Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design

CONSTANTINE D. POLYCHRONOPOULOS, MEMBER, IEEE

Abstract—In this paper, we present two new compiler optimizations for parallelism detection and address their influence on architecture design. By examining the structure and characteristics of parallel programs in general, and the code resulting from these transformations in particular, we can isolate potential overhead sources, and propose architectural enhancements that reduce their impact on performance. The first compiler optimization we discuss is cycle shrinking which can be used to parallelize certain types of serial loops. Then we consider run-time dependence analysis and show how it can be performed through compiler-inserted bookkeeping and control statements. Loops with unstructured parallelism that cannot benefit from existing optimizations can be parallelized through run-time dependence checking. Finally, we discuss barrier synchronization, one of the most serious sources of run-time overhead in parallel programs. To reduce the impact of barriers, we briefly discuss the implementation of distributed barriers through the use of a set of shared registers.

Index Terms—Barrier synchronization, compilers, data dependences, hardware synchronization, loop transformations, parallel computers, parallel loops, run-time dependence testing.

I. INTRODUCTION

PARALLEL processor systems offer tremendous potential for reducing the execution time of large numeric-intensive applications. Parallel algorithms and programs can run faster on such systems by simultaneously executing on many processors. Experience with parallel machines, so far, has indicated that only a small fraction of the available parallelism in programs can be exploited on such machines. This is due to several types of overheads that severely restrict performance. Typical overhead activities during parallel execution include synchronization, communication, and scheduling activities, as well as extra random delays due to memory and network conflicts. These sources of overhead can be reduced significantly by both compiler optimizations and architectural enhancements.

In this paper, we discuss compiler optimizations for parallelism detection, and we address their influence on architecture design. By examining the structure and characteristics of parallel programs we can isolate potential overhead bottlenecks, and propose architectural enhancements to reduce their impact on performance. Automatic program restructuring is a subject that has been studied for many years by a few researchers, and which has attracted great attention in recent years [2]–[4], [8], [11], [17], [23], [24], [27], [33]. Many vectorizing compilers have been developed in the past ten years [14], [5], [13], [9]. However, restructuring for parallel machines is still a relatively new subject. Some of the parallelizing compilers under development are the University of Illinois Parafrase II, the Rice University PFC, and the IBM PTRAN. Such compilers will inevitably influence the design of future parallel machines.

Here we focus on two new techniques for transforming serial programs into a parallel form suitable for execution on parallel processor systems. These transformations can be implemented in a restructuring compiler with minimal effort, and for certain types of serial loops they can obtain significant improvements. More specifically, we discuss in detail cycle shrinking, a compiler transformation that can parallelize certain types of serial loops. Three versions of this transformation (based on different uses of dependence distance information) are presented and compared. Next we discuss run-time dependence analysis that can be performed through compiler-inserted bookkeeping and control statements. Loops with unstructured parallelism that cannot benefit from existing optimizations can be parallelized through run-time dependence checking. Subsequently, we consider barrier synchronization, the most serious and the most frequent source of overhead in parallel loops. To reduce the impact of barriers, we briefly discuss the implementation of distributed barriers through the use of a set of shared registers.

Specifically, Section II gives a short introduction to data dependences and defines the basic concepts. Section III discusses cycle shrinking in detail; Section III-A covers singly nested loops, while Section III-B discusses the application of cycle shrinking to complex nested loops. Run-time dependence checking and the RDC transformation are presented in Section IV. In Section V, we discuss the limitations of cycle shrinking and other transformations in the presence of barrier overheads, and consider a hardware solution. The conclusion is given in Section VI.

II. DATA DEPENDENCES

The material presented in this paper requires some knowledge of data dependences and some associated concepts. Before we give a short introduction to data dependences, however, we need to establish a basic notation.

A program is a list of \( k \in N \) statements. \( S_i \) denotes the \( i \)th
statement in a program (counting in lexicographic order). $I_j$ denotes a DO loop index, and $i_j$ denotes a particular value of $I_j$. $N_j$ is the upper bound of a loop index $I_j$, and all loops are assumed to be normalized, i.e., the values of an index $I_j$ range between 1 and $N_j$. The stride $r_j$ of a loop index $I_j$ is a positive constant by which $I_j$ is incremented each time it is updated. We have two types of statements, scalar and indexed statements.

An indexed statement is one that appears inside a loop, whose execution is explicitly or implicitly controlled by an index (e.g., vector statements). All other statements are scalar. The degree of a program statement is the number of distinct loops surrounding it, or the number of dimensions of its operands. $S(I_j, I_k, \cdots, I_m)$ denotes a statement of degree $k$, where $I_j$ is the index of the $j$th loop or dimension. An indexed statement $S(I_1, I_2, \cdots, I_k)$ has $N_1 \times N_2 \times \cdots \times N_k$ different instances, one for each value of each of $I_j$ ($j = 1, \cdots, k$). The first and last instance of a statement of degree $k$ is $S(1, 1, \cdots, 1)$ and $S(N_1, N_2, \cdots, N_k)$, respectively. $S_i$ will be used in place of $S(I_1, I_2, \cdots, I_k)$ whenever the set of indexes is obvious. The order of execution is defined as the pairs of statements instances $S_i, S_j$ as follows. For (scalar) statements with degree 0, we say that $S_i$ is executed before $S_j$ and denoted by $S_i \preceq S_j$ if $i \leq j$, i.e., if $S_i$ lexically precedes $S_j$. For $S_i, S_j$ with degree $k$ (if they have different degrees, then scalar order of execution applies as well), we have the following three cases. If $S_i$ and $S_j$ do not have common indexes, then $S_i \preceq S_j$ if $i \leq j$. If they have the same indexes, then $S_i(i_1, \cdots, i_k) \preceq S_j(j_1, \cdots, j_k)$ if $i \leq j$ and there is an $1 \leq m \leq k$ such that $i_l = j_l$, ($l = 1, \cdots, m$), and $i_{m+1} < j_{m+1}$. If $i > j$, it is as above but with $m < k$. If $S_i$ and $S_j$ have $m < k$ common indexes and $i_l = j_l$, ($l = 1, \cdots, m$), then $S_i \not\preceq S_j$ iff $i \leq j$. Otherwise, the previous definition holds.

We also need to extend the definition of the IN($S$) and OUT($S$) sets (i.e., the sets of variables read and written by statement $S$, respectively). We denote by OUT(S($i_1, i_2, \cdots, i_k$)) the set of variable instances (not necessarily different) defined by statement instance S($i_1, i_2, \cdots, i_k$). Similarly, IN(S($i_1, i_2, \cdots, i_k$)) is the set of variable instances used by the same statement instance.

Now we can proceed with the definition of data dependencies. Two statements $S(I_1, I_2, \cdots, I_k)$ and $S(J_1, J_2, \cdots, J_k)$ are involved in a flow dependence $S \delta S'$ if only if there exist index values ($i_1, i_2, \cdots, i_k$) and ($j_1, j_2, \cdots, j_k$), such that $S(i_1, i_2, \cdots, i_k) \preceq S(j_1, j_2, \cdots, j_k)$, and OUT(S($i_1, i_2, \cdots, i_k$)) AND IN(S($j_1, j_2, \cdots, j_k$)) not equal to empty set. An antidependence between $S_i$ and $S_j$ is defined as the flow dependence above, except for the last condition which now is IN(S($i_1, i_2, \cdots, i_k$)) AND OUT(S($j_1, j_2, \cdots, j_k$)) not equal to empty set. It is denoted by $S_i \delta S_j$. An output dependence is again defined as above but with OUT(S($i_1, i_2, \cdots, i_k$)) AND OUT(S($j_1, j_2, \cdots, j_k$)) not equal to empty set, and it is denoted by $S_i \delta S_j$. In all three cases, $S_i(i_1, i_2, \cdots, i_k)$ is called the dependence source and $S_j(j_1, j_2, \cdots, j_k)$ is called the dependence sink. Note that for a pair of statements $S_i$ and $S_j$, $S_i \delta S_j$ denotes the static dependence and implies that a dependence exists at least between a pair of instances of the two statements. Clearly each static dependence may have several different instances.

