A Linear-Time Algorithm for Optimal Barrier Placement

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

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

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 interval families. This result solves the barrier minimization problem for general nested loops.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*

General Terms

Algorithms, Languages, Theory

Keywords

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

1. 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 [10], Co-Array Fortran [3]). 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 [7]. All threads wait for the slowest. Even if all arrive together, latency grows as $\log n$ with n threads.

Careful programmers may do a good job of placing barriers. But experimental evidence indicates that in automatically parallelized code there is a lot of room to improve the barrier placement and reduce synchronization cost [8]. Moreover, it is difficult 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 [5] and discuss algorithms for inner loops but not more complicated program regions. O'Boyle and Stöhr [8] make several interesting contributions. Extending the work of [5], 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.

A longer report, with proofs of all results and some fuller explanations than permitted by conference paper format, is available [4].

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*). 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. When optimizing the placement of barriers in the program, we shall move some and delete other barriers; we therefore change the boundaries of epochs. But we make sure that any two memory operations issued by different threads, that access the same location, and whose temporal order was enforced by a barrier in the original code, are still ordered by a barrier in the optimized code. To do this, we carefully define **interthread dependences** (see Section 2.1).

Enforcing all interthread dependences is, in general, not enough for correctness. Indeed, after we move or delete barriers, two memory operations of a single thread that occur in separate epochs in the original program may occur in the same epoch in the optimized program. In a relaxed consistency model, this might change the order in which the effect of these operations (on one thread) becomes visible to other threads, and this might lead to unforeseen errors. So we shall assume a sequentially consistent shared-memory model here.

Another concern is the interaction of barriers and other synchronization primitives, such as post/wait and the implementation of wait as a while loop on a shared semaphore. Consider this example:

```
S1:   x = 0;
B1:   barrier;
      if (mythread == 0) {
S2:       while (x == 0) {}
      }
      if (mythread == 1) {
S3:      x = 1
      }
```

The interthread dependences enforced by a barrier are from S1 to S2 and from S1 to S3. If an optimizer were to insert a barrier as follows:

```
S1:   x = 0;
B1:   barrier;
      if (mythread == 0) {
S2:       while (x == 0) {}
      }
B*:   barrier;
      if (mythread == 1) {
S3:      x = 1
      }
```

this would result in deadlock. Thread 0 is now in an infinite while loop and will never emerge, will never reach B*. By placing barriers where they were not located in the original code, it is possible to make such a correct program incorrect. The problem here is caused by the dependence from the write at S3 to the reads at S2. Unlike the interthread dependences enforced by barriers, this one is from a predecessor statement (S3) to a successor (S2) that comes earlier in the sequential program execution order. It is this situation that makes it unsafe to interpolate the barrier. In the example, we could remove this possibility by simply reversing the order of the two if-statements.

Thus, we must limit the kinds of synchronization, other than barriers, allowed in our model. The unoptimized code may not synchronize by testing in a while loop the value of a shared variable whose value is changed by a different thread, in a statement that comes later in the sequential execution order.

Alternatively, we would need to extend the dependence analysis model to capture these "backwards" interthread dependences and thereby find the points in the program where barriers cannot safely be placed, and perhaps eliminate these possibilities by an allowable reordering of statements. In lieu of such an analysis, we could extend the language model and allow a programmer to create "barrier free" code regions into which the optimizer cannot place any barrier.

Following Aiken and Gay, we also 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. In particular, while threads may use locks to protect code regions, sequentializing accesses to certain data objects, no barrier may occur in any lock-protected program region. Structural correctness may be a language requirement [12]. We can also optimize programs in looser languages if we discover at compile time that they have no structural correctness violations.

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.

To simplify the discussion, 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.

- For optimality, we assume that loops don't 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. Thus, 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, 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 $S(\vec{i}_S) \prec_{\text{seq}} B(\vec{i}_B) \prec_{\text{seq}} T(\vec{i}_T)$.

