CSRD Technical Report 1349

Parallelizing While Loops for Multiprocessor Systems†

Lawrence Rauchwerger and David Padua

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801

Corresponding Author: Lawrence Rauchwerger.
email: rwerger@csrd.uiuc.edu telephone: (217) 333-6578, fax: (217) 244-1351.

Abstract

Current parallelizing compilers treat `while` loops and `do` loops with conditional exits as sequential constructs because their iteration space is unknown. Motivated by the fact that these types of loops arise frequently in practice, we have developed techniques that can be used to automatically transform them for parallel execution. We succeed in parallelizing loops involving linked lists traversals — something that has not been done before. This is an important problem since linked list traversals arise frequently in loops with irregular access patterns, such as sparse matrix computations. The methods can even be applied to loops whose data dependence relations cannot be analyzed at compile-time. We outline a cost/performance analysis that can be used to decide when the methods should be applied. Since, as we show, the expected speedups are significant, our conclusion is that they should almost always be applied — providing there is sufficient parallelism available in the original loop. We present experimental results on loops from the PERFECT Benchmarks and sparse matrix packages which substantiate our conclusion that these techniques can yield significant speedups.

†Research supported in part by Intel and NASA Graduate Fellowships and Army contract #DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

‡A preliminary version of this paper will be presented at the *Proceedings of 9th International Parallel Processing Symposium*, April 1995, Santa Barbara, CA.

1 Introduction

Most current parallelizing compilers treat `while` loops and `do` loops with conditional exits as sequential constructs. Since these types of loops arise frequently in practice, techniques for extracting their available parallelism are highly desirable.

In the most general form, we define a `while` loop as a loop that includes one or more *recurrences* that can be detected at compile time, a *remainder*, whose dependence structure can be either analyzed statically (as being parallel or sequential) or is unknown at compile time, and one or more *termination conditions*. Sometimes the termination conditions form part of one of the recurrences, but they can also occur in the remainder, e.g., *conditional exits* from `do` loops. Assuming, for simplicity, that there are no cross-iteration data dependences in the *remainder*, there are two potential problems in the parallelization of `while` constructs:

- *Evaluating the recurrences.* If the recurrences cannot be evaluated in parallel, then the iterations of the loop must be started sequentially, leading in the best case to a pipelined execution (also known as a `doacross`).

- *Evaluating the termination conditions.* If the termination conditions (loop exits) cannot be evaluated independently by all iterations, the parallelized `while` loop could continue to execute beyond the point where the original sequential loop would stop, i.e., it can *overshoot*.

Although the concurrent evaluation of recurrences is in general not possible, some special cases lend themselves to either full or partial parallelization. There are parallel algorithms to solve simple inductions (the case of `do` loops) [30] and associative recurrences [5, 15, 12, 13] but the evaluation of general recurrences has always been of a sequential nature. The concurrent evaluation of the `while` loop termination condition has been dealt with only in the case when it is loop invariant with respect to the remainder (a property we define later as *remainder invariant*). In other words, the exit conditions that have been dealt with so far are those dependent on the terms of the recurrence, and since these recurrences are executed sequentially, the exit conditions do not pose a problem for parallelization.

The task of parallelizing a `while` loop becomes even more difficult if the data dependence structure in the remainder cannot be determined statically. For example, there may exist additional recurrences in the the remainder that cannot be detected by the compiler. For input data dependent irregular access patterns this problem is intractable with traditional compile–time methods and has not been addressed so far.

In this paper we present a general framework for the automatic transformation of any `while` loop for parallel execution, provided that its remainder is indeed parallel. The basic strategy of our methods is to evaluate in parallel the recurrences that can be statically identified and speculatively execute remainder concurrently, and then later, to "undo" the effects of any iterations that overshot the termination condition, i.e., iterations that would not have been performed by the original sequential version of the loop. We describe techniques for parallelizing loops involving linked list traversals. This is an important problem since linked list traversals arise frequently in loops with irregular access patterns, such as sparse matrix computations. In many cases, our parallelization of loops involving linked lists can be done without overhead or side effects.

Our framework for `while` loop parallelization can be viewed as a step toward providing `while` loop counterparts for the existing constructs for parallel execution of `do` loops, e.g., `doall`, `doacross`, `doany`. These new parallel constructs could be called `while-doall, while-doacross`, and `while-doany` and could prove useful in the parallel programming (manual parallelization) of applications.

The methods described here extend previous works [9, 30] in that they:

1. can handle remainder variant termination conditions,

2. can test at run-time for cross-iteration data dependences in the remainder,

3. do not require work and storage for saving the values computed in the recurrence,

4. support both static and dynamic scheduling, and

5. present a comprehensive analysis and solution package for parallelizing `while` loops for multiprocessors.

The techniques are capable of extracting a substantial fraction of available parallelism in a loop. In particular, we show that in the worst case our techniques will extract *at least* 20-25% of the parallelism inherent in the loop, which can amount to significant speedups on massively parallel processors. Therefore, we conclude that as long as there is