For each data dependence involving statements $S_i(i_1, i_2, \cdots, i_k)$ and $S_j(j_1, j_2, \cdots, j_k)$ of degree $k$, we define the rth distance $\phi_r$, or $\phi_r(\delta)$, to be $\phi_r = j_l - i_l$, (1 $\leq r \leq k$). The k-tuple ($\phi_1, \phi_2, \cdots, \phi_k$) is called the dependence distance vector. The true distance or simply distance $\Phi_{ij}$ is defined as

$$\Phi_{ij} = \sum_{r=1}^{k} \phi_r \prod_{m=r+1}^{k} N_m$$

and gives the total number of iterations between the source and the sink of a dependence.

The program data dependence graph, or DDG, is a directed graph $G(V, E)$ with a set of nodes $V = \{S_1, S_2, \cdots, S_n\}$ corresponding to statements in a program, and a set of arcs $E = \{e_{ij} = (S_i, S_j) | S_i, S_j \in V\}$ representing data dependences between statements. Fig. 1 shows the DDG of a set of scalar statements.

Several different transformations exist, that based on dependence information can restructure serial programs into vector or parallel form, suitable for execution on SIMD and MIMD systems. In a restructured program, parallelism is usually explicitly specified via parallel constructs. In this paper, we only consider parallel constructs used are various types of parallel loops. A DO loop denotes an ordinary Fortran loop, which is to be executed serially. A loop whose iterations can execute in parallel and in any order is called DOALL. The dependencies in certain loops may allow only partially overlapped execution of successive iterations. These loops are called DOACROSS and are mentioned in only a few cases in this paper [23], [11]. A loop can be marked as being DO, DOALL, or DOACROSS after the necessary dependence analysis for that loop has been carried out.

Loop parallelization, discovering and packaging unstructured parallelism, interprocedural analysis, compile-time overhead analysis, and many other complex tasks can be carried out automatically by a restructuring compiler [26], [25]. In many cases, user assertions can be useful or necessary. Automating the parallelization process makes parallel program writing possible for nonexperts, and in general, results in more efficient code.

We plan to implement the transformations presented in this paper in Parafuse II, an interactive multilingual restructurer which is currently under development at the University of Illinois. Parafuse II will encompass most of the existing technology in the area, as well as new methods that have been developed recently. The compiler is designed to allow easy integration of new technology with minimal effort. In the rest of this paper, we discuss specific methods without reference to implementation details.

III. CYCLE SHRINKING

Some of the most important transformations are those that vectorize and parallelize serial loops. These transformations use the data dependence graph of a loop to determine whether the existing dependences allow the loop, or parts of it, to execute in parallel without violating the semantics. If no dependences exist, this task is simple. In most cases, however, the dependence graph is complex, and appropriate tests are necessary to determine whether a pattern of dependences can allow vectorization or parallelization. Usually loops with
dependence graphs that do not form strongly connected components can become fully or partially parallel. When the dependence graph forms a cycle, node splitting can be used to break the cycle, assuming at least one of the dependences is an antidependence [14], [4].

In many serial loops, dependence cycles are impossible to break even after all known techniques are used. Such dependence cycles usually involve only flow dependences. In general, loops whose statements are involved in a dependence cycle are considered to be serial. Cycle shrinking can be used to extract any parallelism that may be (implicitly) present in a serial loop.

A. Simple Loops

Cycle shrinking is useful in cases where the dependence cycle of a loop involves dependences with distance greater than one. This scheme transforms a serial DO loop into two perfectly nested loops; an outermost serial and a parallel innermost loop. It is based on the observation that although there is a static flow dependence $S_i \delta S_j$ between two statements $S_i$ and $S_j$ of a loop, there may be instances of $S_i$ and $S_j$ that are not involved in a dependence (since the dependence distance is greater than one). Cycle shrinking extracts these dependence-free instances of the statements inside a loop, and creates an inner parallel loop.

We can best describe cycle shrinking by considering initially dependence cycles where all dependences have the same distance $\lambda > 1$. In such cases, cycle shrinking will speed up the loop by a factor of $\lambda$, called the reduction factor. First let us consider the case of singly nested serial loops with $n$ statements $S_1, S_2, \cdots, S_n$ that are involved in a dependence cycle such that $S_i \delta S_j \delta \cdots S_{i-\lambda} \delta S_1$. Furthermore, assume that all $n$ dependences have a common distance $\lambda > 1$. For constant distance dependences, the array subscript expressions of elements that are involved in a dependence must be of the form $I = a$, where $a \geq 0$ and $I$ is the loop index. Then (assuming loops are always normalized without loss in generality) we have the following.

Lemma 1: The iteration space $[1 \cdots N]$ of a loop whose dependences are of constant distance $\lambda$, can be partitioned into subintervals or groups $V_0 = [1 \cdots \lambda]$, $V_1 = [\lambda + 1 \cdots 2\lambda]$, $\cdots$, $V_r = [(\lambda + 1) \cdots N]$, so that the iterations of each group $V_j$ are free of dependence sinks, assuming that groups $V_1, V_2, \cdots, V_{r-1}, (j = 1, 2, \cdots, r)$ have been executed. (For simplicity, assume that the first dependence starts from the first iteration of the loop.)

Proof: First let us suppose that all dependence sources (for each instance of the static dependence cycle) belong to the same iteration of the loop. Then, since all dependences have the same distance $\lambda$, it follows that all dependence sinks also belong to the same iteration of the loop. Let $i_1$ and $i_2$ be two such iterations; $i_1$ involving the dependence sources of some instance of the dependence cycle, and $i_2$ involving the corresponding dependence sinks. Clearly such $i_1$ and $i_2$ should satisfy $i_2 - i_1 = \lambda$. Since no other dependences exist, it follows that all instances of statements in iterations $i_1, i_1 + 1, \cdots, i_1 + \lambda - 1$ contain no dependence sink and thus they are independent (assuming iterations $1, 2, \cdots, i_1 - 1$ have been executed). The statement of the lemma then follows directly.

Now let us consider the other extreme where the $n$ dependence sources in a serial loop (and their corresponding sinks) belong to different iterations of that loop. Without loss in generality, assume that the first instance of $S_1 \delta S_2$ corresponds to $I = 1$, $S_2 \delta S_1$ corresponds to $I = 2$, $\cdots$, $S_n \delta S_1$ corresponds to $I = n$, where $n \leq N$. The cases of $n = \lambda$ and $n > \lambda$ are identical. Let us consider $n > \lambda$.

From the above hypothesis and since all dependences are of constant distance $\lambda$, it is clear that the sink of $S_1 \delta S_2$ belongs to iteration $I = \lambda + 1$, the sink of $S_2 \delta S_1$ belongs to iteration $I = \lambda + 2$, $\cdots$, and the sink of $S_n \delta S_1$ belongs to iteration $I = n$. Then, iterations $I = 1, 2, \cdots, \lambda$, are free of dependence sinks and thus independent. Therefore, iterations $V_0 = [1 \cdots \lambda]$ can be executed in parallel satisfying the first instance of the $n$ dependences in the cycle. Then group $V_1 = [\lambda + 1 \cdots 2\lambda]$ can be constructed in the same way and so on. The general case is identical.

Lemma 1 indicates that serial loops with dependences that have equal (constant) distances can be partially parallelized. This can be done by transforming the serial loop into two perfectly nested loops. The innermost loop is a DOALL and at each time its index runs over the iterations of a group $V_j, (j = 1, 2, \cdots, r)$, where $r = [N/\lambda]$. The outer loop is a serial DO loop whose index runs across groups $V_j, (j = 1, 2, \cdots, r)$.

Consider for example the loop of Fig. 2(a). The dependence graph is shown next to it. Such a loop would be treated as serial by the existing compilers. However, if cycle shrinking is applied, the same loop will be transformed to that of Fig. 2(b). The transformed loop can now be executed $\lambda = K$ times faster. The larger the distance $\lambda$, the greater the speedup. Values of $\lambda = 2$ or 3 occur in numerical programs. Another example is shown in Fig. 3 where the loop has been unrolled to clearly show the effect of cycle shrinking. Two iterations at a time can be executed in parallel in this case.

