, until all intervals are enforced. This mechanism leads to a correct barrier placement, with the same number of barriers, thus optimal. Furthermore, this solution can be found in a greedy manner, in linear time, provided that the endpoints t_i are sorted, as in Algorithm 1. Since barriers are placed just before the tail of independent intervals, this also shows that the maximum size $\alpha(G)$ of an independent set in an interval graph G is equal to the minimum size $\theta(G)$ of a clique cover (of course $\alpha(G) \leq \theta(G)$ for any graph G).

Algorithm 1 Barrier placement for a straight-line code

```
Input: $\mathcal{I}$ is a set of $n \geq 1$ intervals $I_i = ]h_i, t_i[$, $1 \leq i \leq n$, with $h_i < t_i$ and $i \leq j \Rightarrow t_i \leq t_j$
Output: an optimal barrier placement for $\mathcal{I}$
  procedure GREEDY($\mathcal{I}$)
      $i = 1$
      repeat
          $z = t_i$                                   ▷ tail of the first uncut barrier so far
          insert a barrier just before $z$
          repeat
              $i = i + 1$
          until ($i > n$) or ($h_i \geq z$)           ▷ until one finds an uncut barrier
      until $i > n$
  end procedure
```

3.2 Inner-loop barrier minimization and the Hsu-Tsai algorithm

A **circular interval family** (CIF) [2] is a collection $\mathcal{F}$ of open subintervals of a circle in the plane, where points on the circle are represented by integer values, in clockwise order. Each circular interval I_i in $\mathcal{F}$ is defined by two points on the circle as $]h_i, t_i[$, where h_i and t_i are integers, and represents the set of points on the circle lying in the clockwise trajectory from h_i to t_i. For example, on the face of a clock, $]9, 3[$ is the top semicircle. By convention, $]t, t[\cup \{t\}$ represents the full circle.

Two circular intervals that do not overlap are **independent**. A set of intervals is independent if no pair overlaps; let $\alpha(\mathcal{F})$ be the maximum size of an independent set in $\mathcal{F}$. A set of intervals, each pair of which overlaps, is a clique and, if they all contain a common point z, is a **linear clique**. In this case, they can be cut (by a barrier) at the point z. Note that in a circular interval family there can be nonlinear cliques: take, for example, the intervals $]0, 6[$, $]3, 9[$, and $]8, 2[$. A set of linear cliques such that each interval belongs to at least one of these cliques, is a **linear clique cover**; let $\theta(\mathcal{F})$ (resp. $\theta_l(\mathcal{F})$) be the minimum size of a clique cover (resp. linear clique cover). It is easy to see that the problem of finding the smallest set of barriers that enforces all dependences in an inner loop is equivalent to the problem of finding a minimum linear clique cover (**MLQC**) for the CIF $\mathcal{F}$ given by the dependences. It is important to note that, as long as the intersections of intervals (and thus cliques) are concerned, a circular interval family is fully described by the clockwise ordering of the endpoints of the intervals, i.e., the exact value and position of endpoints is not important.

The MLQC problem for an arbitrary CIF was solved with a linear-time algorithm – $O(n \log n)$ if the endpoints are not sorted; ours are, given the program description – by Hsu and Tsai [10]. We

[2] The graph algorithms literature also uses the term circular arc graph for the graph with an edge between two overlapping intervals. Hsu and Tsai use the term circular arc family (CAF) for the set of circular intervals.

use this fast solver as the basis of our algorithm for solving the nested loop barrier minimization problem. Let us summarize here how it works.

To make explanations simpler, let us assume first, as Hsu and Tsai do, that all the endpoints of the intervals in $\mathcal{F}$ are different. Given an interval $I_i =]h_i, t_i[$, define NEXT(i) to be the integer j for which $I_j =]h_j, t_j[$ is the interval whose **head** h_j is contained in $]t_i, t_j[$ and whose **tail** t_j is the first encountered in a clockwise traversal from t_i. The function NEXT defines a directed graph $D = (V, E)$, whose vertex set V is $\mathcal{F}$ (the set of intervals) and E is the set of pairs of intervals (I_i, I_j) with $j = \text{NEXT}(i)$. The out-degree of every vertex in D is exactly one; therefore, D is a set of directed "trees" except that in these trees, the root is a cycle. An important property is that any vertex with at least one incoming interval in D (it is the NEXT of another interval) is **minimal** meaning that it does not contain any other interval in $\mathcal{F}$. Hsu and Tsai define GD(i) to be the maximal independent set of the form $I_{i_1}, \dots, I_{i_k}$, with $i_1 = i$, and $i_t = \text{NEXT}(i_{t-1})$, $2 \le t \le k$, and they let LAST(i) = NEXT(i_k).

Theorem 1 (Hsu and Tsai [10]) *Any interval I_i in a cycle of D is such that $GD(i)$ is a maximum independent set, and so $|GD(i)| = \alpha(\mathcal{F})$. Furthermore, if $\alpha(\mathcal{F}) > 1$, placing a barrier just before the tail of each interval in $GD(i)$, and if $LAST(i) \neq i$, an extra barrier just before the tail of $LAST(i)$, defines a minimum linear clique cover, which is also a minimum clique cover.*

Algorithm 2 Barrier placement for an inner loop

```
Input: $\mathcal{F}$ is a set of $n \ge 1$ circular intervals $I_i = ]h_i, t_i[$, $1 \le i \le n$, such that $i \le j \Rightarrow t_i \le t_j$
Output: NEXT$(i)$ for each interval $I_i$ and a MLQC for $\mathcal{F}$, i.e., an optimal barrier placement
    procedure HsuTsai($\mathcal{F}$)
        $i = 1$; $j = i$
        for $i = 1$ to $n$ do
            if $i = j$ then                ▷ $i$, current interval, may have "reached" $j$, current potential next
5:              $j = \text{Inc}(i, n)$           ▷ Inc$(i, n)$ is equal to $i + 1$ if $i < n$, and 1 otherwise
            end if
            while $h_j \notin [t_i, t_j[$ do                              ▷ intervals still overlap
                $j = \text{Inc}(j, n)$           ▷ Inc$(j, n)$ is equal to $j + 1$ if $j < n$, and 1 otherwise
            end while
10:         NEXT$(i) = j$; MARK$(i) = 0$
        end for                             ▷ at this point, NEXT$(i)$ is computed for all $i$
        $i = 1$                  ▷ start the search for a cycle, could start from any interval actually
        while MARK$(i) = 0$ do
            MARK$(i) = 1$; $i = \text{NEXT}(i)$
15:     end while                 ▷ until we get back to some interval (cycle is detected)
        $j = i$
        repeat
            insert a barrier just before $t_j$; $j = \text{NEXT}(j)$                  ▷ intervals in GD$(i)$
        until $I_i$ and $I_j$ overlap
20:     if $j \neq i$ then
            insert a barrier just before $t_j$                  ▷ special case for LAST$(i) \neq i$
        end if
    end procedure
```

If $\alpha(\mathcal{F}) = 1$, $\mathcal{F}$ is a clique, so that $\theta(\mathcal{F}) = 1$ as well. Thus, Theorem 1 shows that for a circular interval family, $\theta(\mathcal{F})$ is either $\alpha(\mathcal{F})$ or $\alpha(\mathcal{F}) + 1$. It gives a way to construct an optimal barrier placement for inner loops. It also gives a constructive mechanism to find a minimum clique cover when $\alpha(\mathcal{F}) > 1$, and this clique cover is even formed by linear cliques. In Algorithm 2, Lines 1–11 compute the function NEXT for each interval, the last lines from 12 compute GD(i) and place the barriers accordingly. The test [3], Line 7, is satisfied if $i = j$ thus the case where NEXT(i) $= i$ is taken into account correctly. The fact that tails are in increasing order is used to start the search for NEXT($i + 1$) from NEXT(i). This implies that j traverses at most twice all intervals and that the algorithm has linear-time complexity. To make the study complete, it remains to consider two special cases: a) what happens when $\alpha(\mathcal{F}) = 1$, b) what happens when some endpoints are equal.