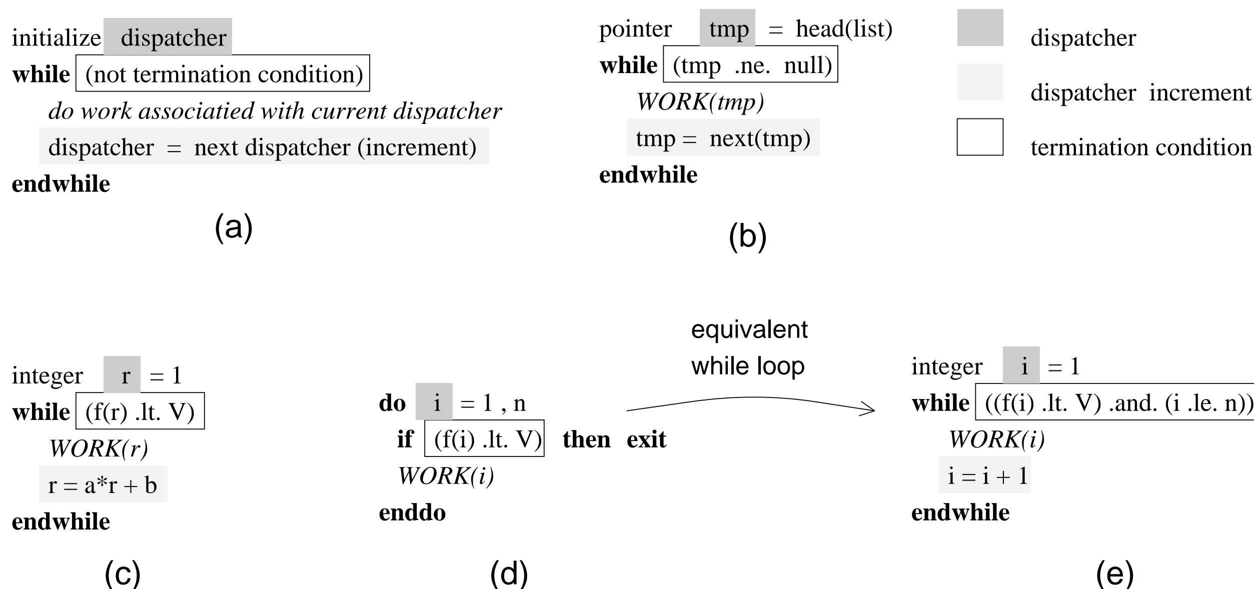

Figure 1:

sufficient available parallelism in the loop, our techniques will prove to be beneficial. We have obtained experimental results on loops from the PERFECT Benchmarks and sparse matrix packages on the Alliant FX/80 which substantiate this conclusion.

We begin in Section 2 by introducing a taxonomy of `while` loops based on the difficulties they present for parallelization. Then, in Sections 3 and 4, for each case in the taxonomy we give the necessary transformations and methods for automatically parallelizing the `while` loop, under the assumptions that there are no cross–iteration data dependences in remainder, and that there is only one recurrence controlling the loop. In Section 5, these methods are augmented to include loops whose access patterns cannot be analyzed at compile-time, and in Section 6 we describe how loops with an arbitrary number of recurrences can be handled. In Section 7, we discuss a cost/performance analysis that can be used to determine when our methods should be applied, and in Section 8 we give some strategies for reducing the overhead of our methods. We present some experimental results in Section 9. In Section 10 we discuss related work.

2 Transforming `While` Loops for Parallel Execution

`While` loops have often been treated by parallelizing compilers as an intrinsically sequential constructs because their iteration space is unknown [9]. A related case which is generally also handled sequentially by compilers is the `do` loop with a conditional exit. In this paper we propose techniques that can be used to execute such loops in parallel. In order to clarify our presentation we first consider loops which (a) *contain a single statically detectable recurrence*, and (b) *have no cross–iteration data dependences except those in this recurrence*. Later, we relax these constraints and show how to deal with loops with multiple recurrences and unknown cross–iteration data dependences.

In this case, a `while` loop can be considered as a parallel loop controlled by a recurrence. In general `while` loops can exhibit several dependent or independent (of one another) recurrences. We call the dominating recurrence, which precedes the rest of the computation in the dependence graph, the *dispatching recurrence*, or simply the *dispatcher* (see Figure 1(a)). In the most general case, the terms of the dispatcher must be evaluated sequentially. An example of this case is a pointer used to traverse a linked list; since the values of the dispatcher (the pointer) must be evaluated in sequential order, iteration i of the loop cannot be initiated until the dispatcher for iteration $i-1$ has been evaluated (see Figure 1(b)). However, sometimes the evaluation of the terms of the dispatching recurrence can be parallelized. In particular, if the dispatcher is an associative recurrence, then the computation of its terms can be parallelized using techniques such as parallel prefix computations (see Figure 1(c)). Finally, in the best case, the dispatcher has the simpler form of an induction, and each point in the dispatcher's domain can be independently and concurrently

Loop Terminator	Dispatcher							
	Monotonic Induction		Not Monotonic Induction		Associative Recurrence		General Recurrence	
	Overshoot	Parallel	Overshoot	Parallel	Overshoot	Parallel	Overshoot	Parallel
RI	NO	YES	YES	YES	YES	YES-PP	NO	NO
RV	YES	YES	YES	YES	YES	YES-PP	YES	NO

Table 1: A taxonomy of `while` loops and their dispatcher's potential for parallel execution. The notation PP implies parallelizable with a parallel prefix computation.

evaluated using the closed form solution of the induction. In this case, all iterations of the `while` loop can be executed simultaneously since aside from the dispatching recurrence we assumed no other dependences. An example of a dispatcher with a closed form solution is a `do` loop (see Figure 1(d-e)).