Let us consider now the case where the distance of each dependence (in the static dependence cycle) is constant, but distances between different static dependences are different. In this case, we have a cycle $S_1 \delta S_2 \cdots S_i \delta S_j$ where $\phi_i$ is the distance of the $i$th dependence. Without loss in generality, assume that $\phi_1 \geq \phi_2 \geq \cdots \geq \phi_r$. Then we have the following.
DO I = 1, N
   s1: A(I*K) = B(I) - 1
   s2: B(I*K) = A(I) + C(I)
ENDO

(a)

DO I = 1, N, K
DOALL J = I, I+K-1
   A(J*K) = B(J) - 1
   B(J*K) = A(J) + C(J)
ENDO
ENDO

(b) Fig. 2. (a) Example of loop with constant distance dependence cycle. (b) The transformed loop of (a).

DO I = 3, N
   A(I) = B(I-2) - 1
   B(I) = A(I-3) * K
ENDO

(a)

DO J = 3, N, 2
DOALL I = J, J+1
   A(I) = B(I-2) - 1
   B(I) = A(I-3) * K
ENDOALL

(b) Fig. 3. (a) A serial loop. (b) The transformed loop. (c) The unrolled version of (b).

DO I = 4, N
   A(41-I) = M * B(3I-1)
   B(41-I) = C(I) + A(3I-1)
ENDO

(a)

DO J = 4, N, 2
DOALL I = J, J+1
   A(41-I) = M * B(3I-1)
   B(41-I) = C(I) + A(3I-1)
ENDOALL

(b) Fig. 4. Another application of cycle shrinking.

DO I = 4, N
   X(I) = Y(I) + Z(I)
   Y(I+3) = X(I-3) * W(I)
ENDO

(a)

DO J = 1, N, 3
DOALL I = J, J+2
   X(I) = Y(I) + Z(I)
   Y(I+3) = X(I-6) * W(I)
ENDOALL
ENDO

(b) Fig. 5. Cycle shrinking with varying dependence distances.

I = 3: A(3) = B(1) - 1
   B(3) = A(0) * K
I = 4: A(4) = B(2) - 1
   B(4) = A(1) * K
I = 5: A(5) = B(3) - 1
   B(5) = A(2) * K
I = 6: A(6) = B(4) - 1
   B(6) = A(3) * K
I = 7: A(7) = B(5) - 1
   B(7) = A(4) * K
I = 8: A(8) = B(6) - 1
   B(8) = A(5) * K
I = 9: A(9) = B(7) - 1
   B(9) = A(6) * K
I =10: A(10) = B(8) - 1
   B(10) = A(7) * K
I =11: A(11) = B(9) - 1
   B(11) = A(8) * K
I =12: A(12) = B(10) - 1
   B(12) = A(9) * K

(c)

Corollary 1: If \( \lambda = \min (\phi_1, \phi_2, \cdots, \phi_n) = \phi_1 \), then Lemma 1 holds true.

Proof: The proof follows directly from Lemma 1. For example, if \( I = i_1 \) is the iteration that contains the source of the static dependence \( S_i \delta_1 S_2 \), then iterations \( I = i_1, i_1 + 1, \cdots, i_1 + \phi_1 - 1 (= i_1 + \lambda - 1) \) contain no dependence sink and are thus independent. By definition then, it follows that iterations \( I = i_1 + \lambda, i_1 + \lambda + 1, \cdots, i_1 + 2\lambda - 1 \) contain sinks whose corresponding sources have been executed. The proof is identical for the other cases.

An example of a serial loop whose dependence cycle involves dependences with different distances is shown in Fig. 4(a). In this case, \( \phi_1 = 4 \) and \( \phi_2 = 3 \). According to Corollary 1, the cycle can be reduced by a factor of \( \lambda = \min (3, 4) = 3 \) resulting in the loop of Fig. 4(b).

The next case to be examined is when the distance of each static dependence is different for different instances of the dependence. In singly nested loops, this happens when we have array subscripts of the form \( a \leq b \) where \( a \geq 1 \) or \( a \leq -1 \). An example of such case is shown in Fig. 5(a). Here the distance of each static flow dependence varies between different instances of the dependence. The minimum distance between all instances of both dependences in the example is \( \lambda = 2 \). (Actually distances are monotonically increasing between successive iterations.) Corollary 1 can be applied in this case to transform the loop as in Fig. 5(b). Even though cycle shrinking extracts some parallelism from this loop, it is still a rather conservative solution in the sense that it leaves some parallelism unexploited. A more general and powerful approach will be discussed in Section IV. Lemma 1 and Corollary 1 can be summarized in the following.

Corollary 2: A sufficient condition for a loop dependence cycle \( S_i \delta_1 \delta_2 \cdots \delta_n \) to be reduced by a factor of \( \lambda \) is

\[
\lambda = \min_{i \leq k} \{ \phi(\delta_i) \}.
\]  

(2)

Allen and Kennedy stated Corollary 2 in [4] where they use it in a similar transformation for blocking and vectorizing loops.
Theorem 1: Consider a DO loop with \( k \) statements which are always involved in a dependence cycle. If the reduction factor of the cycle is \( \lambda \), then cycle shrinking increases the total parallelism in the loop by a factor of \( \lambda + k \).

Proof: Since cycle shrinking creates \( \lambda \) independent iterations, the parallelism is increased by a factor of \( \lambda \). Furthermore, all statement instances (in a group with \( \lambda \) independent iterations) are also independent. This is true because all \( k \) statements are involved in the cycle, and according to Lemma 1, each \( \lambda \)-iteration group is free of dependence sinks. Thus, within each group all \( \lambda + k \) statement instances can execute in parallel.

B. Complex Loops

In this section, we discuss cycle shrinking for complex nested loops. There are three versions of cycle shrinking that can be used for nested loops. They differ in the way individual and true distances are used to compute the reduction factor, and they are discussed below.

Simple Shrinking: In this version, the dependence cycle is considered separately for each individual loop in the nest. For a loop at nest-depth \( l \), only the \( l \)th elements of the distance vector are considered. For each loop in the nest, cycle shrinking is applied separately as in the single loop case.

True Dependence Shrinking (or TD Shrinking): In this version, only true distances are used. Each dependence in the dependence cycle is labeled by its true distance computed by (1). Cycle shrinking is then applied as if the nested loop was a single loop. In this case, a multidimensional iteration space is treated as a linear space. It will be shown later that, as a result of this, loops are blocked by cycle shrinking in an “irregular” fashion.

Selective Shrinking: Here we consider each component of the distance vectors separately as in the case of simple cycle shrinking. In a loop nest of depth \( k \), we thus have \( k \) different dependence cycles, one for each individual loop. Each dependence in a cycle is labeled with the corresponding element of its distance vector. Next, selective shrinking computes the reduction factor \( \lambda_i \) (\( i = 1, 2, \ldots, k \)) [using (2)] for each loop in the nest starting with the outermost loop. The process stops when for some \( 1 \leq j \leq k, \lambda_j \geq 1 \). Then the \( j \)th loop in the nest is blocked by a factor of \( \lambda_j \). In addition, all loops nested inside the \( j \)th loop are transformed to DOALL’s.

One can make a number of interesting observations at this point.

1) The true distances depend on the loop size. This is not true with the individual distances. Thus, in general, the larger the loop sizes the longer the true distances, and therefore the more the potential parallelism.

2) An implication of the previous observation is that more parallelism can potentially be exploited by blocking the loops in an “irregular” fashion.

3) In general, the loop upper bounds must be known in order to compute and compare the true distances. The values of the distances per se are not needed, but determining the minimum true distance in a cycle is essential for cycle shrinking to work in this mode.

It is also clear that, by definition, selective shrinking is always better than simple shrinking. In addition, we have the following theorem.