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.

2.2 Barriers, Dependence Level, and NCIF

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** [11]: 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

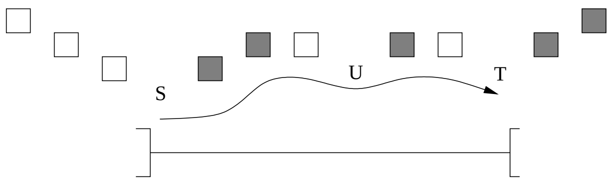

Figure 1: Interval for a loop-independent dependence (basic case).

dependence is represented by an open interval $]S, T[$ (see the bottom of Figure 1), and any barrier placed inside this interval enforces the dependence. 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

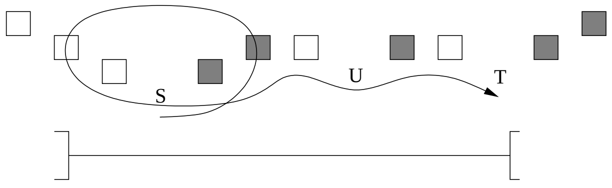

Figure 2: Case of an interval, for a loop-independent dependence, left-extended to a DO.

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

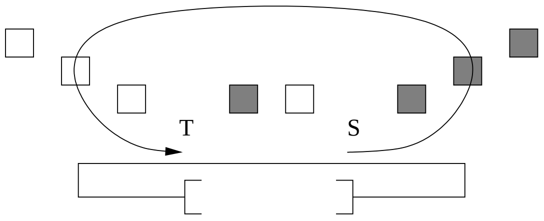

Figure 3: Circular interval for loop-carried dependence (basic case).

control points where a barrier needs to be inserted (and any such control point is fine) can be represented by a **circular 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 4th DO in the example) or an ENDDO after T (the ENDDO of the 3rd loop in the example).

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 a Placement Better?

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 (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 $(T_i)_{1 \le i \le n}$ and $(U_i)_{1 \le i \le n}$ 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, $1 \le i \le n$, and*
- *if, for each i, $1 \le i \le n$, $T_i = U_i$, then $b(t) \le b(u)$,*

If $T \preceq U$ and T and U are different weighted trees, we say that T is less than U (that we denote by $T \prec U$).

Now we can compare barrier placements P and Q: 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. We prove that they do in the lemma below. 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 correct placements P and Q, there is a correct placement as good or better than both P and Q. Consequently, optimal placements exist.*

PROOF. The proof (which is perfectly safe to skip) 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 placement 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 n loops $(L_i)_{1 \le i \le n}$, consider two placements 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 placements 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 placements 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 placements R_i to a placement 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 true even if we restrict to the finite set of placements that place in each loop at most as many barriers as statements plus one (i.e., one barrier between any two statements). Thus, given any two correct placements, there is a correct placement as good or better than each. This implies that there are optimal placements. □

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. Furthermore, 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 placement minimizes, among all correct placements, the number of barrier calls that occur at runtime.*

PROOF. (Again, the proof may be skipped.) It suffices to show that if Q, with tree cost U, is not optimal (in terms of $\preceq$), then there exists a better placement P, with tree cost $T \prec U$, such that P does not induce more dynamic barriers than Q.

Consider Q a placement 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 placement 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 placement for L'. Replace in Q the barriers in L' (i.e., in L' and deeper) by the barriers of any optimal placement for L'. This gives a partially correct placement: 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 placement 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. □

A 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 loop, $k = 1$) 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.

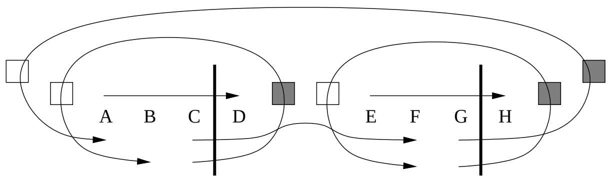

Figure 4: A 2D example and its (unique here) optimal placement.