Lemma 3 *When $\alpha(\mathcal{F}) = 1$, Algorithm 2 is still valid to find a minimum linear clique cover.*

Proof. When $\alpha(\mathcal{F}) = 1$, $\mathcal{F}$ itself is a clique, and $\alpha(\mathcal{F}) = \theta(\mathcal{F}) = 1$. But what about a minimum *linear* clique cover? Let us show that, actually, the procedure in Theorem 2 still leads to a minimum linear clique cover, for any interval I_i in a cycle of D, so Algorithm 2 is still correct for optimal barrier placement.

If one barrier is necessary (i.e., $\mathcal{F}$ is nonempty) and sufficient to cut all intervals (i.e., if there is a linear clique cover of size 1), consider such a barrier and let t_i be the first tail encountered in a clockwise traversal from this barrier. Let j be any other interval. In a clockwise traversal from h_j, one gets the barrier, then t_j (since I_j is cut by the barrier). Furthermore, t_i occurs between the barrier and t_j, by definition of i, thus between h_j and t_j. Thus NEXT(i) $= i$. Conversely, if I_i is such that NEXT(i) $= i$, place a barrier just before t_i. By definition of NEXT, the head h_j of any interval I_j is not in $[t_i, t_j[$, thus t_i belongs to I_j, i.e., I_j is cut by the barrier. In this case, one barrier is enough, and thus optimal.

To show that Algorithm 2 is correct, we need to prove more: we need to prove that if there is a cycle of length 1, then any cycle is of length 1 so that the number of barriers placed by the algorithm does not depend upon the choice of the cycle in Lines 12–16. Assume this is not the case and consider two intervals I_i and I_j, with NEXT(i) $= i$ and NEXT(j) $\neq j$, and such that $]t_i, t_j[$ does not contain the tail of an interval in a cycle of D. Since I_j is cut by a barrier just before t_i, we get h_j, then t_i, and t_j in a clockwise traversal from h_j. Consider $k =$ NEXT(j); then $k \neq i$ (otherwise I_j is not in a cycle). By choice of i and j, t_j appears before t_k in a clockwise traversal from t_i. Thus, in a clockwise traversal from t_i, one finds t_i, t_j, h_k, t_k, t_i, but this is impossible since I_k is cut by the barrier before t_i.

It remains to consider the case where D does not contain a cycle of length 1. In this case, we know that at least 2 barriers are needed (previous study) and that for any interval I_i in a cycle of D, NEXT(i) $\neq i$. Since NEXT(i) overlaps with i ($\alpha(\mathcal{F}) = 1$), NEXT(i) $=$ LAST(i) and Algorithm 2 will thus place 2 barriers, one just before t_i and one just before t_j with $j =$ LAST(i). It remains to prove that this barrier placement is correct. Assume the converse and let I_k be an interval, not cut by any of these 2 barriers. By definition of j, t_k cannot appear between t_i and t_j in a clockwise traversal from t_i (otherwise it is cut by the barrier before t_i), therefore t_k is between t_j and t_i. Then, h_k must be in $]t_j, t_i[$ also, otherwise I_k is cut by one of the barriers. But since $j =$ NEXT(i) and I_j and I_i overlap, in a clockwise traversal from t_i, we get t_i, h_j, t_j, h_i, h_k, t_k, t_i, and I_k is contained in I_i, which is not possible since I_i is in a cycle of D, thus minimal. ■

Lemma 4 *Algorithm 2 is correct even if not all endpoints are different.*

Proof. It is easy to see that from any set $\mathcal{I}$ of open circular intervals $I_i =]h_i, t_i[$, one can build a set $\mathcal{I}'$ of open circular intervals $I'_i =]h'_i, t'_i[$, all endpoints being different, which needs the same minimum number of barriers. For that, it suffices to sort the endpoints following a total order $\prec$

[3]The test is equivalent to $h_j \in]t_i, t_j[$ since endpoints are all different, but we use $h_j \in [t_i, t_j[$ instead to handle the case of equal endpoints correctly; further discussion of equal endpoints follows shortly.

among points that keeps the original strict inequalities (i.e., $p \prec q$ whenever $p < q$, p and q head or tail) and places tails before heads in case of equality (i.e., $p \prec q$ if $p = q$, p is a tail and q is a head).

Given a barrier placement for $\mathcal{I}$, one can get a barrier placement for $\mathcal{I}'$, with same number of barriers, as follows. First, move each barrier, in the clockwise order, and place it just before the first encountered tail. Then, for each barrier b placed just before t_i in $\mathcal{I}$, place a barrier b' in $\mathcal{I}'$ *just before any* tail that corresponds to the value t_i in $\mathcal{I}$ (thus in particular after any head in $\mathcal{I}'$ that corresponds to a head in $\mathcal{I}$ strictly before t_i in clockwise order). It is easy to see that if I_j is cut by b in $\mathcal{I}$, then I'_j is cut by b' in $\mathcal{I}'$. The converse is obviously true.

As for Algorithm 2, one can first change $\mathcal{I}$ into $\mathcal{I}'$ so that all endpoints are different. But, as already noted, only the relative positions of the endpoints according to $\prec$ matters. Algorithm 2 works implicitly with the order $\prec$. Heads are considered after tails in case of equality (because intervals are open) thanks to the test $h_j \notin [t_i, t_j[$ (instead of $h_j \notin]t_i, t_j[$), Line 7. Also, in case of equality, tails are considered in some fixed order so that the function NEXT is defined in a coherent way, the order $\prec$ given by the input. Note also that when $j = i$ in Line 7, we need to go out of the loop because NEXT(i) is indeed equal to i. This is correct since $h_i \in [t_i, t_i[$ (full circle), even if $h_i = t_i$, while this would not be correct with the test $h_j \notin]t_i, t_j[$ for the very particular case of a single interval $]t, t[$. This shows that Algorithm 2 is correct in all cases. ■

4 Optimal barrier placement in nested loops of arbitrary structure

The setting now is a loop nest of depth two or more. An algorithm for optimal barrier placement is known only for a semiperfect (only one loop in the body of any other loop) loop nest. Here, we provide such an algorithm for a nest of any nesting structure.

If a barrier placement is optimal with respect to the hierarchical tree cost of Section 2.3, then it places a smallest allowable number of barriers in each innermost loop. The number of barriers in the strict body of a loop L of height ≥ 1 is the smallest possible among all correct barrier placements for L whose restriction to each loop that L contains is optimal for the contained loop. As optimality is defined "bottom-up," it is natural to begin to try to solve the problem that way.

4.1 Basic bottom-up strategy

Before explaining our algorithm, let us consider a basic (in general sub-optimal) bottom-up strategy. A similar strategy is used by O'Boyle and Stöhr to handle the cases that are not covered by their optimal algorithm, i.e., the programs with IF-THEN-ELSE or loops containing more than one inner loop. This strategy is optimal for innermost loops but, except by chance, not for loops of height ≥ 1.