Theorem 2: Let \( S_1 \delta_1 S_2 \delta_2 \cdots S_k \delta_k S_1 \) be a dependence cycle with \( k \) statements that are nested inside \( m \) loops of sizes \( N_1, N_2, \ldots, N_m \), respectively (counting from the outermost to the innermost loop). Let also \( \phi_i \) and \( \phi_t \) be the distance vector and the true distance vector of \( \delta_i \), \( i = 1, \ldots, k \), respectively. The reduction factor obtained by TD shrinking is greater than or equal to the total reduction factor obtained by simple shrinking. Or, equivalently,

\[
\min \left( \Phi_1, \cdots, \Phi_k \right) \geq \prod_{i=1}^{m} \min \left( \phi_i, \phi_t \right) = \Phi_t.
\]

Proof: Without loss in generality, assume that \( \Phi_1 = \Phi_2 = \cdots = \Phi_k \) and let \( \phi_t = \min \left( \phi_1, \phi_t, \cdots, \phi_k \right) \), \( i = 1, 2, \cdots, m \). By definition we have \( \Phi_1 = \sum_{i=1}^{m} \phi_i \Pi_{j=i+1}^{m} N_j \), and

\[
N_i \geq \phi_i \phi_t \phi_t' \phi_t'' \cdots \phi_t'^{m-1} N_m.
\]

From (3) we have \( \Pi_{i=2}^{m} N_i \geq \Pi_{i=2}^{m} \phi_t' \phi_t'' \cdots \phi_t'^{m-1} \), and thus,

\[
\phi_t \prod_{i=2}^{m} N_i \geq \phi_t' \prod_{i=2}^{m} \phi_t'' \phi_t'' \cdots \phi_t'^{m-1}
\]

and from (4) it follows directly that

\[
\Phi_1 = \sum_{i=1}^{m} \phi_t \prod_{i=2}^{m} N_i \geq \prod_{i=1}^{m} \phi_t \prod_{i=1}^{m} \min \left( \phi_t, \phi_t', \cdots, \phi_t'' \right)
\]

which proves the theorem.

It remains to compare TD shrinking to selective shrinking. Consider the true distance \( \Phi \) of a dependence as defined by (1), and let \( \mu_i = \phi_i \Pi_{j=i+1}^{k} N_j \). Then we have the following.

Lemma 2: If \( \Phi \) is positive, then

\[
\mu_i = \sum_{j=i+1}^{k} \mu_j, \quad (i = 1, 2, \cdots, k-1).
\]

Proof: For simplicity and without loss in generality, let us assume that loops are normalized. By definition we have

\[
N_{i+1} - 1 \geq \phi_{i+1}, \quad (i = 0, 1, 2, \cdots, k-r).
\]

Relation (5) can be rewritten as

\[
\phi_i \prod_{i=r+1}^{k} N_i \geq \phi_{i+1} \prod_{i=r+2}^{k} N_i + \phi_{i+2} \prod_{i=r+3}^{k} N_i + \cdots + \phi_{k-1} N_k + \phi_k.
\]

By using (6) it is enough to show that

\[
\prod_{i=r+1}^{k} N_i \geq (N_{r+1} - 1) \prod_{i=r+2}^{k} N_i + (N_{r+2} - 1) \prod_{i=r+3}^{k} N_i + \cdots + (N_{k-1} - 1) N_k + N_k - 1
\]
DO I = 3, N1
  DO J = 5, N2
      A(I, J) = B(I-3, J-5)
      B(I, J) = A(I-2, J-4)
  ENDDO
ENDO

(a)

DO I = 3, N1, 2
  DO J = 5, N2, 4
      DOALL I = I1, I1 + 1
        DO ALL J = J1, J1 + 3
            A(I, J) = B(I-3, J-5)
            B(I, J) = A(I-2, J-4)
        ENDDO
      ENDDO
  ENDDO
ENDO

(b)

DO I = 3, N1, 2
  DO J = 5, N2
      DOALL I = I1, I1 + 1
        DO ALL J = J1, J1 + 3
            A(I, J) = B(I-3, J-5)
            B(I, J) = A(I-2, J-4)
        ENDDO
      ENDDO
  ENDDO
ENDO

(c)

Fig. 6. Simple and selective shrinking for multiply nested loops.

or

\[
\prod_{i=r+1}^{k} N_i \geq \prod_{i=r+1}^{k} N_i - \sum_{i=r+2}^{k} N_i - \cdots - N_k + N_{k+1} - 1
\]

and by cancelling out the corresponding terms we have \(0 \geq -1\). Thus, (5) is always true.

Based on Lemma 2, we can now derive the condition under which TD shrinking is more effective than selective shrinking. This condition is stated in the following theorem.

**Theorem 3:** Let \(\Phi_m = \sum_{k=1}^{N} \phi_k \Pi_{i=1}^{k} N_i\) be the minimum true distance in a dependence cycle of a nested loop of nest-depth \(k\). Let also \(\lambda_T\) and \(\lambda_S\) be the reduction factors obtained for this loop by TD shrinking and selective shrinking, respectively. Then we have the following:

1. If there exist \(r_1, r_2\) with \(1 \leq r_1 < r_2 \leq k\) such that \(\phi_{r_1} > 0, \phi_{r_2} < 0,\) and \(\phi_j = 0, (j = r_1 + 1, r_1 + 2, \cdots, r_2 - 1),\) then \(\lambda_T < \lambda_S\).
2. In all other cases \(\lambda_T \geq \lambda_S\).

**Proof:** a) If such \(r_1\) and \(r_2\) exist, then from Lemma 1 it follows immediately that

\[
\lambda_T = \sum_{j=1}^{k} \phi_j \prod_{i=r_1}^{k} N_i \leq \phi_{r_1} \prod_{i=r_1}^{k} N_i + \sum_{j=r_2}^{k} \phi_j \prod_{i=j+1}^{k} N_i
\]

\[
\leq \phi_{r_1} \prod_{i=r_1+1}^{k} N_i \leq \lambda_S.
\]

b) Again using Lemma 1, this statement is proved in the same way.

Thus, TD shrinking and selective shrinking are always superior to simple shrinking. Theorem 3 gives us the condition for selecting between the other two schemes, depending on the distance vector.

Let us consider the application of the three versions of cycle shrinking to the loop of Fig. 6(a). Here the two statements are involved in a dependence cycle \(S_1 \delta S_2\) and \(S_2 \delta S_1\). The distance vectors for the two dependences are \((\phi_1, \phi_4) = (2, 4),\) and \((\phi_1, \phi_2) = (3, 5).\)

Simple shrinking considers the dependence graph with respect to indexes \(I\) and \(J\) individually. For the loop of Fig. 6(a), both graphs are cycles. Cycle shrinking can then be applied on each cycle separately. For the cycle corresponding to the outer loop, the distances are 2 and 3. Cycle shrinking will shrink the outer loop by a factor of \(2 = \min(2, 3).\) Similarly the inner loop will be shrunk by a factor of \(4 = \min(4, 5).\) The resulting loop is shown in Fig. 6(b). The transformed loop can execute eight iterations at a time in parallel, resulting in a speedup of 8.

The loop of Fig. 6(a) will be transformed to that of Fig. 6(c) if selective shrinking is used. Since the outermost loop has a reduction factor of 2, it is blocked as in Fig. 6(c) and the inner loop becomes a DOALL. In this case, \(\lambda = 2N_2 - 8.\)

Now let us compute the true distance for each dependence in the cycle and compare it to the individual distances. The true distances give us the total number of loop iterations (irrespective of which loop) over which a dependence travels. Here we have

\[
\Phi_{12} = \phi_1^1 (N_2 - 4) + \phi_1^2
\]

\[
\Phi_{21} = \phi_2^1 (N_2 - 4) + \phi_2^2
\]

In the case of our example, the value of \(N_2\) need not be known. By symbolic comparison of \(\Phi_{12}\) and \(\Phi_{21}\), it is easy to determine that

\[
\Phi_{12} = \min(\Phi_{12}, \Phi_{21}) = 2N_2 - 4.
\]

Thus, the reduction factor \(\lambda = 2N_2 - 4\) gives us the total number of loop iterations that can be executed in parallel during each step. For this example, TD shrinking works better than selective shrinking for any integer \(N_2\).