Another difficulty with parallelizing a `while` loop is that the termination condition (*terminator*) of the loop may be *overshot*, i.e., iterations could be executed that would not be executed by the sequential version of the loop. In the context of our analysis we define the terminator as *remainder invariant* or *RI* if it is only dependent on the dispatcher and values that are computed outside the loop; if it is dependent on some value computed in the loop then it is considered to be *remainder variant* or *RV*. If the terminator is RV, then iterations larger than the last valid iteration could be performed in a parallel execution of the loop, i.e., iteration i cannot decide if the terminator is satisfied in the remainder of some iteration $i' < i$. Overshooting may also occur if the dispatcher is an induction, or an associative recurrence, and the terminator is RI. An exception in which overshooting would not occur is if the dispatcher is a monotonic function, and the terminator is a threshold on this function, e.g., $d(i) = i^2$, and $tc(i) = (d(i) < V)$, where V is a constant, and $d(j)$ and $tc(j)$ denote the dispatcher and the terminator, respectively, for the jth iteration. Overshooting can also be avoided when the dispatcher is a general recurrence, and the terminator is RI. For example, the dispatcher tmp is a pointer used to traverse a linked list, and the terminator is $(tmp = null)$ (see Figure 1(b)). In the most general case, the exit from a `while` loop may be caused by one of many termination conditions; this situation which will require a combination of several solutions.

From the discussion above we conclude that the techniques needed to parallelize a `while` loop depend on the type of its *dispatcher* and *terminator*. We can therefore summarize our discussion through the taxonomy of `while` loops given in Table 1.

In the next section we discuss methods for parallelizing `while` loops under the two previously mentioned assumptions:

1. There is one and only one recurrence (dispatcher), which can be detected statically
2. The only cross-iteration data dependences in the loop are carried by the controlling recurrence (the dispatcher)

Later, in Sections 5 and 6, we show how our methods can be generalized when these two restrictive conditions are relaxed.

3 Parallelizing the Dispatcher

Clearly, the most important factor affecting the amount of available parallelism in a `while` loop (assuming no cross-iteration dependences) is the amount of parallelism available in its dispatching recurrence. To aid our analysis of the dispatching recurrence, it is convenient to extract, at least conceptually, this recurrence from the original `while` loop by *distributing* [28] the original loop into two `do` loops with conditional exits:

1. A loop that evaluates the terms of the dispatcher (recurrence) and any termination condition that is strongly connected to the dispatcher.
2. A loop consisting of the remainder loop which uses the values of the recurrence (computed by the first loop), and its associated termination condition.

`while` Loop: induction
```
integer i = 1
while (f(i))
   work(i)
   i = i + 1
endwhile
```

Induction-1
```
integer L[0:nproc-1] = u
doall i = 1,u
   if (L(vpn).gt.i) then
      if (not f(i)) then L[vpn] = i
      else work(i)
   endif
enddo
LI = min(L[1:nproc])
```

Induction-2
```
integer L[0:nproc-1] = u
doall i = 1,u
   if (not f(i)) then
      L[vpn] = i
      QUIT
   endif
   work(i)
enddo
LI = min(L[1:nproc])
```

Figure 2: Parallelizing `while` Loops when the dispatcher is an induction. In the `doall`s, *nproc* is the number of processors, u is an upper bound on the number of iterations of the `while` loop, and vpn is the virtual processor number of the processor executing the iteration.

Note that the original set of termination conditions is distributed appropriately to the two loops. Thus, the first loop may or may not have a termination condition, and the second loop is either a simple `do` loop, or a `do` loop with a conditional exit.

In order to perform the data dependence analysis necessary for loop distribution all array references in the `while` loop have to be associated with a loop counter. We remark that a proper distribution is not possible ([11]) if the dependence structure of body of the loop consists of a single strongly connected component. In either case, for the purposes of parallelizing the dispatcher, the techniques proposed in the sequel remain the same.

Once the dispatcher has been extracted we can attempt to parallelize it. As discussed in the previous section, the extent to which this is possible depends upon the form of the recurrence itself. In its most general form a recurrence can be evaluated only through a sequential method. However, if the recurrence is associative, then parallel algorithms like parallel prefix computations can speed the task of computing its terms by a significant factor, and if the recurrence is an induction, then its evaluation can be done in fully concurrent mode by evaluating its closed form.

In the remainder of this section we present techniques that can be used to extract the maximum available parallelism from `while` loops with one dispatching recurrence. For the cases in which the dispatcher is not an induction, our methods assume that the dispatching recurrence is fully determined before loop entry (e.g., if the dispatcher is traversing a linked list, no list elements may be inserted or deleted during loop execution). Although not all of our methods are fully parallel, they can yield very good speedups – especially if a significant amount of work is performed in the loop body.

We describe the methods without addressing the overshooting problem, and then discuss in Section 4 how they can be augmented to "undo" any iterations that overshot the termination condition. We also assume that there are no cross-iteration dependences in the loop other than those associated with the dispatcher. This restriction is removed in Section 5 where we describe how `while` loops can be speculatively executed in parallel by combining our methods with run-time techniques for detecting the presence of cross-iteration dependences in the loop.

Finally we should consider the case when the loop evaluating the recurrence does not contain a termination condition and therefore does not in itself impose an upper limit on the number of terms to be computed. In this case an upper bound can be inferred from the body of the `while` loop. If that is not possible then the two distributed loops can be executed in a strip–mined fashion until the termination condition is reached, effectively limiting the number of precomputed recurrence terms to the length of the strip.

3.1 The Dispatcher is an Induction

In this section we consider a `while` loop in which the dispatcher $d(i)$ is an induction of the generic form:

$$d(i) = c * i + b$$

where c and b are constants. To simplify our discussion, we assume that the dispatcher of the ith iteration is i, i.e, $d(i) = i$. The fact that all processors can evaluate the dispatcher simultaneously from a closed form solution of the induction relation makes loop distribution and precomputation of the recurrence terms unnecessary.