To place barriers in a loop L, Algorithm 3 places barriers in each inner loop L' first (Line 5). For L', only the dependences that cannot be cut by a barrier in L are considered (the set $\mathcal{D}'$), in other words, in L', only the the essential constraints are considered. Then, depending on the placement chosen for L', it may happen that, in addition to dependences in $\mathcal{D}'$, some others, entering L' (i.e., with tail in L') or leaving L' (i.e., with head in L'), are cut by an inner barrier (Line 6). These dependences need not be considered for the barrier placement in L (Line 7). Next, any remaining dependence that enters (resp. leaves) a deeper loop must be changed to end before the DO (resp. start after the ENDDO) of this loop (Lines 8 and 9), because it must be cut by a barrier in L. Finally, the modified L is handled as an inner loop (Line 11).

Algorithm 3 yields an optimal placement if each loop has a single optimal placement or if, by chance, it picks the right optimal placement at each level. The problem is therefore to modify Algorithm 3 so that it can select judiciously, among the optimal placements for contained loops, those

Algorithm 3 Bottom-up heuristic strategy for barrier placement in a loop nest

Input: A loop nest L, and a set $\mathcal{D}$ of dependences, each with a level
Output: A correct barrier placement, with minimal number of barriers in each innermost loop

```
1: procedure BottomUp($L$, $\mathcal{D}$)
2:     for all loops $L'$ included in $L$ do
3:         let $u_0$ and $v_0$ correspond to the DO and ENDDO of $L'$
4:         $\mathcal{D}' = \{d = (u, v) \in \mathcal{D} \mid u \in L', v \in L', \text{level}(d) > \text{depth}(L')\}$    ▷ need to be cut in $L'$
5:         BottomUp($L'$, $\mathcal{D}'$)    ▷ give a barrier placement in $L'$
6:         CUT $= \{d \in \mathcal{D} \mid d$ cut by a barrier in $L'\}$    ▷ $\mathcal{D}' \subseteq$ CUT
7:         $\mathcal{D} = \mathcal{D} \setminus$ CUT
8:         for each $d = (u, v) \in \mathcal{D}$, $v \in L'$ do $v = u_0$.    ▷ dependence enters $L'$
9:         for each $d = (u, v) \in \mathcal{D}$, $u \in L'$ do $u = v_0$.    ▷ dependence leaves $L'$
10:    end for
11:    HsuTsai($\mathcal{D}$)    ▷ or any other algorithm optimal for a single loop
12: end procedure
```

that cut (Line 6) incoming and outgoing dependences so that the number of barriers determined in L (Line 11) for the remaining dependences (Line 7) is minimized. Our main contribution is to explain how to do this, and, moreover, how to do it efficiently.

4.2 Summarizing inner loop barrier placements: weaving/unraveling

To get the optimal placement for an outer loop, one needs to be able to determine the right optimal placement for each loop L it contains. In particular, one needs to understand how dependences that come into L or go out of L are cut by an optimal placement in L. Our technique is to capture (as explained next) how barriers in L interact with these incoming and outgoing dependences.

Let us first define precisely what we call an incoming, an outgoing, or an internal dependence. A dependence $d = (u, v)$ is **internal** for a loop L if it *needs* to be cut by a barrier inside L (in the strict body of L or deeper), i.e., if $u \in L$, $v \in L$, and $\text{level}(d) > \text{depth}(L)$. The set of internal dependences for L determines the minimal number of barriers for L. Incoming and outgoing dependences for a loop L are dependences that *may* be cut by a barrier inside L, but can also be cut by a barrier in an outer loop: they are not internal for L, but have either their tail in L (**incoming** dependence) or their head in L (**outgoing** dependence). An incoming dependence is cut by a barrier placement for L if there is a barrier between the DO of L and its tail. An outgoing dependence is cut by a barrier placement for L if there is a barrier between its head and the ENDDO of L. Note that a dependence $d = (u, v)$ can be simultaneously incoming and outgoing for a loop L, when $u \in L$, $v \in L$, and $\text{level}(d) \leq \text{depth}(L)$. For such a dependence, when we say that, considered as an incoming dependence, it is *not* cut by a barrier placement for L, we mean that there is no barrier between the DO and the tail of the dependence, even if there is a barrier between its head and the ENDDO (and conversely when the dependence is considered as outgoing). This precision is important to correctly (and with a brief explanation) handle such dependences.

Let L be an innermost loop, with internal dependences represented by a CIF $\mathcal{F}$. Let $\theta_l(\mathcal{F})$ be the number of barriers in any optimal barrier placement for L, or equivalently the size of an MLQC for $\mathcal{F}$. We can find $\theta_l(\mathcal{F})$, and optimal placements, with the Hsu-Tsai algorithm. Each optimal barrier placement for L is a set of barriers placed at precise points in the loop body; obviously, one of these inserted barriers is the leftmost and one of them is the rightmost. Let d be an incoming dependence that can be cut by some optimal barrier placement for L. Denote by RIGHTMOST(d) the rightmost point before which a barrier is placed in an optimal barrier placement for L that cuts d. (This will be the tail of d, the tail of an internal dependence, or the ENDDO of L.) If d and d' are two incoming dependences, with the tail of d to the left of the tail of d', then RIGHTMOST(d) is to the left of RIGHTMOST(d') (they are possibly equal). We will explain later how we can compute the function RIGHTMOST in linear time for all incoming dependences.

To capture the influence of the inner loop L on the barrier placement problem for its parent loop, the key idea is that the inner solution is determined by the rightmost incoming and the leftmost outgoing dependences that it cuts. The same information can be gleaned if we change the tail of each incoming dependence d to RIGHTMOST(d), remove the intervals internal to L, then "flatten" the NCIF by raising the body of L to the same depth as the DO and ENDDO, meaning that in defining an optimal placement for this flattened NCIF, the tree cost function treats barriers between the DO and ENDDO as belonging to the tree node of the parent of L. If L had some internal dependences, an interval from DO to ENDDO is added in their place, guaranteeing that a barrier will be placed between them. This operation, which we call **weaving**, is described in Algorithm 4. After weaving an innermost loop L for an NCIF $\mathcal{F}$, we obtain a new NCIF $\mathcal{F}'$ that corresponds to a nest with same tree structure as $\mathcal{F}$ except that the leaf node of L is gone.

Algorithm 4 Weaving of an innermost loop

```
Input: An innermost loop $L$, a set $\mathcal{D}$ of internal dependences, $\mathcal{D}_{\text{in}}$ of incoming dependences, $\mathcal{D}_{\text{out}}$ of outgoing
    dependences. (Reminder: $\mathcal{D}_{\text{in}} \cap \mathcal{D}_{\text{out}}$ may be nonempty.)
Output: Modify incoming and outgoing dependences, and return a special dependence $d_L$.
    procedure WEAVING($L$, $\mathcal{D}$, $\mathcal{D}_{\text{in}}$, $\mathcal{D}_{\text{out}}$)
        let $u_0$ and $v_0$ be the DO and ENDDO of $L$ (statements in the parent loop of $L$)
        for all $d = (u, v) \in \mathcal{D}_{\text{in}}$ do
            if $d$ is not cut by any optimal barrier placement in $L$ then
5:              $v = u_0$                                   ▷ change its tail to the DO of $L$
            else                                            ▷ summarize the rightmost solution
                $v$ = RIGHTMOST($d$)     ▷ new endpoint considered as a statement in the parent loop of $L$
                if $d$ is also in $\mathcal{D}_{\text{out}}$ and $v$ is now to the right of $u$ then    ▷ possible only if $d \in \mathcal{D}_{\text{in}} \cap \mathcal{D}_{\text{out}}$
                    $u = v$                     ▷ new wrap-around dependence, represented as $]v, v[$
10:             end if
            end if
        end for
        for all $d = (u, v) \in \mathcal{D}_{\text{out}}$ do
            if $d$ is not cut by any optimal barrier placement in $L$ then
15:             $u = v_0$                                   ▷ change its head to the ENDDO of $L$
            end if
        end for
        if $\mathcal{D} \neq \emptyset$ then
            create a new dependence $d_L = (u_0, v_0)$, loop independent at level depth($L$)
20:         return $d_L$
        else
            return $\perp$
        end if
    end procedure
```