Let us see now how the loop of Fig. 6(a) can be transformed under TD shrinking, assuming that the reduction factor \(\lambda\) is known (which is the case for our example). Since the \(\Pi_{i=1}^{k} N_i/X\) different groups must be executed in the order implied by the original loop. That is, if a group runs up to \((I = i, J = j)\), the next group must start from \((I = i, J = j + 1)\) or \((I = i + 1, J = 1).\) For our example, let \(N = N_1 - 3 + 1 = M = N_2 - 5 + 1.\)

The transformed loop is shown in Fig. 7. The temporary variables \(T1\) through \(T4\) are used to mark the coordinates of the initial and the final points in the iteration space. The iteration space has a total of \(NM\) iterations. The transformed loop will sweep the iteration space by executing \(\lambda\) iterations in
parallel at a time. \([NM/\lambda]\) barrier synchronizations are enforced by the outer loop which sees the iteration space as a linear. If the index expressions in the loop body are to be preserved by cycle shrinking, the value of the outer serial loop must be decomposed into two coordinates, an \(I\) and a \(J\) coordinate. Alternatively, we could linearize the corresponding arrays (i.e., \(A\) and \(B\)) in the loop body, and collapse the two DOALL's into a single DOALL with \(\lambda\) iterations. This would eliminate the extra statements inserted by TD shrinking. Linearizing arrays is straightforward and we will not discuss it further [1].

The most serious overhead may arise from the two conditional statements that were inserted between the two DOALL's in Fig. 7. These conditionals set the initial and the final value of the \(J\) coordinate. They can be easily eliminated by unrolling the first and last iterations of the outer DOALL. The resulting loop is shown in Fig. 8. Note that the three DOALL's in Fig. 8 are completely independent. Thus, these three DOALL's can execute concurrently, in addition to executing each of them in parallel. The case of nonconstant distances is treated similarly.

The next case is nonperfectly nested loops. This case, however, is similar to the perfectly nested loop case, since only the common outer loop(s) is considered. The individual loops at the same nest level in such a nonperfectly nested loop can be processed separately. Consider, for example, the loop of Fig. 9(a). Cycle shrinking can be applied to the two inner loops separately if necessary (not in this case, of course, since both loops are DOALL's). The outer loop is serial because the two DOALL's are involved in a flow dependence cycle. Distances for the two dependences need only be computed with respect to index \(I\). In this case, the distances are 2 and 3, and thus \(\lambda = 2\). After cycle shrinking, the loop is transformed to that of Fig. 9(b).

Notice that according to Theorem 1 the improvement is not just a factor of \(\lambda = 2\). In addition to executing every two iterations of the outer loop in parallel, the two innermost DOALL's can also execute in an overlapped fashion since they are completely independent. Therefore, the available parallelism is increased by a factor of 4, assuming the inner loops are of equal size.

C. Computing Dependence Distances

The effectiveness of cycle shrinking depends on how accurately we can compute dependence distances. In particular, what is of interest to us is how accurately we can compute the minimum distance for each static dependence in a cycle. This becomes more difficult as the complexity of array index expressions increases. Fortunately, in real programs array index expressions are usually simple expressions like the ones used so far. A common index expression is of the form \(I \pm a\), where \(a\) is an integer constant. For such simple cases, the difference of the index expressions can actually give us the distance of the corresponding dependence (if it exists). For example, the distance of a hypothetical dependence between two array elements \(A(I \pm a)\) and \(A(I \pm b)\) is \(|a \pm b|\). The distance of all instances of such a dependence is obviously constant. Other fast heuristic schemes for computing the minimum dependence distance can be applied to certain special cases.

In the forthcoming example, dependence distances can be computed (and cycle shrinking can be applied) even when no information on data dependences is available. Consider the following loop.

\[
\text{DO } I = 1, N \\
A(I) = A(2I) \\
\cdots \\
A(3I + 1) = \cdots \\
A(2I - 4) \text{ ENDO.}
\]

Suppose that there is a dependence from the first to the second assignment statement. We can define the increment factor \(\rho\) of the second index expression \(2I - 4\) to be the value by which
DO I = 3, N
DOALL J = 1, N  
  ...
  A(I, J) = B(I-2, J)  
  ...
ENDOALL
DO ALL K = 1, N  
  ...
  B(I, J) = A(I-2, J-4)  
  ...
ENDOALL
ENDO

(a)

DO L = 1, N - 2
DOALL I = L + 1, N  
  ...
  A(I, J) = B(I-2, J)  
  ...
ENDOALL
DO ALL K = 1, N  
  ...
  B(I, K) = A(I-3, K)  
  ...
ENDOALL
ENDO
ENDOALL

(b)

Fig. 9. Cycle shrinking for nonperfectly nested loops.

the value of that expression is incremented for successive values of I. The increment factor of 2I - 4 is obviously ρ = 2. It is easy to observe that for each I, a flow dependence may develop if 3I + 1 ≥ 2I - 4. Let us define the difference D = 3I + 1 - 2I + 4 = I + 5. It can be seen now that for each value of I, the element A(3I + 1) defined by that iteration cannot be consumed before

\[
\begin{bmatrix}
D \\
\rho
\end{bmatrix} = \begin{bmatrix}
I + 5 \\
2
\end{bmatrix}
\]

iterations later. This indicates that the previous loop can be transformed to the following set of DOALL's.

\[
J = 1
\]

L1:  
INC = MIN (N - J, CEIL ((J + 5)/2) - 1)  
DOALL I = J, J + INC  
  ...
  A(3I + 1) = ...  
  ...
  = A(2I - 4)  
ENDOALL
J = J + INC + 1
IF J < N GOTO L1.

No dependence is violated by the transformed loop. Of course, this particular loop could be transformed directly into a single DOALL if appropriate synchronization was used. Whether synchronization could make the loop execute more efficiently than the one shown above depends on several factors such as size of the loop, efficiency of synchronization instructions of a particular machine, the size of D and ρ, etc. We return to this issue in Section IV.

In general, however, distance computation must be done as part of the data dependence analysis phase. This is so because only in this case we can accurately compute distances (at a higher cost), or we can check whether distances are computable at all. A detailed presentation on dependence analysis is given in [8], [33], and [4]. Let us outline here the basic steps for single loops and linear index expressions of the form ai ± bj. In particular, let us answer the existence question of a flow dependence between the two statements of the following loop.

DO I = 1, N  
  A(ai + b) = ...  
  ... = A(cI + d)  
ENDO.

For a flow dependence to exist, we must have values i and j of index I such that 1 ≤ i ≤ j ≤ N and ai + b = cj + d or ai - cj = d - b. This diophantine equation has a solution iff gcd (a, c) (i.e., the greatest common divisor of a, c), divides d - b. If (i0, j0) is any solution, then all solutions (i, j) of the equation are given by

\[
i = i_0 + \frac{tc}{\text{gcd} (a, c)}
\]

\[
= j_0 + \frac{ta}{\text{gcd} (a, c)}
\]

Acceptable solutions are those for which 1 ≤ i ≤ j ≤ N. In [8], it is shown how to compute the set S = {(i, j) | (i, j) a solution and 1 ≤ i ≤ j ≤ N}. From the set S, we can then obtain the distance vector as (j - i) | (i, j) ∈ S). The procedure for multiply nested loops and index expressions with more than one variable is similar. For nonlinear index expressions (that rarely occur in real programs) there is no known algorithm that solves the general problem efficiently.

In Section IV, we examine a run-time data dependence testing scheme that can be applied to complex loops for which cycle shrinking is not very effective.

D. Cycle Shrinking Versus Partition

In its purpose, cycle shrinking is similar to partial loop partition, another scheme for extracting parallelism from seemingly serial loops, which is described in [23]. In this section, we briefly outline the partition method, and then we prove that cycle shrinking is always superior to partition (at least theoretically). Some examples that compare the effectiveness of the two schemes are also given.

Consider the following DO loop.