3. THE HSU-TSAI ALGORITHM AND INNER LOOP BARRIER MINIMIZATION

In this section we summarize known results for a simple loop. First note that barrier minimization for a straight-line code is equivalent to the minimum clique cover for an interval graph. There is a greedy, linear-time algorithm that has been rediscovered in the parallel programming literature [5, 8]: 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. Next, these authors leverage this process to get a quadratic-time algorithm for simple loops modeled with circular intervals: 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. But, as we show next, there is already a linear-time algorithm for this problem, due to Hsu and Tsai [6], in the graph algorithms literature.

A **circular interval family** (CIF) 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; $\alpha(\mathcal{F})$ is 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 non-linear 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**; $\theta_l(\mathcal{F})$ is the minimum size of a 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.

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 [6]. We 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.

We assume, without loss of generality, that the endpoints are all different. Given an interval $I_i =]h_i, t_i[$, let us 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 1; 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 $I_{i_1}, \ldots, I_{i_k}$, with $i_1 = i$, and $i_t = \text{NEXT}(i_{t-1})$, $2 \leq t \leq k$, and they let LAST(i) = NEXT(i_k).

Theorem 1 (Hsu and Tsai [6]). *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 $\alpha(\mathcal{F}) > 1$, then 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 1 Barrier placement for an inner loop

```
Input: $\mathcal{F}$ is a set of $n \geq 1$ circular intervals $I_i = ]h_i, t_i[$, $1 \leq i \leq n$, such that $i \leq j \Rightarrow t_i \leq t_j$
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 = $ 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 = $ 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
        while MARK($i$) = 0 do
            MARK($i$) = 1; $i = $ NEXT($i$)
15:     end while                     ▷ until cycle is detected
        $j = i$
        repeat                         ▷ intervals in GD($i$)
            insert a barrier just before $t_j$; $j = $ NEXT($j$)
        until $I_i$ and $I_j$ overlap
20:     if $j \neq i$ then             ▷ special case for LAST($i$) $\neq i$
            insert a barrier just before $t_j$
        end if
    end procedure
```

Theorem 1 shows that for a circular interval family, $\theta_l(\mathcal{F})$ 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 1, Lines 1–11 compute the function NEXT for each interval, and the lines following 12 compute GD(i) and place the barriers accordingly. The fact that the tails are sorted 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. We prove, in the full paper [4], that the algorithm finds an MLQC in these cases as well.