Assume we generate an optimal placement for the flattened NCIF. The process to go from an optimal placement P' for $\mathcal{F}'$ to an optimal placement P for $\mathcal{F}$ is called **unraveling**. The idea is to find the optimal barrier placement in the body of L that cuts the same incoming and outgoing intervals as were cut by those in P'. Unraveling works as follows. In P', there will be either zero, one, or two barriers between the DO and ENDDO of L (considered as statements in the parent loop of L); not more, because barriers after DO and before ENDDO suffice to cut the special interval d_L (Line 19 in Algorithm 4) and all transformed incoming and outgoing intervals. If zero, then no barriers are needed in L. If two, the one to the left can be moved to just before the DO (so it cuts all incoming intervals) with no loss of correctness. Thus, we can assume there is one. It may occur just before ENDDO (i.e., the tail of d_L), in which case we would select the rightmost optimal solution for L. Or it may occur before the tail of an incoming interval d, which in the original NCIF $\mathcal{F}$ had a different tail. The inner solution we need is then the rightmost one that cuts this incoming dependence in $\mathcal{F}$, i.e., whose leftmost barrier is to the left of the original tail of d, because it will cut exactly the same set of intervals in $\mathcal{F}$ as were cut by the one barrier in $\mathcal{F}'$. This is the unraveling process. The following theorem shows more formally that this weaving/unraveling process is correct.

Theorem 2 *Weaving an innermost loop and unraveling the resulting placement produces an optimal placement.*

Proof. Let $\mathcal{F}'$ be obtained from $\mathcal{F}$ by weaving L. The codes corresponding to $\mathcal{F}$ and $\mathcal{F}'$ are equal except that, in the code for $\mathcal{F}'$, the innermost loop L has been replaced by simple statements, those which correspond to the new tails defined Line 7 of Algorithm 4. We assume that L has at least one internal dependence, otherwise it is clear that $\mathcal{F}$ and $\mathcal{F}'$ are equivalent representation of the dependences since L does not contain any barrier in an optimal barrier placement for $\mathcal{F}$.

Let P be an optimal barrier placement for $\mathcal{F}$. A barrier placement Q for $\mathcal{F}'$ is obtained as follows. First place all barriers in P, which are outside L, at the same place in Q. This cuts all dependences of $\mathcal{F}'$ that correspond to dependences of $\mathcal{F}$ cut by a barrier outside L. Now, add an extra barrier in Q as explained next. In P, the placement of the barriers in L is an optimal placement for L. Consider the leftmost incoming dependence d in $\mathcal{F}$ cut by this inner placement and place in Q a barrier just before its (new) tail in $\mathcal{F}'$ defined Line 7. If d does not exist, place a barrier in Q just before the tail of the new special dependence d_L defined Line 19. It is easy to see that Q is a valid barrier placement for $\mathcal{F}'$. Indeed, this additional barrier cuts d_L, it cuts any dependence that "flows above" L, it cuts any incoming dependence not cut outside L in P since it cuts the leftmost such dependence (the new tail of such a dependence is to the right of this additional barrier because of the non-decreasing property of RIGHTMOST), and it cuts all outgoing dependences not cut outside L in P (i.e., cut by the inner placement) thanks to the definition of the new tails, Line 7.

Conversely, consider an optimal barrier placement Q for $\mathcal{F}'$. The special dependence d_L (defined in Line 19 of Algorithm 4) is cut by some barrier in Q. Consider b the rightmost such barrier and move it as much as possible to the right without changing the way dependences are cut: b is now just before the tail of some dependence d (note that $d = d_L$ is possible), and by construction, it corresponds to the rightmost possible barrier placement in an optimal solution that cuts d (or in a rightmost solution for L if $d = d_L$). Define a barrier placement P for $\mathcal{F}$ by first placing barriers in L according to such a rightmost solution. Then, place all other barriers in P as they are in Q, except that each barrier ($\neq b$) in Q that cuts d_L is moved to the left just before the DO of L (otherwise this would increase the number of barriers in L) [4]. It is easy to see that the barriers in P cut all dependences in $\mathcal{F}$.

This proves that there is direct correspondence between optimal solution for $\mathcal{F}$ and $\mathcal{F}'$: weaving a non-trivial (i.e., with some internal dependences) innermost loop L has the following effects:

- it removes the leaf corresponding to L in the tree cost;

[4]In our implementation however, this case will never happen, see explanations hereafter, after Lemma 7.

- in the tree cost, it adds 1 to the father of the removed leaf, i.e., the inner solution for L is represented by an additional barrier in the loop that surrounds it.

This enables us to "swallow" leaves of the tree, one by one, until the tree is a simple leaf, i.e., corresponds to a CIF. ■

Algorithm 5 Optimal algorithm for barrier placement in a NCIF – bottom-up pass

```
Input: A loop $L$, with a set $\mathcal{E}$ of dependences, each with at least one endpoint in $L$
Output: a dependence $d_L$ that "summarizes" $L$ (and incoming/outgoing dependences are modified)
  procedure OptimalBottomUp($L$, $\mathcal{E}$)
      $\mathcal{D} = \{d = (u,v) \in \mathcal{D} \mid u \in L,\ v \in L,\ \text{level}(d) = \text{depth}(L) + 1\}$      ▷ exclude incoming/outgoing
      for all loop $L'$ included in $L$ do
          $\mathcal{E}' = \{d = (u,v) \in \mathcal{E} \mid u \in L' \text{ or } v \in L'\}$      ▷ internal, incoming, or outgoing
          $d_{L'} = \text{OptimalBottomUp}(L', \mathcal{E}')$
          $\mathcal{D} = \mathcal{D} \cup \{d_{L'}\}$      ▷ add special dependence, unless no barrier in $L'$ (when $d_{L'} = \perp$)
      end for
      $\mathcal{D}_{\text{in}} = \{d = (u,v) \in \mathcal{E} \setminus \mathcal{D} \mid v \in L\}$; $\mathcal{D}_{\text{out}} = \{d = (u,v) \in \mathcal{E} \setminus \mathcal{D} \mid u \in L\}$
      return Weaving($L, \mathcal{D}, \mathcal{D}_{\text{in}}, \mathcal{D}_{\text{out}}$)
  end procedure
```

The weaving/unraveling process leads to a two-passes algorithm, a first bottom-up pass for weaving loops, a second top-down pass for unraveling them. To summarize, we find the optimal placement for a loop nest as follows. First build its NCIF model. Then weave (and remove) innermost loops one at a time until one loop with a simple CIF model remains (see Algorithm 5 for the bottom-up phase). Use the Hsu-Tsai method to find an optimal placement for it. Then successively apply the unraveling process to inner loops in a top-down manner until an optimal placement for the entire nest is obtained. We illustrate this process below on two examples.

Consider again the example of Figure 4. The first innermost loop L_1 has 2 internal dependences $d_1 = (A, D)$ and $d_2 = (C, B)$. All optimal placements have one barrier. The rightmost places a barrier just before D (which cuts the only outgoing dependence $d_4 = (C, F)$); the only incoming dependence $d_3 = (G, A)$ is not cut by any optimal placement thus the weaving procedure moves its tail to the DO of L_1. We introduce a new dependence d_{L_1} to capture the rightmost placement from the DO to the ENDDO of L_1 (remembering that if a barrier is placed just before the tail of d_{L_1} for barrier placement in an outer loop, this means placing a barrier just before D in the inner loop). For the second innermost loop L_2, the situation is the same for internal dependences, one barrier is enough, and the rightmost placement is with a barrier just before H. However, this time, the incoming dependence d_4 is cut by an optimal placement and RIGHTMOST(d_4) is the tail of d_4 (so no change of tails is needed here, this is a particular case). A new dependence d_{L_2} is introduced similarly. The simple CIF obtained after weaving both inner loops is depicted in Figure 5.