DO I = 1, N  
  Si  
  Sj  
  ...  
  Sk  
ENDO

with k statements that are involved in a dependence cycle S1δ1S2⋯δk−1SkδkS1, and let φi be the distance of δi, (i = 1,
Partition transforms the above loop to the following equivalent.

\[
\text{DOALL } J = 1, g \\
\text{DO } I = J, J + [(N - J)/g] \\
S_1 \\
S_2 \\
\cdots \\
S_k \\
\text{ENDOALL} \\
\text{ENDO}
\]

where \( g = \gcd(\phi_1, \phi_2, \cdots, \phi_k) \) is the greatest common divisor of all \( k \) distances in the dependence cycle. For the sake of completeness, the same loop will be transformed by cycle shrinking to the following one.

\[
\text{DO } J = 1, N, \lambda \\
\text{DOALL } I = J, J + \lambda - 1 \\
S_1 \\
S_2 \\
\cdots \\
S_\lambda \\
\text{ENDOALL} \\
\text{ENDO}
\]

where \( \lambda = \min(\phi_1, \phi_2, \cdots, \phi_k) \). The following well-known lemma gives us the first comparison of the two schemes.

**Lemma 3:** If \( \phi_1, \phi_2, \cdots, \phi_k \) are positive integers, then

\[
\min(\phi_1, \phi_2, \cdots, \phi_k) \geq \gcd(\phi_1, \phi_2, \cdots, \phi_k) \text{ or } \lambda \geq g.
\]

Thus, the size of the DOALL created by cycle shrinking is always greater than or equal to the size of the DOALL created by partition. Partition tries to group together all iterations of a DO loop that form a dependence chain. Each such group is executed serially, while different groups can execute in parallel. Dependencies are confined within the iterations of each group and dependences across groups do not exist. In contrast, cycle shrinking groups together independent iterations and executes them in parallel. Dependencies exist only across groups and are satisfied by executing the different groups in their natural order.

The second advantage of cycle shrinking over partition stems from the above difference and from Theorem 1. Since all iterations of a group formed by cycle shrinking are dependence sink-free (in addition to executing them in parallel) all statements inside each iteration can also execute in parallel. For a loop with \( k \) statements, this gives us another speedup factor of \( k \). This is not true, however, for partition, since all iterations within each group (and thus all statement instances) form a dependence chain. Thus, the speedup obtained by cycle shrinking for a loop with \( k \) statements is \( S_{\text{sh}} = \lambda + k \), while the speedup due to partition for the same loop would be \( S_{\text{par}} = g \), where \( \lambda \geq g \) and \( k \geq 1 \).

Let us consider a particular example. The following loop has \( \lambda = 2 \) and \( g = 1 \). Thus, partition is unable to discover any parallelism. Cycle shrinking will transform this loop to the following equivalent.

\[
\text{DO } J = 3, N, 2 \\
\text{DOALL } I = J, J + 1 \\
A(I) = B(I - 2) \\
B(I) = A(I - 3) \\
\text{ENDOALL} \\
\text{ENDO}
\]

In addition to executing every two iterations in parallel, the two statements inside the DOALL are also independent. We thus have a total speedup of 4. Clearly selective and TD shrinking are always better than both simple shrinking and partition. In cases where both latter schemes fail, TD shrinking can still improve parallelism. Consider the following loop.

\[
\text{DO } I = 1, N \\
\text{DO } J = 1, N \\
A(I, J) = A(I - 1, J - 1) + 1 \\
\text{ENDO} \\
\text{ENDO}
\]

Cycle shrinking and partition will fail to transform the above loop. TD shrinking, however, will transform the loop to a form similar to that of Fig. 7 or Fig. 8, by forming a DOALL with \( N + 1 \) iterations.

**E. The Cost of Barriers in Cycle Shrinking**

For an "ideal" parallel machine (one that incurs zero overhead) shrinking is always preferable over partition. In reality, however, overhead is an important factor. The disadvantage of cycle shrinking is that it needs \( [N/\lambda] \) barrier synchronizations as opposed to only 1 required by partition. When \( \lambda \) is very small and the loop contains very few statements, the overhead involved may outweigh the benefit of the resulting parallelism.

To avoid applying cycle shrinking in such cases, the compiler can perform approximate tests to evaluate the potential gain in performance. These tests are described below for simple loops with \( N \) iterations; their generalization to multiply nested loops is straightforward. Let \( \lambda, g, \) and \( k \) be as defined above. Moreover let \( \beta, T_s, T_p, \) and \( T_{s_p} \) be the execution time of a simple loop iteration, the serial execution time of a loop, and the execution time of the transformed loops after shrinking and partitioning, respectively. In general, the overhead associated with barrier synchronization in multiprocessor systems is not constant. However, for our purpose we can assume a worst case overhead of \( \gamma \). For simplicity, let us also assume that our base unit is the execution time of a program "statement." This gives us \( \beta = k \).

For serial loops of the type discussed so far, the compiler must compute \( \lambda \) and \( g \) and make the appropriate selection between cycle shrinking, partitioning, or none of the above (in which case the loop remains serial). Two cases are of interest:

- **Case 1:** \( \lambda > g = 1 \) and \( \lambda k > g > 1 \).
- **Case 2:** \( \lambda > g = 1 \). In this case, cycle shrinking is the only alternative for parallelizing the loop. The question that must be
answered at this point is whether the overhead due to barrier synchronization can potentially outweigh the parallelism introduced by shrinking. The loop can be transformed if

$$T_1 = N\beta > T_2 = \frac{N\beta}{\lambda k} + \frac{N\gamma}{\lambda}.$$  

If we use $\beta = k$ and simplify the inequality, we get

$$\lambda k > \gamma + 1.$$  

(7)

The worst case barrier overhead $\gamma$ (which is architecture and machine dependent), can be supplied by the compiler. $\lambda$ and $k$ are computed as part of the transformation. Since this is only a rough test, it is conceivable that even though (7) may hold true, in reality $T_1 < T_2$. However, $\gamma$ can be chosen conservatively so that the error margin can be negligible.

**Case 2:** $\lambda k > g > 1$. In this case, a performance improvement of at least $g$ is secured. The question is whether shrinking is potentially better than partitioning. Again an approximate test can be formulated as follows. If

$$T_2 = \frac{N\beta}{g} > T_1 = \frac{N\beta}{\lambda k} + \frac{N\gamma}{\lambda}$$

then shrinking is preferred, otherwise partitioning is chosen. After simplification we arrive at

$$\lambda k > g(\gamma + 1).$$  

(8)

Tests (7) and (8) do not always guarantee correct selection between the two methods, but they do provide the compiler with a simple test for avoiding gross errors. In Section V, we take up again the issue of barrier synchronization and give a brief description of a hardware scheme that we have proposed and are currently investigating.

**IV. Run-Time Dependence Testing**

Most techniques that have been developed so far to analyze array subscripts and determine loop dependences solve this problem at compile time. This, of course, is desirable because there is no run-time overhead. Another alternative would be to determine data dependences dynamically at run time. In this section, we consider the problem of run-time dependence checking or RDC, and propose a hybrid solution.

Resolving data dependences at run time is certainly not a new idea. One could say that this is the main purpose behind the concept of data flow [7]. Tomasulo in [30] introduced the tagged-token scheme that allowed dynamic detection of dependences by the hardware. Other hardware mechanisms were discussed in [32] and [31]. All these schemes make little or no use of compile-time information to aid the solution of the problem. Nicolau in [21] discusses run-time disambiguation (RTD), a compiler solution to some restricted instances of the problem. RTD is rather limited in scope and specifically suited for trace scheduling. The main idea behind RTD is to have the compiler introduce assertions about the relationship of different program variables (typically loop indexes). These assertions are associated with probability estimates that need to be supplied by the user or the compiler.

The RDC transformation proposed here is a new scheme that relies on precise information obtainable at compile or run time, as opposed to relying on probabilistic assertions. Also, RDC targets dependences different than those resolved by RTD. RDC is important for many reasons. The first and most obvious reason is that in many loops there exist array subscripts that are not amenable to compile-time analysis. In such cases, subscript expressions are complex integer functions or even unknown functions about which the compiler cannot draw any conclusions. Another reason (which we consider even more important) for using RDC is the following. Even when the compiler can accurately determine loop dependences, not all instances of the statements may be involved in a dependence, or the distances of the same static dependence may vary between its different instances. This means that even when static dependences indicate that a particular loop is serial, we may still have several iterations of that loop that are dependence free and which could execute in parallel. RDC detects such cases and exploits all parallelism in loops that would otherwise be considered serial. Only iterations that are involved in a dependence are executed in a restricted manner in order as to satisfy these dependences.