4. OPTIMAL BARRIER PLACEMENT IN NESTED LOOPS WITH 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.

Algorithm 2 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 DO and ENDDO of $L'$
 4:         $\mathcal{D}' = \{d = (u, v) \in \mathcal{D} \mid u, v \in L', \text{level}(d) > \text{depth}(L')\}$
                                                        ▷ 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 barrier in $L'\}$ ▷ $\mathcal{D}' \subseteq$ CUT
 7:         $\mathcal{D} = \mathcal{D} \setminus$ CUT
 8:         for all $d = (u, v) \in \mathcal{D}$, $v \in L'$ do      ▷ dep. enters $L'$
 9:             $v = u_0$
10:         end for
11:         for all $d = (u, v) \in \mathcal{D}$, $u \in L'$ do      ▷ dep. leaves $L'$
12:             $u = v_0$
13:         end for
14:     end for
15:     HsuTsai($\mathcal{D}$)   ▷ or any optimal algorithm for single loop
16: end procedure
```

To place barriers in a loop L, Algorithm 2 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 9 and 12), because it must be cut by a barrier in L. Finally, the modified L is handled as an inner loop (Line 15).

Algorithm 2 yields an optimal placement if each loop has a unique optimal placement or if, by chance, it picks the right optimal one at each level. The problem is thus to modify Algorithm 2 to select judiciously, among the optimal placements for contained loops, those that cut (Line 6) incoming and outgoing dependences so that the number of barriers determined in L (Line 15) 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 level$(d) >$ 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.

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 leftmost incoming and the rightmost 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 3. 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 3 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 re-
    turn 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
              $u = v$             ▷ 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 indepen-
           dent 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 3) 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 the 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 arcs in $\mathcal{F}$ as were cut by the one barrier in $\mathcal{F}'$. This is the unraveling process.

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. 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. In the full paper [4], we fill in the gaps in the informal proof we have given here of this, our main result:

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

Consider again the example of Figure 4, which we reproduce in Figure 5 for convenience. The first innermost loop L_1

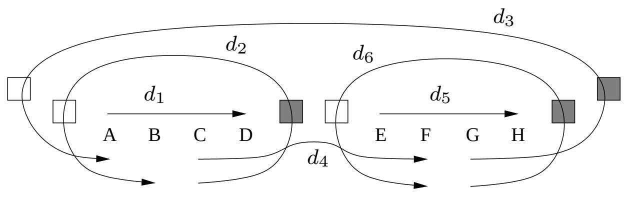

Figure 5: The 2D example of Figure 4.

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 6.

The function NEXT is: 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 as using the rightmost

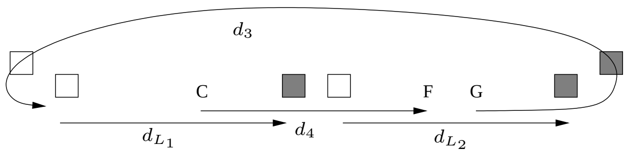

Figure 6: Woven CIF for the NCIF of Figure 5.

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.

Consider now an example of O'Boyle and Stöhr [8], Figure 7. Only one barrier is needed in the innermost loop L_1

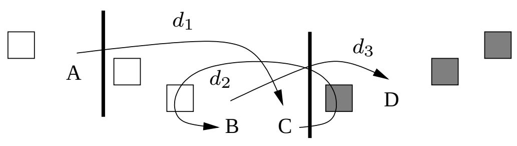

Figure 7: A 3D example from O'Boyle and Stöhr.

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 8. Now, the innermost loop L_2 has two

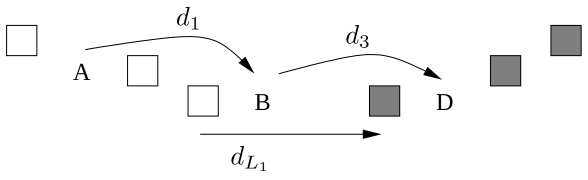

Figure 8: Woven NCIF for the NCIF of Figure 7.

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 9. 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 7, 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).

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 7, d_1 ends strictly after C, then it can be cut by an

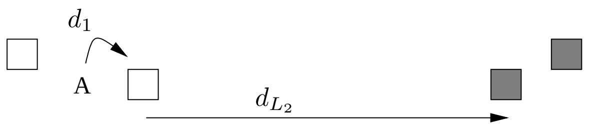

Figure 9: Woven CIF for the NCIF of Figure 8.

optimal placement for L_1 (with a barrier just before its tail) that cuts all dependences. In Figure 8, 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 9, 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

To get a linear-time algorithm for optimal barrier placement, it remains to compute the function RIGHTMOST in linear time. For that, we analyze more precisely the structure of rightmost placements in a CIF. *This section is included for completeness; it may be skipped at first reading. Computing RIGHTMOST is a complexity issue; the structure and correctness of the weaving/unraveling process are not affected. All details are given in the full paper [4]. Here we give only the highlights.*