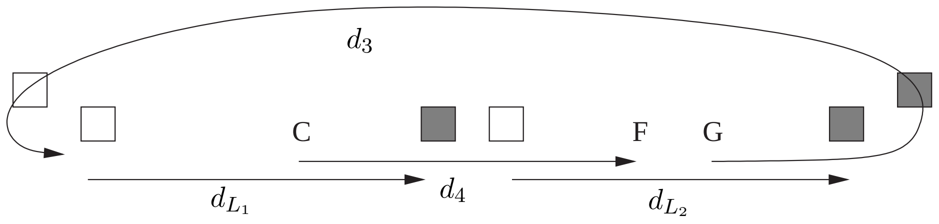

Figure 5: Woven CIF for the NCIF of Figure 4.

We have NEXT$(d_3) = d_{L_1}$, NEXT$(d_{L_1}) = d_{L_2}$, NEXT$(d_{L_2}) = d_{L_1}$, and NEXT$(d_4) = d_3$. Therefore, the Hsu-Tsai algorithm tells us that two barriers are needed, one before the tail of d_{L_1}, one before the tail of d_{L_2}. The unraveling procedure interprets this, following Theorem 2, as using the rightmost placement for L_1, i.e., placing a barrier just before D, and the rightmost placement for L_2, i.e., placing a barrier just before H, as depicted in Figure 4.

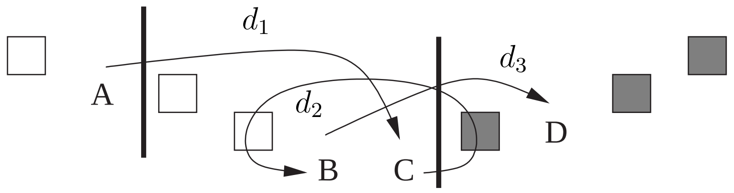

Figure 6: A 3D example from O'Boyle and Stöhr.

Consider now an example of O'Boyle and Stöhr [13], Figure 6. Only one barrier is needed in the innermost loop L_1 for the internal (loop-carried) dependence $d_2 = (C, B)$. The rightmost placement places a barrier just before the ENDDO of L_1. This cuts the outgoing dependence $d_3 = (B, D)$. The incoming dependence $d_1 = (A, C)$ can also be cut by an optimal placement in the innermost loop, with a (rightmost) barrier before B – so d_1 is (A, B) now – but in this case, d_3 is not cut. Therefore, weaving the innermost loop leads to the NCIF in Figure 7.

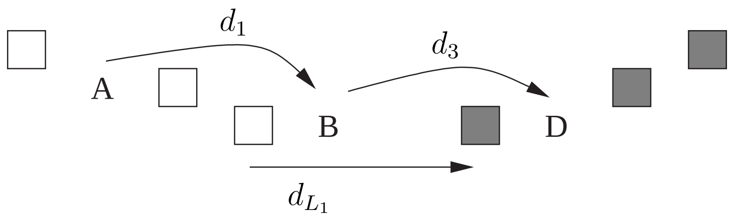

Figure 7: Woven NCIF for the NCIF of Figure 6.

Now, the innermost loop L_2 has 2 internal dependences, d_{L_1} and d_3, and only one barrier is needed. The incoming dependence d_1 cannot be cut by an optimal placement (if a barrier cuts d_1, it cannot cut d_3). Thus, weaving L_2 leads to the simple CIF in Figure 8. Two barriers are needed, one before the tail of d_2, i.e., just before the DO of the second loop, and one before the tail of d_{L_2}. This second barrier is interpreted as the rightmost placement for L_2, i.e., a barrier just before the tail of d_{L_1}. This one again is interpreted as the rightmost placement for L_1, i.e., a barrier just before the ENDDO of this loop. The final barrier placement, in Figure 6, has one barrier at depth 3 and one barrier at depth 1. This solution is optimal: it has lower tree cost than the alternative, barriers before B (depth 3) and D (depth 2).

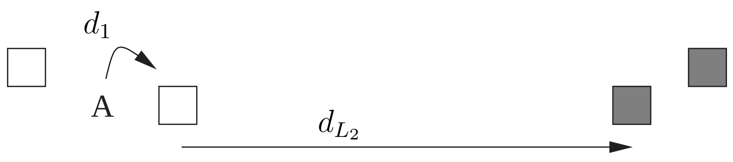

Figure 8: Woven CIF for the NCIF of Figure 7.

In these two examples, the recursive calls to the top-down unraveling barrier placement were always done with the special dependences d_L (i.e., the rightmost placement in each inner loop was always selected). This is not always the case. It may happen that the recursive call is done with an incoming dependence d that indicates the rightmost optimal placement that cuts d. For example, if in the NCIF of Figure 6, d_1 ends strictly after C, then it can be cut by an optimal placement for L_1 (with a barrier just before its tail) that cuts all dependences. In Figure 7, d_1 and d_3 will then overlap, and an optimal placement for L_2 will cut both. The tail of d_1 will not be moved to the DO, so in Figure 8, d_1 and d_{L_2} will overlap, and d_1 will be selected by the Hsu-Tsai algorithm, with only one barrier needed. This barrier will be interpreted as the rightmost placement for L_2 that cuts d_1, i.e., with a barrier before the tail of d_1, and this barrier will be interpreted deeper as the rightmost placement for L_1 that cuts d_1, i.e., with a single barrier before the tail of d_1.

4.3 A linear-time algorithm to compute the function RIGHTMOST

In Algorithm 4, we did not explain how to compute RIGHTMOST(d) for an incoming dependence d for a loop L and, in particular, the rightmost solution among all optimal solutions that cut d too. An obvious (but inefficient) strategy is as follows. First, compute the minimal number of barriers for L using the Hsu-Tsai algorithm (Algorithm 2) applied to $\mathcal{F}$, the internal dependences for L. Then, add $d = (u, v)$ to $\mathcal{F}$, reasoning as if it starts just after the DO (i.e., $u = u_0$), and run Algorithm 2 again. If one extra barrier is needed, d can never be cut by an optimal solution for L and we are in the case of Line 5. Otherwise, add to $\mathcal{F} \cup \{d\}$, one at a time, each outgoing dependence $e = (w, x)$, reasoning as if it ends just before the ENDDO (i.e., $x = v_0$), and run Algorithm 2 again. If one extra barrier is needed, the outgoing dependence cannot be cut by an optimal solution for L that also cuts d. If all outgoing dependences can be cut this way, run Algorithm 2 again with an extra "outgoing" dependence that starts and ends just before the ENDDO of L to capture the possibility of a rightmost barrier just before the ENDDO. This way, we can identify RIGHTMOST(d) by finding the outgoing dependence with rightmost head that is cut by an optimal solution for L that cuts d too. The total complexity is $O(n^3)$ – $O(n^2)$ calls to Algorithm 2 – to compute RIGHTMOST(d) for all incoming dependences d.

To get a linear-time algorithm for optimal barrier placement for a NCIF, the previous strategy is not enough. We need to be able to compute the function RIGHTMOST (for a non empty CIF $\mathcal{F}$) in linear time for all incoming dependences. For that, we analyze more precisely the structure of rightmost placements in a CIF.