The main idea behind RDC is to exploit the information about the problem available to the compiler, and based on that information, have the compiler generate code that resolves dependences (not amenable to compile-time analysis) at run time. Even though the principle is the same for different types of array subscripts, the mechanisms of the transformation vary for different cases. A transformation of RDC of array variables with subscripted subscripts is discussed in detail in [27] and [29]. In the next section, we consider RDC for linear subscript expressions. In the first part of this section, we consider RDC for singly nested loops that contain array subscripts which cannot be analyzed with the existing compile-time techniques, or serial loops with varying dependence distances which can indeed have some parallelism. RDC can be extended for multiply nested loops following the same procedure.

**A. The RDC Transformation for Nonsubscripted Subscripts**

Cycle shrinking can be used to partially parallelize serial loops when the dependence cycles involve dependences with constant distances. When the dependence distances vary between different iterations, cycle shrinking can still be applied as it was shown in Section III-B. In such cases, however, shrinking is rather conservative. RDC is a more suitable technique since it sequentializes only those iterations that are involved in a true dependence. All remaining iterations can execute in parallel.

Consider, for example, the loop of Fig. 10. The distance of the flow dependence $S_i \delta S_j$ can take the values 1, 2, 3, ···. If cycle shrinking is applied, a distance of 1 must be assumed. That amounts to a purely serial loop. In reality, however, the loop of Fig. 10 is not totally serial. Fig. 11 shows the unrolled version of this loop for the first ten iterations, and the corresponding data dependences. It is clear that some of the iterations (e.g., iterations 1, 2, 3, 5, 7, 9) could be executed in
parallel. RDC can detect such "unstructured" parallelism in statically serial loops.

The basic steps of the transformation are shown in Fig. 12. The transformation which is carried out by the compiler has two phases. An implicit and an explicit phase. The implicit part involves computations performed by the compiler which are transparent to the user. The explicit phase transforms the loop itself.

The scalar dependence graph or SDG of a loop is a dependence graph which shows dependencies based on the name of the variables involved and not on the subscripts. Many dependences in an SDG may be superfluous. The name of a dependence is the name of the variables involved in that dependence. Index expressions in array references are assumed to be 1-to-1 functions. Also loops are assumed to be normalized for simplicity in notation.

The basic idea is to be able to determine at run time whether

a particular iteration of a loop depends on one or more previous iterations. This requires some recording of relevant information from previous iterations. For a loop DO I = 1, N and for a dependence \( S_i S_j \) in that loop, we define the dependence source vector (or DSV) \( R_j \) to be a vector with \( N \) elements, where nonzero elements indicate the value of \( I \) for which \( S_i \) is a dependence source, and zero elements in \( R_j \) correspond to values of \( I \) for which \( S_j \) is not involved in a dependence.

The first step of the transformation is to create a DSV \( R_j \) for each flow dependence \( S_i S_j \) in a cycle. If \( e_i(I) \) is the index expression of the left-hand side of \( S_i \), then DSV \( R_j \) has subscripts in the range \( [e_i(I)] \cdots [e(N)] \), assuming index expressions are monotonically increasing/decreasing functions. (If the latter assumption is relaxed, the range is given by \([\min_{i \leq N} e_i(I) \cdots \max_{i \leq N} e_i(I)]\).) The compiler initializes all elements of \( S_j \) to zero. A single bit vector \( V \) with subscripts in the range \([1 \cdots N]\) is also created and is initialized to zero. Vector \( V \) is called the synchronization vector. Then for all values of \( j \), the compiler initializes the elements of \( R_j \) and \( V \) as follows:

```for (all values of j)
  B_j(e_i(I)) = I

V(U) = 1
```

Notice that only a single synchronization vector is needed for all dependences. This constitutes the implicit phase of

```RUN-TIME DEPENDENCE CHECKING

Input:
A cycle of static dependences \( S_1 \delta S_2 \delta \cdots \delta S_n \delta S_1 \).

IMPLICIT PHASE

Step 1:
For each static dependence \( \delta \) create the source vector \( R_\delta(1:N) \) and initialize it to zero. Assuming that array subscripts are 1-to-1 mappings the compiler executes the following loop:

```DOALL I = 1, N
  B_\delta[e_\delta(I)] = I
ENDOALL```

where \( e_\delta(I) \) is the value of the index expression of the source of the \( \delta \)-th static dependence when the loop index is \( I \).

EXPLICIT PHASE

Step 2:
For the \( \delta \)-th static dependence insert at the beginning of the target loop the following statement:

```IF (1 \leq R_\delta[e_\delta(I)] < I) THEN WAIT ON V[e_\delta(I)]
```

where \( e_\delta(I) \) is the value of the index expression of the end of the \( \delta \)-th static dependence when the loop index is \( I \).

Step 3:
For each static dependence \( \delta \) insert at the end of the target loop the following statement:

```CLEAR V[e_\delta(I)]
```