In this method, referred to as *Induction-1*, the loop is run as a `doall` and a test of the termination condition of the `while` loop is inserted into the loop body (see Figure 2). During the parallel execution, each processor keeps track of the lowest iteration it executed that met the termination condition. Then, after the `doall` has terminated,

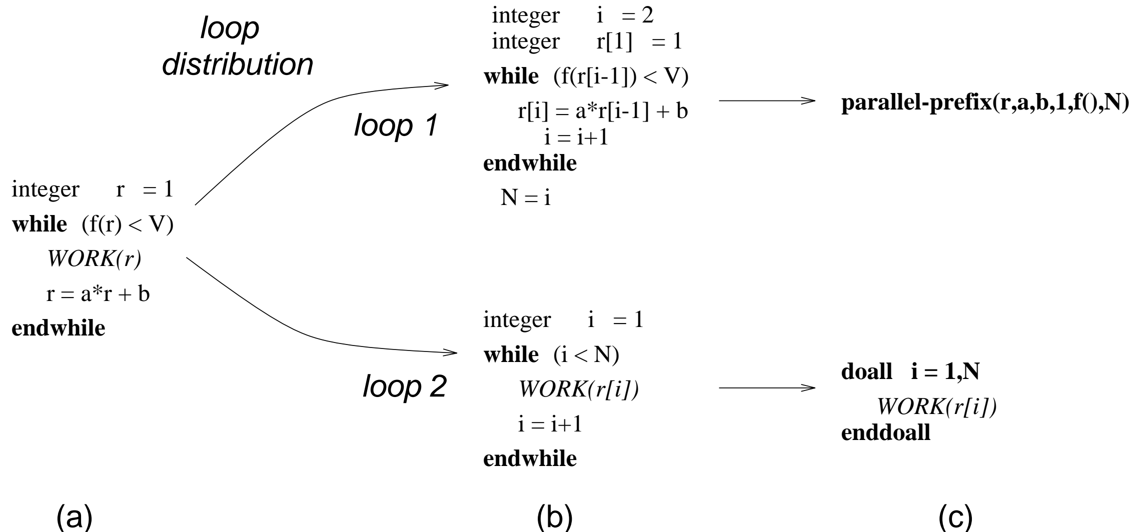

Figure 3: The loop in (a) with the associative dispatcher is distributed into the two loops in (b), which are then transformed into the parallel prefix computation and the `doall` shown in (c).

the last iteration that would have been executed by the sequential version of the `while` loop is found by taking the minimum of the processor-wise minima. This iteration must be found so that any iterations that need to be undone can be identified.

On computers, such as the Alliant [1], in which iterations are issued in order, the test $L[vpn] > i$ is unnecessary. In order to terminate the parallel loop cleanly before all iterations have been executed, a *QUIT* operation similar to the one on Alliant computers [1] could be used. Once a QUIT command is issued by an iteration, all iterations with loop counters less than that of the issuing iteration will be initiated and completed, but no iterations with larger loop counters will be begun. If multiple QUIT operations are issued, then the iteration with the smallest loop counter executing a QUIT will control the exit of the loop. An optimized version of the method is shown as *Induction-2* in Figure 2.

3.2 The Dispatcher is an Associative Recurrence

We now consider a `while` loop in which the dispatcher is an associative recurrence. Examples of such dispatchers can have the form:

$$x(i) = a * x(i-k) + b \qquad \text{or} \qquad x(i) = a * x(i-k)^b$$

where $i = 1, n$ and a, b and k are constants. The terms of this relation can be evaluated for $i = 1, n$ using a parallel prefix computation. This technique has been well documented in the literature ([16]) and gives a logarithmic speedup, i.e., it can be done in $O(n/p + \log p)$ time, where p is the number of processors and n is the number of terms to be computed. Thus, after loop distribution, the first loop can be transformed into a parallel prefix computation, and the second loop can be executed as a `doall` using the terms of the dispatcher which were computed by the first loop. An example is shown in Figure 3.

In the example, no overshooting occurs because the terminator is a threshold function and the dispatcher is monotic increasing. However, if this had not been the case, then overshooting might have occurred and we we would have also needed to find the last valid iteration in order to decide which iterations to undo. Unfortunately, this cannot be done in loop 1 (of the distributed loop) without bringing additional computation from the loop body into loop 1. In this situation, it is probably best to strip-mine the loop, and to find the last valid iteration inside loop 2 (in the same manner as in Figure 2 for *Induction-1*). The drawback of this approach is that loop 1 could potentially calculate a large number of superfluous values of the dispatcher.

```
*General-1*
ptr tmp = head(list)
doall i = 1,u
      ptr pt
      lock(list)
      pt = tmp
      tmp = next(tmp)
      unlock(list)
      if (pt .eq. null) QUIT
      work(pt)
enddoall
```

```
*General-2*
doall i = 1, nproc
      integer j
      ptr pt
      pt = head(list)
      do j = 1,vpn
         pt = next(pt)
         if (pt .eq. null) goto 2
      enddo
1     work(pt)
      do j = 1,nproc
         pt = next(pt)
         if (pt .eq. null) goto 2
      enddo
      goto 1
2     continue
enddoall
```

```
*General-3*
doall i = 1, u
      integer j, prev
      ptr pt
      prev = 1
      pt = head(list)
loop
      do j = 1, i - prev
         pt = next(pt)
         if (pt .eq. null) QUIT
      enddo
      work(pt)
      prev = i
enddoall
```