We start with an elementary property, similar to the main property of Hsu and Tsai (Theorem 1). We use the notations of Section 3. Remember that $\alpha(\mathcal{F})$ is the maximum size of an independent set in $\mathcal{F}$ and $\theta_l(\mathcal{F})$ is the minimum size of a linear clique cover for $\mathcal{F}$, which is also the optimal number of barriers for $\mathcal{F}$.

Lemma 5 *For any minimal interval I_i in $\mathcal{F}$, placing a barrier just before the tail of each interval in GD(i), and if LAST(i) $\neq i$, an extra barrier just before the tail of LAST(i), defines a valid barrier placement P_i. If LAST(i) $= i$, then $|GD(i)| = \theta_l(\mathcal{F}) = \alpha(\mathcal{F})$ and P_i is optimal. Furthermore, if $\theta_l(\mathcal{F}) = \alpha(\mathcal{F}) + 1$ then, for any minimal interval I_i, LAST(i) $\neq i$, $|GD(i)| = \theta_l(\mathcal{F}) - 1$, and P_i is optimal.*

Proof. The sequence GD(i) is defined as $I_{i_1}, \dots, I_{i_k}$ with $i_1 = i$ and $i_t = \text{NEXT}(i_{t-1})$, $2 \leq t \leq k$, and LAST$(i) = \text{NEXT}(i_k)$ overlaps with I_i. Let us prove that P_i is valid. Any interval in $\mathcal{F}$ whose head is between (clockwise) the tail of $I_{i_{t-1}}$ and the tail of I_{i_t} has its tail after the tail of I_{i_t} by definition of the function NEXT. Therefore it is cut by the barrier placed just before the tail of I_t. Similarly, when LAST$(i) = i$, an interval whose head is between the tail of I_{i_k} and the tail of I_i is cut by the barrier placed just before the tail of I_i and, when $\mathrm{LAST}(i) \neq i$, an interval whose head is between the tail of I_{i_k} and the tail of $\mathrm{LAST}(i) = \mathrm{NEXT}(i_k)$ is cut by the barrier placed just before the tail of $\mathrm{LAST}(i)$. In this latter case, it remains to consider an interval whose head is between the tail of $\mathrm{LAST}(i)$ (which belongs to I_i) and the tail of I_i. Since I_i is minimal, the tail of such an interval must be after the tail of I_i and therefore is cut too. This proves that P_i is valid.

Since P_i is valid, we get $|\mathrm{GD}(i)| \geq \theta_l(\mathcal{F})$ if $\mathrm{LAST}(i) = i$ and $|\mathrm{GD}(i)| + 1 \geq \theta_l(\mathcal{F})$ otherwise. Furthermore, $|\mathrm{GD}(i)| \leq \alpha(\mathcal{F})$ since $\alpha(\mathcal{F})$ is the maximum size of an independent set and $\alpha(\mathcal{F}) \leq \theta_l(\mathcal{F})$. These inequalities show that $\theta_l(\mathcal{F}) = \alpha(\mathcal{F}) = |\mathrm{GD}(i)|$ whenever there exists a minimal interval I_i such that $\mathrm{LAST}(i) = i$. Conversely, this means that if $\theta_l(\mathcal{F}) = \alpha(\mathcal{F}) + 1$, then, for any interval I_i, $\mathrm{LAST}(i) \neq i$. And, in this latter case, we have $|\mathrm{GD}(i)| \leq \alpha(\mathcal{F}) < \theta_l(\mathcal{F}) \leq |\mathrm{GD}(i)| + 1$, i.e., $|\mathrm{GD}(i)| = \theta_l(\mathcal{F}) - 1$, and P_i is optimal since it uses $|\mathrm{GD}(i)| + 1 = \theta_l(\mathcal{F})$ barriers. ■

For each loop-independent interval I_i, we define $\mathrm{GDR}(i)$ the maximal sequence $I_{i_1}, \ldots, I_{i_n}$ of independent intervals such that $i_1 = i$, $i_t = \mathrm{NEXT}(i_{t-1})$ for $2 \leq t \leq n$, and the tail of I_{i_t} is to the right of the tail of $I_{i_{t-1}}$: $\mathrm{GDR}(i)$ is similar to $\mathrm{GD}(i)$ (it is a subset) except that we stop the sequence when we have to go back to the beginning of the loop (GDR stands for GD to the Right). All intervals in $\mathrm{GDR}(i)$ are loop-independent. We define $\mathrm{RIGHT}(i) = i_n$ and $\mathrm{LENGTH}(i) = n$. The functions RIGHT and LENGTH can be computed, for all intervals in $\mathcal{F}$, in linear time. Indeed, we just propagate values for RIGHT and LENGTH *backwards*, in the graph D defined by the function NEXT, starting from the loop-independent intervals whose NEXT is to the left of them, thanks to the relation $\mathrm{RIGHT}(i) = \mathrm{RIGHT}(\mathrm{NEXT}(i))$ and $\mathrm{LENGTH}(i) = \mathrm{LENGTH}(\mathrm{NEXT}(i)) + 1$.

To identify the rightmost placement for a CIF $\mathcal{F}$ for a loop L, we first check whether an optimal placement with a barrier just before the ENDDO of L exists. For that, define $\mathrm{FIRST}(\mathcal{F}) = i$ such that I_i is the loop-independent interval with leftmost tail in $\mathcal{F}$, and let $n = \mathrm{LENGTH}(i)$ (if I_i does not exist, $\mathcal{F}$ has only loop-carried intervals and we let $n = 0$). When $n \geq 1$, I_i is minimal by construction. Let $j = \mathrm{RIGHT}(i)$ and $k = \mathrm{NEXT}(j)$. If I_k is loop-independent then, by definition of $\mathrm{FIRST}(\mathcal{F})$, $k = i$, thus $\mathrm{GDR}(i) = \mathrm{GD}(i)$ and, according to Lemma 5, $n = |\mathrm{GD}(i)| = \theta_l(\mathcal{F}) = \alpha(\mathcal{F})$. If I_k is loop-carried, then two cases are possible. If I_k does not overlap with I_i then, by definition of $\mathrm{FIRST}(\mathcal{F})$, $\mathrm{NEXT}(k) = i$ thus $|\mathrm{GD}(i)| = |\mathrm{GDR}(i)| + 1 = n + 1$ and, according to Lemma 5, $n + 1 = \theta_l(\mathcal{F}) = \alpha(\mathcal{F})$. If I_k overlaps with I_i, then $\mathrm{LAST}(i) = k \neq i$, $\mathrm{GDR}(i) = \mathrm{GD}(i)$, and, according to Lemma 5, if $\theta_l(\mathcal{F}) = \alpha(\mathcal{F}) + 1$, then $n = \theta_l(\mathcal{F}) - 1$, otherwise n can be either $\theta_l(\mathcal{F})$ or $\theta_l(\mathcal{F}) - 1$.

Lemma 6 *A loop L with a CIF $\mathcal{F}$ has an optimal barrier placement (with $\theta_l(\mathcal{F})$ barriers) with a barrier just before the ENDDO if and only if $n = \theta_l(\mathcal{F}) - 1$, where $i = FIRST(\mathcal{F})$ and $n = LENGTH(i)$. In this case, we get a rightmost placement by placing a barrier just before the tail of each interval in GDR(i), plus a barrier just before the ENDDO of L.*

Proof. If $\mathcal{F}$ has only loop-carried intervals (i.e., $n = 0$), then $\theta_l(\mathcal{F}) = 1$ and a barrier just before the ENDDO does cut all intervals in $\mathcal{F}$. Otherwise, let $i = \mathrm{FIRST}(\mathcal{F})$ and $n = \mathrm{LENGTH}(i)$. We add to $\mathcal{F}$ (virtually, just for the reasoning) a new loop-independent interval $I_j = (u, v)$, where u and v are both to the right of any other endpoint in $\mathcal{F}$. $\mathcal{F}$ has an optimal barrier placement with a barrier just before the ENDDO if and only if $\theta_l(\mathcal{F})$ barriers are sufficient to cut all intervals in $\mathcal{F}' = \mathcal{F} \cup \{I_j\}$, i.e., iff $\theta_l(\mathcal{F}) = \theta_l(\mathcal{F}')$.