We use the notations of Section 3. For each loop independent interval I_i, we define GDR(i) the maximal sequence $I_{i_1}, \dots, I_{i_n}$ of independent intervals such that $i_1 = i$, $i_t = \text{NEXT}(i_{t-1})$ for $2 \le t \le n$, and the tail of I_{i_t} is to the right of the tail of $I_{i_{t-1}}$: GDR(i) is similar to 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 GDR(i) are loop-independent. We define RIGHT(i) $= i_n$ and 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, using the relation RIGHT(i) = RIGHT(NEXT(i)) and LENGTH(i) = LENGTH(NEXT(i)) + 1.

Then, 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 $\mathcal{F}' = \mathcal{F} \cup \{I_i\}$. 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 $\mathcal{F}$. 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 Algorithm 1, provided that internal intervals and incoming intervals are sorted by increasing tail. 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 $\mathcal{F}$.

As a corollary of 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 maximal depth of a loop.

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.

While our method is theoretically superior to the simpler method of O'Boyle/Stöhr, in practice it may not yield a large reduction in dynamic barrier counts, as the number of barriers placed in innermost loops is the same. We do know, however, that it won't increase the number of barriers. Also, it is faster (linear complexity) than the algorithm of O'Boyle/Stöhr for nested loops.

More importantly, removing barriers may change the load balance characteristics of a program in such a way that an optimized program that makes fewer calls to barrier can nevertheless run slower. Consider the example of Figure 10, for a SPMD code with two threads (each thread is depicted horizontally, the corresponding statements from left to right, either as a rectangle for long task, or a square for a short task), and two interthread dependences. In Figure 10, the minimum number of barriers is 1. The execution time is the longest time (in grey in the figure) to reach the barrier (one square, one rectangle, for the first thread), the cost of a barrier, and again the maximum time to end the program (one rectangle, one square, for the second thread). However,

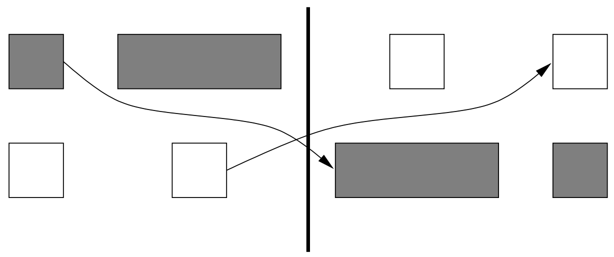

Figure 10: Total runtime = two rectangles + two squares + 1 barrier.

if we use two barriers instead of one as in Figure 11, the execution time is one square, one barrier, one rectangle and one square, one barrier, and one square, which may be less if the time for a square and a barrier is less than the time for a rectangle. In other words, in case of workload imbalance, minimizing the number of barriers may increase the execution time. It would be quite interesting to extend our model, including load balancing effects and solving for an optimum runtime, rather than minimum barrier count.

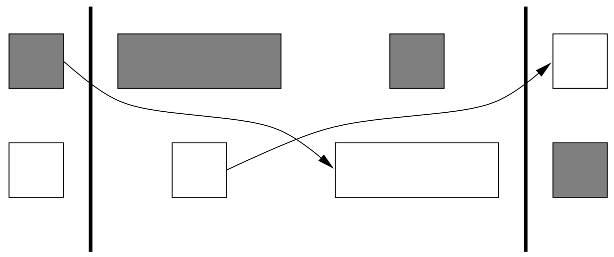

Figure 11: Total runtime = 1 rectangle + three squares + 2 barriers.

Other optimizations, 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 (see [9] for example).

Note too that we have made some important restrictions on the kind of program to be optimized.

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.

6. 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] A. Aiken and D. 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] Co-Array Fortran. `http://www.co-array.org/`.

[4] A. Darte and R. Schreiber. Nested circular intervals: A model for barrier placement in SPMD codes with nested loops. Technical Report RR2004-57, LIP, ENS-Lyon, Dec. 2004. `http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-57.pdf`.

[5] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, 1991.

[6] W.-L. Hsu and K.-H. Tsai. Linear time algorithms on circular-arc graphs. *Information Processing Letters*, 40(3):123–129, 1991.

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

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

[9] C.-W. 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.

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

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

[12] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, Sept-Nov 1998.