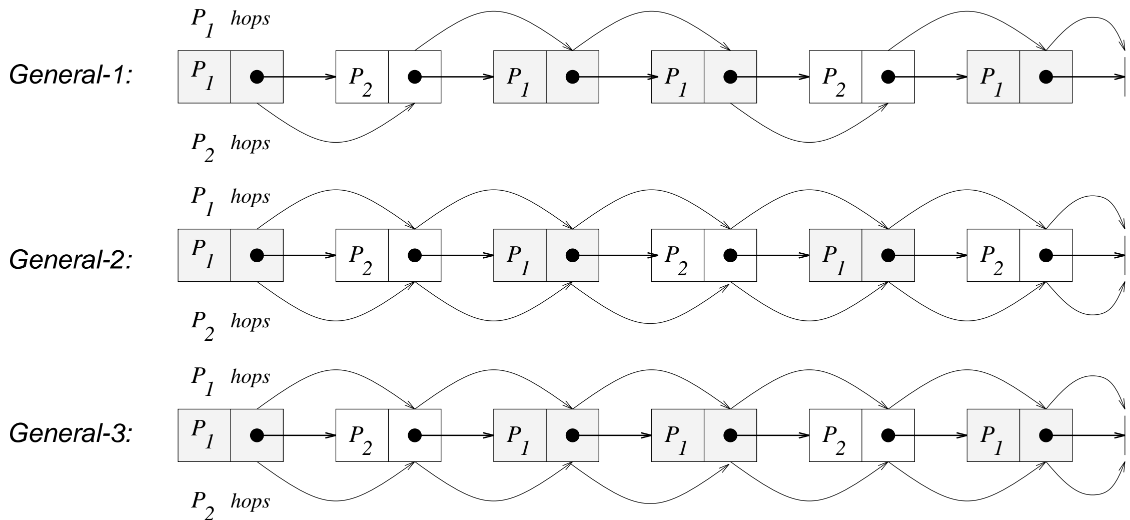

Figure 4: In the `doallS`, u is an upper bound on the number of iterations of the `while` loop, *nproc* is the number of processors, and vpn is the virtual processor number of the processor executing the iteration. In General-3, the operations before the **loop** label are executed just once by each processor, and the operations after the **loop** label are executed for every iteration.

3.3 The Dispatcher is a General Recurrence

This section presents several methods for parallelizing loops with inherently sequential dispatchers. These techniques do not attempt to parallelize the dispatcher itself since, as we have mentioned before, it can be represented by a continuous chain of flow dependences. Instead we are trying to speedup the `while` loop as a whole by taking advantage of the parallelism of the remainder of the `while` loop body, i.e., we attempt to maximize the overlap between iterations. Thus, if there is not sufficient parallelism available in the loop remainder, then the original `while` loop should be executed sequentially. For simplicity, we describe the methods as applied to a `while` loop that traverses a linked list.

We first notice that when loop distribution is applied the evaluation of the dispatcher is completely sequential, i.e., we cannot perform a parallel prefix computation. In this case, since the parallel execution of the remainder cannot be started before all the terms of the dispatcher have been computed sequentially, it is not clear that the restructuring of the `while` loop into a sequential dispatcher loop and a parallel remainder will be beneficial. This is especially true if the terminator is RV since the loop distribution scheme would either involve moving portions of the parallel remainder containing the termination condition to the *sequential* recurrence loop, or entail the *sequential* computation of unneeded terms of the dispatcher (those beyond the last iteration) which are stored in additional memory. It is possible that strip-mining the loop could improve these costs for RV–type terminators. However, this option would drastically increase the overhead of parallelization since the strips would then be executed as a `doalls`, separated by

synchronization barriers. In fact, it is even possible that a *slowdown* could occur due to this increased overhead.

We now describe several methods that enable iterations of the loop body to be executed concurrently but do not use loop distribution. One simple method, referred to as *General-1*, is to serialize the accesses to the *next*() operation. This technique is equivalent to hardware pipelining which has been well studied in the literature [10]. The cost of synchronization and the limited amount of parallelism may make this scheme unattractive. A method, *General-2*, which avoids explicit serialization, is to compute the whole recurrence in each processor in private storage and assign to processor i the privatized values k of the recurrence such that $k = i \bmod nproc$, where *nproc* is the total number of processors. A third method, *General-3*, dynamically assigns iterations to the processors like *General-1*, and also avoids explicit serialization like *General-2*. In this method, each processor records the last iteration that it processed ($prev$) and the value of the recurrence at that point (pt). Then, when it is assigned a new iteration i, it calculates the values of the recurrence between $prev$ and i. Examples of all three methods for the `while` loop of Figure 1(b) are shown in Figure 4.

We first contrast the loop distribution approach with the general strategy of embedding the sequential evaluation of the dispatcher inside the parallel execution of the loop as is done in the other methods described above. Notice that the performance of both strategies is likely to be similar for loops in which the terminator is RI, i.e., when overshooting is not possible. However, when the terminator is RV, the loop distribution approach would prove to be inferior due to the reasons mentioned above, i.e., the extra sequential computation performed in loop 1, or when strip-mining the costs of the associated `doalls` and synchronizations, none of which are concerns for the other methods.

We now consider the relative advantages/disadvantages of the methods that do not use loop distribution. In addition to the fact that *General-1* explicitly serializes accesses to $next()$, and no such serialization is used in *General-2* or *General-3*, there are some other differences between the methods. First, in *General-1* the recurrence is computed (the list is traversed) just once by all processors cooperatively, but in *General-2* and *General-3* each processor computes the entire recurrence. Second, in *General-1* and *General-3* the values of the recurrence are dynamically allocated to the processors, but in *General-2*, processor i, $0 \leq i < nproc$, is statically assigned values congruent to $i \bmod nproc$. Another point, related to this last difference, is that the iteration execution pattern of the methods that dynamically assign iterations to processors (*General-1* and *General-3*) may be quite different from that of *General-2* that statically assigns iterations to processors. In particular, the span of iterations (i.e., difference between the lowest and smallest iteration numbers) that are executing at any given time might be larger for the static assignment method than for the dynamic assignment method. If the termination condition of the loop is RV, then it is likely that more iterations would need to be undone in the static assignment method than in the dynamic assignment methods.