Fig. 12. The RDC transformation.
```
DOALL I = 1, N
COBEGIN
    IF (1 ≤ R_i(I+1) < I) WAIT ON V(I+1);
    IF (1 ≤ R_i(I-1) < I) WAIT ON V(I-1);
COEND
A(2I-1) = B(I-1) * 1
B(2I+1) = A(I+1) * C(I)
CLEAR V(I)
END

Fig. 13. The loop of Fig. 10 after the RDC transformation.

transformation. It creates a vector R_i for each dependence δ_i and stores in it the values of the original loop index I, for which we may have a potential dependence source. For each such dependence δ_i, the explicit phase of RDC inserts the statement

IF (1 ≤ R_i(h_i(I)) < I) THEN WAIT ON V(h_i(I))

at the beginning of the target loop, where h_i(I) is the index expression of the dependence sink at the right-hand side of statement S_{i+1}. When checking the value R_i(h_i(I)), we can have an out-of-bound condition if h_i(I) lies outside the interval [e_i(I) · · · e_i(N)]. In this case, however, no dependence is possible. At the end of the target loop, the compiler also inserts the statement

CLEAR V(I).

The effect of this is to detect possible unsatisfied dependences at run time, and synchronize the execution of loop iterations that are involved in a dependence. Iterations which are not possibly involved in a dependence are executed in parallel without any constraints. The clear statement resets the elements of V which correspond to statements with dependence sources that have completed execution. This in effect frees the statements (iterations) with the corresponding dependence sinks, to execute at any time. It is also worth noting that all the IF tests can be carried out in parallel as shown in Fig. 12 by the cobegin/coend clause. An iteration of the transformed loop is blocked if one or more of the tests fail.

The transformed loop of Fig. 10 is shown in Fig. 13. The appropriate declarations of vectors R and V are omitted. When the transformed loop is executed as a DOALL, only those iterations that are involved in a dependence will be sequentialized. All other iterations will be executed in parallel. For example, if N = 10 in Fig. 13, iterations 1, 2, 3, 5, 7, and 9 will be free to execute in parallel immediately. Iterations 4, 6, and 8 must wait for iterations 3 and 5 to complete. Finally iteration 10 must wait for iteration 6 to complete. In a system with six processors, the transformed loop will take only three cycles to complete as opposed to ten cycles that would be required for the original serial loop.

Finally, let us consider the extra storage requirements introduced by the transformation. For each dependence handled by RDC, we need to maintain a dependence source vector. For all practical purposes, we can assume that the size of these vectors is equal to the number of iterations of the loop. Therefore, for a loop with N iterations and k dependences, the storage requirements of RDC grow as O(kN), i.e., linearly on the size of the loop.

V. HARDWARE BARRIER SYNCHRONIZATION

As pointed out earlier, the cost of barrier synchronization depends on the machine architecture, the type of code (loop, cobegin/coend, etc.), and the number of processors involved in updating the barrier, just to mention a few. Most of the factors that influence the value of γ are dynamic. Many researchers have emphasized the significance of barrier overhead on the granularity of parallelism that can be exploited on a parallel machine [12], [18], [26]. Since very little about barrier overhead can be predicted and dealt with a priori, the most significant reduction of this overhead can be obtained at the hardware design level. Optimizing barrier operations at the hardware level may be costly, but it may also prove to be an excellent "investment" for many applications.

We have followed this approach by extending and improving a hardware barrier synchronization mechanism proposed in [27]. A detailed presentation of this design and simulation results can be found in [28]. Here we can only give a sketchy description of the main idea. Let us describe the basic mechanism in the context of cycle shrinking.

Typically, the effect of cycle shrinking would be to transform a serial loop into a doubly nested loop, where the inner loop is a DOALL and the outer loop is serial. During parallel execution of such a loop, all λ iterations of the inner DOALL must complete before the next iteration of the outer serial loop can start. This ordering can be enforced with barriers. A barrier can be viewed as a program-defined variable which is used as a counter. In fact, many implementations of barrier synchronization use this approach [34]. In our example, a barrier can be associated with the inner DOALL. The value of the barrier is initially set to λ, the number of loop iterations (reduction factor), assuming that loop is executing on λ processors.

As each processor finishes its own iteration, it decrements the barrier by 1, and checks whether its value has become 0. Processors that find the value of the barrier to be nonzero are blocked. Eventually, the last processor to finish sets the barrier to 0, and finally all processors are freed to proceed with the next iteration of the outer loop. All these update and test operations are atomic (indivisible). For large values of λ, the resulting overhead due to contention at the barrier can be very significant.

This overhead may become even more critical when the size of the loop body is relatively small. Thus, the effectiveness of cycle shrinking and many other compiler transformations depend heavily on the cost of barrier synchronization. Any hardware synchronization primitive can be used to implement barrier synchronization in software. This approach, however, would be very costly (in time), especially for loops with small loop bodies.

The barrier mechanism that we proposed involves the use of special bit-addressable registers. Instead of full words, we use single bit barriers that are set (1), or cleared (0). When a barrier associated with a loop is set, no incoming processor is allowed to dispatch new iterations. When several processors try to update a barrier simultaneously, all but one will be denied access to the barrier and they will be forced, for example, into a busy waiting. The busy wait generates several
unnecessary requests which, in the case of shared memory machines, clog up the network. This is known as the "hot-spot" phenomenon [25].

Combining networks [12], [25], [18] can be used to combine simultaneous update requests, thus reducing or eliminating contention due to barriers. However, processors that find a barrier set busy, wait issuing unnecessary tests until the barrier is cleared. In a combining network, these tests can clearly be combined but, nevertheless, excessive and unnecessary traffic can still be generated. Moreover, combining cannot be used in a bus-connected multiprocessor and it has been indicated that it is very costly.

Our solution is more general (in the sense that the network type is irrelevant), and it avoids both hot spots (since there is no busy waiting) and unnecessary processor latencies. The notion of barriers as shared variables is in effect eliminated. We discuss this scheme in the context of loop barriers although the solution is general. In our case, a barrier is not a shared variable but instead a single bit register BR that is writable (set) by a single processor which is determined dynamically during execution. The bit barriers are also cleared automatically by the special hardware without the intervention of the processors, or the operating system. The basic hardware module that implements barrier synchronization is shown in Fig. 14. At most p/2 such units are needed in a machine with p processors. The module consists of a bit addressable register R, an enable switch, a module that checks for the all 0's condition in log₂ p time, and a one-bit barrier register BR. The R register is p bits wide and the i(th) bit R(i) can be set/cleared only by the i(th) processor in the machine (i = 1, 2, ..., p). (This can be done in software by associating an id number with each processor, or by directly hardwiring the i(th) bit of each R register to the i(th) processor.)

Consider the case of the doubly nested loop mentioned above, and let us denote the inner DOALL with L₂ and the outer serial loop with L₁. L₁ uses a BR register (initially cleared) as its barrier (the corresponding R register is also cleared). The first processor to dispatch an iteration of L₁ will set BR to 1 disabling, therefore, any other processor from dispatching new iterations of L₁. Several processors, though, can get different iterations of L₂. Each processor that dispatches one or more iterations of L₂ goes through a cycle of setting its corresponding R bit to 1 (when dispatching a new iteration), and resetting that bit to 0 if it completes. (Thus, the same bit in an R register may be set/cleared several times if the same processor dispatches several different processes of the same task.) The processor that dispatches the last iteration(s) of L₂ will also enable the switch in the module (Fig. 14). From the time the enable switch is set on (by the last processor) the module will start testing for the all 0's condition. This can be done in a fan-in fashion and can also be pipelined so that the condition is tested every clock cycle thereafter. When the all 0's condition becomes true, the BR register is cleared automatically. If the last processor to work on the task executes for more than log p clock cycles, then this barrier synchronization mechanism involves zero overhead. Also, processors do not need to busy wait and repeatedly test for BR = 0. They can either execute tasks that are not constrained by that barrier, or they can be suspended until an interrupt, generated by the barrier module and broadcast to all relevant processors, arrives (not shown in Fig. 14).

The user program can access only the BR registers and the remaining configuration is transparent to the user. BR registers are associated with loop indexes and accessing a barrier is now part of accessing a loop index (an operation that must be done anyway). Any conflicts that may occur now will occur during accesses to loop indexes but not to barrier variables. Therefore, all the overhead associated with the explicit manipulation of software barriers is eliminated in this case. The full configuration with interconnected barrier modules and other design tradeoffs, as well as performance estimates, are discussed in [28].

VI. Conclusions

Parallelism in algorithms and programs may be implicit, or may be explicitly specified at several different levels. When parallelism exists in fixed-size "quantums," it is rather easy to understand and exploit. The unstructured nature of parallelism makes it efficient exploitation and parallel programming very complex tasks. Designing methods and tools that automatically perform these tasks is thus a very important research subject.

In this paper, we discussed cycle shrinking and run-time dependence checking, two such compiler schemes that can be used to automatically transform serial loops to a parallel form. Both cycle shrinking and RDC aim to parallelize serial loops in different ways. Given a loop with a dependence cycle, one or the other scheme will be more appropriate for that loop. Which method is more appropriate in each case should be the compiler's responsibility to discover with relatively little effort. Cycle shrinking performs the loop parallelization at compile time, and packages loop parallelism in "quantums." Run-time dependence checking, however, prepares the loop for run-time parallelization. This is achieved by inserting appropriate code in the source program, which automatically performs dependence checking and bookkeeping during program execution. In this case, parallelism is exploited at run time in an unstructured fashion.

We also proposed a hardware solution for barrier synchronization, which greatly reduces the overhead associated with nested serial and DOALL loops. Fast barrier synchronization is very important for such loops due to the presence of
frequent fork/join operations. The barrier module outlined here is an example of specialized hardware components which can be used to carry out vital operations in a fast and efficient manner. Such components will become more frequent in future high-performance parallel machines.

ACKNOWLEDGMENT

The author would like to thank the anonymous referees for many useful comments and suggestions. Many thanks are also due to Dr. U. Banerjee and M.-R. Haghighat for reading earlier versions of this paper and providing many important comments.

REFERENCES


Constantine D. Polychronopoulos (S’85-M’86) was born on April 5, 1958, in Patras, Greece. He received the Dipl. degree in mathematics from the University of Athens, Athens, Greece, in 1980, the M.S. degree in computer science from Vanderbilt University, Nashville, TN, in 1982, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1986. Currently he is an Assistant Professor in the Department of Electrical and Computer Engineering, and a Senior Software Engineer at the Center for Supercomputing Research and Development, of the University of Illinois. At CSRD he participates in research and development of parallelizing compilers and is in charge of the Parafase II project. His research interests include restructuring compilers, scheduling, synchronization and interprocessor communication, and architectures for parallel processing.

Dr. Polychronopoulos has served on the program committees for the International Conference on Supercomputing, the International Symposium on Computer Architecture, and Supercomputing ‘88. He was a Fulbright Scholar in 1981-1982 and he is a member of the Association for Computing Machinery and the IEEE Computer Society.