By construction, we have $\mathrm{NEXT}(\mathrm{RIGHT}(i)) = j$ and $\mathrm{NEXT}(j) = i$. Thus, these intervals form a cycle in the graph D' defined by the function NEXT for $\mathcal{F}'$, which shows, thanks to Theorem 1, that $n + 1$ barriers are needed for $\mathcal{F}'$. Furthermore, placing one barrier just before the tail of each interval of $\mathrm{GD}(i)$ (defined in $\mathcal{F}'$), i.e., one barrier just before the tail of each interval in $\mathrm{GDR}(i)$

(defined in $\mathcal{F}$) and one just before the ENDDO (the tail of I_j), is an optimal solution for $\mathcal{F}'$. No additional barrier is needed compared to $\mathcal{F}$ if and only if $n = \theta_l(\mathcal{F}) - 1$. ■

Lemma 7 *If a loop L with a CIF $\mathcal{F}$ has no optimal barrier placement with a barrier just before the ENDDO of L, a rightmost placement is obtained by placing a barrier just before the tail of each interval in GD(i) (plus an extra barrier before the tail of LAST(i) if LAST(i) $\neq$ i) where I_i is the interval with rightmost tail in a cycle of D, the graph defined by the function NEXT.*

Proof. Let $i = \text{FIRST}(\mathcal{F})$, $n = \text{LENGTH}(i)$. To identify the rightmost point in an optimal solution for $\mathcal{F}$, we introduce, as in the previous lemma, a new loop-independent interval $I_j = (u, v)$ where v is just before the ENDDO of the loop (i.e., to the right of any other endpoint in $\mathcal{F}$) and we identify the rightmost position for u for which $\mathcal{F}' = \mathcal{F} \cup \{I_j\}$ needs only $\theta_l(\mathcal{F})$ barriers and not $\theta_l(\mathcal{F}) + 1$. Let D' be the graph defined by the function NEXT for $\mathcal{F}'$. Note that $i = \text{FIRST}(\mathcal{F}')$ and $\text{NEXT}(j) = i$.

Suppose that $\theta_l(\mathcal{F})$ barriers are enough for $\mathcal{F}'$, i.e., $\theta_l(\mathcal{F}) = \theta_l(\mathcal{F}')$. Since there is no optimal solution for $\mathcal{F}$ with a barrier just before the ENDDO, $n = \theta_l(\mathcal{F})$ (Lemma 6). Thus, I_j does not belong to a cycle of D', otherwise $n + 1 = \theta_l(\mathcal{F}) + 1$ barriers would be needed following GD(j), and possibly LAST(j), i.e., $\{I_j\} \cup \text{GDR}(i)$. Therefore, I_j is cut because its head u is to the left of the tail of some interval in a cycle of D'. Adding I_j to $\mathcal{F}$ can only change the NEXT of some intervals in $\mathcal{F}$, those whose NEXT in $\mathcal{F} \cup \{I_j\}$ are now j. Therefore, since I_j is not in a cycle of D', any interval in a cycle of D' was already in a cycle of D (the converse may not be true however). This proves that u is to the left of the tail of some interval in a cycle of D. Conversely, if this is the case, there is an optimal solution for $\mathcal{F}$ that cuts also I_j, thus $\mathcal{F}'$ needs only $\theta_l(\mathcal{F})$ barriers.

In other words, the rightmost barrier in an optimal barrier placement for $\mathcal{F}$ is just before the rightmost tail of an interval in a cycle of D. There is no need to consider other intervals. ■

To study the optimal barrier placements for $\mathcal{F}$ in a loop L with respect to an incoming dependence, i.e., a dependence whose tail v is in L, we treat it as an internal dependence $I_i = (u, v)$ for L, where u is just to the right of the DO of L (i.e., to the left of any other endpoint in $\mathcal{F}$) and we study $\mathcal{F}' = \mathcal{F} \cup \{I_i\}$, thanks to Lemmas 6 and 7. Below, we assume that I_i does not contain an interval in $\mathcal{F}$ (i.e., is minimal in $\mathcal{F}'$), otherwise it is always cut by an optimal barrier placement for $\mathcal{F}$, and the rightmost such placement can be found thanks to Lemmas 6 and 7 applied to $\mathcal{F}$. Note that if I_i is minimal in $\mathcal{F}'$, then $i = \text{FIRST}(\mathcal{F}')$.

Remark: we can now explain the footnote of Page 18. Apply the previous lemmas to $\mathcal{F}'$, assuming that $\mathcal{F}'$ needs also $\theta_l(\mathcal{F})$ barriers, i.e., $\theta_l(\mathcal{F}) = \theta_l(\mathcal{F}')$. When $n = \theta_l(\mathcal{F}') - 1$, the rightmost barrier placement consists in placing a barrier just before the tail of each interval in GDR(i), plus a barrier just before the ENDDO. Since an interval of the form d_L is, by construction, minimal and loop-independent, it is going to be cut only once by such a barrier placement. When $n = \theta_l(\mathcal{F}')$ and $\theta_l(\mathcal{F}') = \alpha(\mathcal{F}')$, then we will place barriers just before the tails of a sequence GD(j) of independent intervals, thus again, an interval d_L can be cut only once. The case $n = \theta_l(\mathcal{F}')$ and $\theta_l(\mathcal{F}') = \alpha(\mathcal{F}') + 1$ is not possible as seen from the different cases analyzed previously (see properties just before Lemma 6).

Thanks to Lemmas 6 and 7, we now have almost everything we need to find in linear time, for each incoming dependence I_i, the rightmost optimal barrier placement for $\mathcal{F}$ that cuts it. We just need to define RIGHT(i), LENGTH(i), LAST(i), and LASTCUT(i) (we don't update these functions for intervals in $\mathcal{F}$, this would be more costly and useless anyway) and to show how to use them. We first compute j = NEXT(i) in $\mathcal{F}'$. If I_j is loop-independent and to the right of I_i, we let RIGHT(i) = RIGHT(j), LENGTH(i) = LENGTH(j) + 1. Otherwise, we let RIGHT(i) = i

and LENGTH(i) $= 1$. Then, if RIGHT(i) $\neq i$, we consider $k =$ NEXT(RIGHT(i)) as defined in $\mathcal{F}$ (otherwise, $k = j$). Since the head of I_i is before the tail of any interval in $\mathcal{F}$, either the tail of I_k is to the right of the tail of I_i and LAST(i) $= i$, or LAST(i) $= k$ (I_k is then loop-carried since I_i is minimal in $\mathcal{F}'$). We also compute LASTCUT(i) $= l$ such that I_l belongs to a cycle of D and the tail of I_l is the rightmost tail to the left of the tail of I_i (the interval I_l may not exist).

Computing the functions LASTCUT and NEXT for all incoming intervals can be done in linear time, with an algorithm similar to what we did in Algorithm 2 for the function NEXT, provided that internal intervals and incoming intervals are sorted by increasing tails. Given these functions, the next theorem shows how to determine, in constant time, whether an incoming interval can be cut by an optimal placement for $\mathcal{F}$ and, if this is the case, where is the rightmost barrier.