In the example `while` loop from Figure 1(b), no overshooting occurs because the termination condition is RI. However, if the termination condition had been RV, then overshooting might have occurred and in order to determine which iterations needed to be undone, we would have also needed to find the last valid iteration. This could be done in the same manner as shown in Figure 2 for *Induction-1*. With the loop distribution method the performance would be taxed additionally with the *serial* computation of the overshot values of the dispatcher.

4 Undoing Iterations that Overshoot the Termination Condition

Perhaps the easiest method for "undoing" iterations that overshot the termination condition is to checkpoint prior to executing the `doall`, and to maintain a record of *when* (i.e., iteration number) a memory location is written during the loop. Note that since all iterations of the `while` loop have been assumed independent, each memory location will be written during at most one iteration of the loop. Then, after the `doall` has terminated and the last valid iteration is known, the work of iterations that have overshot can be undone by restoring the values that were overwritten during these iterations. This solution may require as much as three times the actual memory needed by the original `while` loop: one copy for checkpointing, one for the actual loop data, and one for the time-stamps. It is possible that this increase in memory requirements might degrade the performance of the parallel execution of the `while` loop. However, in some situations the memory usage is quite reasonable and, as we will show in Section 9, this scheme performs very well.

It might be possible to reduce the cost of checkpointing by identifying and checkpointing a point of minimum state in the program prior to the parallel execution of the `while` loop. Alternatively checkpointing could be avoided by privatizing all variables in the loop, copying in any needed values, and copying out only those values that are live after the loop and have time-stamps less than or equal to the last valid iteration. Privatized variables need not be backed up because the original version of the variable can serve as the backup since it is not altered during the parallel execution.

```
do i = 1, n
  if (f(i) .eq.  true)
    then exit
  A[i] = 2*A[i]
enddo
```

(a)

```
    do i = 1, n/2
      if (f(i) .eq.  true)
        then exit
s4:   tmp = A[2*i]
      A[2*i] = A[2*i-1]
s6:   A[2*i-1] = tmp
    enddo
```

(b)

```
    do i = 2, n
      if (f(i) .eq.  true)
        then exit
s4:   A[i] = A[i] + A[i-1]
    enddo
```

(c)

Figure 5:

If the access pattern of any array in the loop is known to be sparse, then the memory requirements could be reduced by using hash tables for the private version of the array. Less memory would be needed in this case since only the elements of the array accessed in the loop would be inserted into the hash table.

A simple way to reduce the memory requirements is to *strip mine* the loop, i.e., execute the first s iterations, then the next s iterations, etc., for some suitable value of s. Then, the memory needed to maintain the time-stamps would be bounded by the product of s and the number of write accesses performed in an iteration. However, this method would introduce global synchronization points in the `doall` and potentially reduce significantly the amount of obtainable parallelism. A better method of controlling memory usage at the application level is discussed in section 8.2.

Finally, we mention that time-stamping can be avoided completely if one is willing to execute the parallel version of the `while` loop twice. First, the loop is run in parallel to determine the number of iterations (using one of the methods of Sections 3.1-3.3). Then, since the number of iterations is known, the second time the loop can simply be run as a `doall`.

5 `While` Loops with Unknown Cross-Iteration Dependences

We now consider `while` loops for which the compiler cannot statically determine the access pattern of a shared array A that is referenced in the loop. The dependences between the statements referencing the shared array may be difficult and/or impossible for the compiler to analyze for a number of reasons: very complex subscript expressions which could only be computed statically through deeply nested forward substitutions and constant propagations across procedure boundaries, nonlinear subscript expressions (a fairly rare case) and, most frequently, subscripted subscripts.

The iterations of such a loop can be executed in parallel, without synchronization, if and only if the desired outcome of the loop does not depend in any way upon the execution ordering of the data accesses from different iterations. In order to determine whether or not the execution order of the data accesses affects the semantics of the loop, the *data dependence* relations between the statements in the loop body must be analyzed [20, 14, 2, 28, 31]. There are three possible types of dependences between two statements that access the same memory location: *flow* (read after write), *anti* (write after read), and *output* (write after write). Flow dependences express a fundamental relationship about the data flow in the program. Anti and output dependences, also known as memory-related dependences, are caused by the reuse of memory, e.g., program variables.

If there are no anti, output, or flow dependences between accesses in different iterations of a loop, then all the iterations of the loop are independent and the loop, as is, can be executed in parallel. For example, there are no dependences in the loop in Figure 5(a). If there are flow dependences, then the iterations are not independent because values that are computed (produced) in some iteration of the loop are used (consumed) during some later iteration. For example, the loop in Figure 5(c) has a flow dependence between statement `s4` of iteration $i-1$ and statement `s4` of iteration i. In order to remove certain types of memory-related dependences a transformation called *privatization* can be applied to the loop. Privatization creates, for each processor cooperating on the execution of the loop, private copies of the program variables that give rise to anti or output dependences (see, e.g., [4, 17, 19, 25, 26]). For example, the anti dependences between statement `s6` in iteration $i-1$ and statement `s4` in iteration $i+1$ in the loop shown in Figure 5(b) can be removed by privatizing the temporary variable `tmp`. We use the following criterion to determine whether a variable may be privatized.

Privatization Criterion. Let A be a shared array that is referenced in a loop L. A can be *privatized* if and only if every read access to an element of A is preceded by a write access to that same element of A within the same

iteration of L.

Note that, according to the above criterion, a variable cannot be privatized if it is initialized by reading a value that is computed outside the loop. Such variables could be privatized if a *copy-in* mechanism for the external value is provided. The *last value copy-out* problem is the conceptual analog of the copy-in problem. If a privatized variable is *live* after the termination of the loop, then the privatization technique must ensure that the correct value is copied out to the original (non privatized) version of that variable. It should be noted that the need for values to be copied into or out of private variables occurs infrequently in practice.

Now, consider a `while` loop for which the cross-iteration dependences cannot be analyzed at compile-time. Instead of executing the `while` loop sequentially, the compiler could decide to speculatively execute the loop in parallel as a `doall` (using one of the techniques described in the previous section) and to test at run-time whether there were any cross-iteration dependences in the loop. If the test finds that there were cross-iteration dependences in the loop, then it will be re-executed sequentially. In addition, if it is suspected that some memory-related dependences could be removed by privatization, then the compiler may further elect to privatize those arrays that need it in the speculatively parallelized `while` loop. In order to speculatively parallelize a `while` loop as outlined above we need *an error (hazard) detection method* to test the validity of the speculative parallel execution.

5.1 Detecting Errors in the Parallel Execution

There are essentially two types of errors (hazards) that could occur during the speculative parallel execution of the `while` loop: (i) exceptions and (ii) the presence of cross-iteration dependences in the loop. A simple way to deal with exceptions is to treat them like an invalid parallel execution, i.e., if an exception occurs, abandon the parallel execution, restore the values of any altered program variables, and execute the loop sequentially. Recently, we have proposed a run-time technique, called the *Privatizing* `doall` *test* (*PD test*) [21] and the more powerful *LRPD test* [22], for detecting the presence of cross-iteration dependences in a loop [21]. This test was originally developed to test at run-time whether a `do` loop was fully parallel, i.e., whether it could be executed as a `doall`. However, the PD test can also be adapted to detect cross-iteration dependences in a `while` loop since such a loop is essentially a generalization of a `do` loop, i.e., it is just a `do` loop with an unknown iteration space.

Before discussing how the PD test is used for `while` loops, we first need to briefly describe the types of operations it performs, and the data structures it uses (see [21] for a complete description of the test). The PD test is applied to each shared variable referenced during the loop whose accesses cannot be analyzed at compile-time. For convenience, we discuss the test as applied to a shared array A. Briefly, the test traverses shadow array(s) during the speculative parallel execution using the access pattern of A, and after loop termination performs some final analysis to determine whether there were cross-iteration dependences between the statements referencing A. Separate shadow arrays A_r and A_w are used to keep track A's read and write accesses, respectively, and, if necessary, another shadow array A_p is used for testing whether A was validly privatized. For each access (read or write) to A, some simple computation is performed on the appropriate shadow array. For example, the first time an element of A is written *in an iteration*, the corresponding element in A_w is marked. The analysis performed after loop termination, which determines whether there were any cross-iteration dependences between statements referencing A, involves computations such as counting the total number of marked elements in A_w and determining whether any element is marked in both A_w and A_r. It is important to note that the post-execution analysis is fully parallel, regardless of the nature of the original loop. The time required by the PD test is $O(a/p + \log p)$, where p is the number of processors, and a is the total number of accesses made to A during the loop.

We now discuss how to use the PD test on `while` loops. The general strategy is to combine the PD test (applied on the remainder loop) with the techniques described in Sections 3.1-3.3 for transforming `while` loops into `doall` loops.

If it is known that the parallel execution of the `while` loop will not overshoot, then the shadow variable accesses of the PD test can be inserted directly into the `doall` loop that is created for the `while` loop. When overshooting may occur, a simple solution is to initialy assume that there are no cross-iteration dependences, and execute the loop twice. First, the loop is run in parallel to determine the number of iterations (using one of the methods of Sections 3.1-3.3), and once the number of iterations is known the resulting `do` loop can be speculatively parallelized using the PD test as mentioned above.

In order to avoid executing the parallel version of the `while` loop twice, the PD test can be incorporated directly into the `while` loop methods as follows. Suppose that some shared array A in the `while` loop will be privatized

and tested using the PD Test, and assume that it is known that A is *not live* after the loop. In this case, all writes to the shadow arrays used for the PD Test will be time-stamped (just like all other variables), and for each shadow element we will maintain the minimum iteration that marked it. Everything proceeds as before, except that in the post-execution analysis of the PD test, those marks in the shadow arrays with minimum time-stamps greater than the last valid iteration will be ignored.

If a privatized shared array under test is live after the loop, then the backup method for the privatized array must be more sophisticated. The reason for this is that it is possible for a private variable to be written in more than one iteration of a valid parallel loop. In order to handle this problem, we can keep a time-stamped (by iteration number) trail of all write accesses to the privatized array. If the test passes, the live values need to be copied out: the appropriate value would be the value with the latest time-stamp that was not larger than the last valid iteration number, and could be found in the time-stamped trail of write accesses. Methods for reducing the memory requirements are given in Section 8.

If the termination condition of the `while` loop is dependent (data or control) upon a variable with unknown dependences, then special care must be taken. If it turns out that there are no cross-iteration dependences in the loop, then the techniques mentioned above would work as before. However, if there is a dependence between statements in different iterations accessing some variable, and the termination condition is dependent upon that variable, then some difficulties may arise if the loop is executed in parallel. For example, the last valid iteration of the loop might be incorrectly determined, or, even worse, the termination condition might never be met (an infinite loop). In this situation, the best solution is probably to strip-mine the loop, and to run the PD test on each strip.

6 Transforming Arbitrary `while` Loops for Parallelization

In the previous sections we made the simplifying assumption that the `while` loop had only one recurrence (dispatcher) and proposed methods for parallelizing it. In this section we extend our methods to the case when the loop under consideration contains an arbitrary number of statically detectable recurrences.