Theorem 3 *Let I_i be an incoming dependence for a loop L with a CIF $\mathcal{F}$ and let $\theta_l(\mathcal{F})$ be the minimal number of barriers for $\mathcal{F}$. If I_i contains an interval of $\mathcal{F}$, then a rightmost placement for $\mathcal{F}$ cuts I_i. Otherwise:*

- *If LAST(i) $= i$ and LENGTH(i) $= \theta_l(\mathcal{F})$, I_i is cut by an optimal placement for $\mathcal{F}$ with barriers before the tails of intervals in GDR(i), the rightmost one just before RIGHT(i).*
- *If LAST(i) $\neq i$ and LENGTH(i) $= \theta_l(\mathcal{F})-1$, I_i is cut by an optimal placement for $\mathcal{F}$, barriers before the tails of intervals in GDR(i), plus a rightmost barrier just before the ENDDO.*
- *If LAST(i) $\neq i$ and LENGTH(i) $\geq \theta_l(\mathcal{F})$, I_i can be cut by an optimal placement for $\mathcal{F}$ if and only if $j =$ LASTCUT(i) exists. In this case, barriers are just before the tails of intervals in GD(j), the rightmost barrier being just before the tail of I_k in GD(j) where NEXT(k) $= j$.*

In all other cases, I_i cannot be cut by an optimal barrier placement for $\mathcal{F}$.

Proof. Consider I_i the representation of an incoming dependence as an internal interval and assume that I_i is minimal in $\mathcal{F}' = \mathcal{F} \cup \{I_i\}$. We have $i =$ FIRST($\mathcal{F}'$). We have $n-1 \leq \theta_l(\mathcal{F}) \leq n+1$, where $\theta_l(\mathcal{F})$ is the minimal number of barriers for $\mathcal{F}$ and $n =$ LENGTH(i) ≥ 1.

Suppose first that LAST(i) $= i$. In this case, the sequence GDR(i) $=$ GD(i) forms a cycle in the graph D' defined by the function NEXT for $\mathcal{F}'$. According to Theorem 1, $\mathcal{F}'$ needs n barriers (thus $n \geq \theta_l(\mathcal{F})$). If $n = \theta_l(\mathcal{F}) + 1$, I_i cannot be cut by an optimal barrier placement for $\mathcal{F}$. If $n = \theta_l(\mathcal{F})$, it can be cut and, according to Lemmas 6 and 7 applied to $\mathcal{F}'$, the rightmost barrier is just before the rightmost tail of an interval I_j in a cycle of D' and not just before the ENDDO. Let $k =$ NEXT(j) in $\mathcal{F}'$. I_k cannot be loop-carried otherwise LAST(i) $\neq i$ (the tail of RIGHT(i) is to the left of (or equal to) the tail of I_j, its NEXT would be loop-carried too). Thus, NEXT(j) $= i$ and finally, following the function NEXT, $j =$ RIGHT(i). Therefore, the case LAST(i) $= i$ is complete: either $n = \theta_l(\mathcal{F}) + 1$ and I_i cannot be cut by an optimal solution for $\mathcal{F}$, or $n = \theta_l(\mathcal{F})$ and the rightmost barrier is just before the tail of RIGHT(i).

Now suppose that LAST(i) $\neq i$. If $n = \theta_l(\mathcal{F}) - 1$, according to Lemma 6 applied to $\mathcal{F}'$, the barrier placement defined from GDR(i), plus a barrier just before the ENDDO of the loop, is a rightmost solution for $\mathcal{F}$ that cuts I_i too. If $n \geq \theta_l(\mathcal{F})$, suppose that $\mathcal{F}'$ needs $\theta_l(\mathcal{F})$ barriers too (i.e., I_i can be cut by an optimal solution for $\mathcal{F}$). According to Lemmas 6 and 7, the rightmost barrier is just before the rightmost tail of an interval I_j in a cycle of D' and not before the ENDDO. But I_i does not belong to a cycle of D' otherwise, according to Theorem 1, $\mathcal{F}'$ needs $n+1$ barriers, i.e., more than $\mathcal{F}$. Therefore, with the same reasoning as for Lemma 7, I_j belongs to a cycle of D and I_i is cut by a barrier just before the tail of I_k with $k =$ NEXT(j). Thus LASTCUT(i) exists. Furthermore, I_j is the unique interval in a cycle of D such that NEXT(j) $=$ LASTCUT(i). Indeed, consider I_l whose tail is to the right of the tail of I_j. Either NEXT(l) $= k$ and then I_l is not in a cycle of D' since two different intervals in a cycle cannot have the same NEXT, or NEXT$(l) = i$ and again I_l is not in a cycle of D' since I_i is not in a cycle of D'. Conversely, if LASTCUT(i) exists, in this clear that $\mathcal{F}'$ needs only $\theta_l(\mathcal{F})$ barriers. Therefore, the case LAST$(i) \neq i$ is complete too: either $n = \theta_l(\mathcal{F}) - 1$ and there is a solution with a rightmost barrier just before the ENDDO, or $n \geq \theta_l(\mathcal{F})$ and there is a solution if and only if LASTCUT(i) exists and the rightmost barrier is just before the tail of I_j such that I_j is in a cycle of D and NEXT$(j) =$ LASTCUT(i). ■

Thanks to this theorem, we can find an optimal barrier placement for an NCIF in linear time. During the whole weaving/unraveling process, each interval is examined a constant time for every loop that it enters or leaves, and a constant time in the loop for which it is internal (as it will eventually be, once inner loops are woven). The overall complexity is therefore $O(nd)$ where n is the number of intervals and d the height of the nest. If the endpoints of each interval are represented by vector of dimension equal to the depth of each statement (so as to precise in each loop it belongs), the complexity is $O(n)$, where n is the size of the input.

5 Conclusion

We have presented a fast algorithm that solves the barrier minimization problem. As with most claims for optimality in programming optimization, ours is true (at least we believe it) up to the assumptions and definitions we have made. Other techniques, including statement reordering, loop fusion and distribution, and other loop transformations, can affect the synchronization cost, and ultimately the runtime, of parallel code. Some dependences can be enforced by point-to-point synchronization at possibly lower cost that with a barrier. Removing barriers may change the load balance characteristics of a program. Thus, considerable experience will be required to determine the best combination of optimizations for practical application of the tools for parallel program optimization that this and other theoretical research provide.

References

[1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA'89)*, pages 396–406. ACM Press, 1989.

[2] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL'98)*, pages 342–354. ACM Press, 1998.

[3] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (PoPL'87)*, pages 63–76. ACM Press, 1987.

[4] Andrea C. Arapaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing'93*, pages 262–273. ACM Press, 1993.

[5] C. D. Callahan. *A global approach to detection of parallelism.* PhD thesis, Rice University, 1987.

[6] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999. See also: upc.nersc.gov.

[7] Co-Array Fortran. `http://www.co-array.org/`.

[8] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* New York: Academic Press, 1980.

[9] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, 1991.

[10] Wen-Lian Hsu and Kuo-Hui Tsai. Linear time algorithms on circular-arc graphs. *Information Processing Letters*, 40(3):123–129, 1991.

[11] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[12] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998. See also: www.co-array.org.

[13] Michael O'Boyle and Elena Stöhr. Compile time barrier synchronization minimization. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):529–543, 2002.

[14] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. In *PPoPP'95: Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155. ACM Press, 1995.

[15] Unified Parallel C. `http://upc.gwu.edu/`.

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

[17] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, Sept-Nov 1998.

Understanding the weaving process. Photography: Vincent Moncorge, May 2004.