After constructing the data dependence graph of the loop, we distribute the initial loop into a loop formed by the hierarchically top level recurrences and a second loop containing the remainder of the statements. The recurrence(s) extracted are then parallelized (if possible) with the methods described in Section 3. The second loop is then treated in the same way, with the difference that we have already computed its data dependence graph. This means that, for analysis purposes, we now extract the top level recurrence and attempt to parallelize it. Essentially this method amounts to a recursive application of the techniques described in the previous sections. This process stops after all recurrences have been extracted from the original `while` loop. The remaining loop may take several forms:

- A fully parallel loop
- A sequential loop whose dependence structure is not of the form of any recurrence detectable by the compiler
- A loop whose access pattern cannot be analyzed statically

Once this loop distribution is completed we will fuse all the contiguous loops of the same type, i.e., sequential, parallel. The resulting sequence of loops will be combined (fused) according to the following criteria:

- Maximize granularity
- Maximize the code to be executed in parallel
- Balance the overhead of parallelization with the potential speedup, especially in the case when pseudo-parallel code is generated (see Section 3.3)

The method proceeds bottom–up (based on the data dependence graph) and analyzes the nature of the loops. If the first loop is sequential, we fuse it with all following contiguous sequential loops. When the first parallelizable loop is found, we generate a distinct, new loop to which all next contiguous parallel loops are fused. If a new sequential loop is encountered, it is fused to the existing block. The decision of whether to generate parallel code for a parallelizable loop (using one of the methods of Section 3.3) depends on its potential parallel performance. In particular, if the overhead of parallelization is not offset by the parallel execution, then sequential code should be generated and fused to the immediately preceding sequential block, if any. In particular, if the overhead of parallelization is not offset by the

parallel execution, then sequential code should be generated and fused to the immediately preceding sequential block, if any. In the end this algorithm produces a parallel loop if enough parallelism is available followed by a sequential loop if any. We can exploit the availability of a dependence graph by scheduling the sequential loops in a `doacross` fashion.

We remark that fusing associative recurrences evaluated by parallel prefix computations must be done carefully if there is data flow between the recurrences. Similarly, loops parallelized with the PD test should be fused with care – if at all – to loops that they dominate in the data dependence graph since the cost of a failed test will be increased for the resulted loop.

7 Predicting Performance

Although it is not strictly necessary for the compiler to perform any cost/performance analysis, the overall usefulness of techniques for parallelizing `while` loops will be enhanced if their overhead can be avoided when they are unlikely to yield speedups. The main factors that the compiler should consider when deciding whether to parallelize a `while` loop are: the probability that the iterations of the loop are independent, the speedups that could be obtained using the techniques, and any potential slowdown that might result. In order to perform this analysis and to predict the parallelism of the loop, the compiler should use both static analysis and run-time statistics (collected on previous executions of the loop); in addition, directives about the parallelism of the loop might prove useful.

In this paragraph we will evaluate the ratio between the ideal speedup obtainable by hand parallelization and the real speedup obtained using the techniques presented in this paper. The ratio computed here is independent of the actual number of iterations in the `while` loop. If an absolute execution time needs to be predicted then an estimation of the loop counts has to be made. Given a `while` loop L, the *ideal speedup*, Sp_{id}, of L is the ratio between its sequential execution time, T_{seq}, and its ideal parallel execution time, T_{ipar} (i.e., the time required by an optimal parallel execution of the loop). *If there is not enough parallelism available in the loop, i.e., Sp_{id} is small, then it should not be parallelized.* For convenience, we partition T_{seq} into T_{rem} and T_{rec}, where T_{rec} is the time to compute the entire dispatching recurrence, and T_{rem} is the time spent in the remainder of the loop. In general, $T_{ipar} = T_{rem}/p + T_{rec}$, where p is number of processors, i.e., the recurrence is evaluated sequentially and all other work is done in parallel. However, if the dispatcher is an induction or an associative recurrence, then the dispatching recurrence can be evaluated in parallel as well, i.e., $T_{ipar} = (T_{rem} + T_{rec})/p$, with an additional term of $\log p$ in the case of the associative recurrence. An example in which there is not enough parallelism available in the loop to justify its parallelization is when the dispatcher is a more complex recurrence and $T_{rem} < T_{rec}$, i.e., the loop essentially consists of evaluating the dispatcher, which must be done sequentially.

When our run-time techniques are applied, the *attainable speedup*, Sp_{at}, will be reduced by the overhead of the methods. This overhead can be divided into T_b, T_d, and T_a, representing the overhead incurred before, during or after the parallel execution of the loop, respectively.

$$Sp_{id} = \frac{T_{rem} + T_{rec}}{T_{ipar}} \qquad\qquad Sp_{at} = \frac{T_{rem} + T_{rec}}{T_{ipar} + T_b + T_d + T_a}$$

The overhead incurred before the parallel execution, T_b, represents the cost of any checkpointing needed to maintain the original values so that iterations can be undone, or, if the PD test is applied, and fails, so that the loop can be re-executed sequentially. The overhead during the parallel execution, T_d,includes the cost of time-stamping so that invalid iterations can be undone, and the accesses to the shadow variables if the PD test is applied. The overhead after the parallel execution, T_a, includes the time needed to undo any iterations found to be invalid, and the post-execution analysis of the PD test, if it is applied.

Assuming that the PD test is not applied, let a denote the number of accesses made during the parallel execution of the loop, excluding those inserted by our techniques. Since all operations contributing to the overhead incurred before and after the parallel execution can be executed in parallel, in the worst case we have $T_b \approx T_a = O(\frac{a}{p})$. The number of operations of the overhead incurred during the parallel execution is also proportional to a, but the extent to which these operations can be parallelized is dependent upon Sp_{id}, the maximum parallelism available in the original loop, i.e., $T_d = O(\frac{a}{Sp_{id}})$. However, the worst case for the obtained speedup is when $Sp_{id} \approx p$, so that $Sp_{at} = O(\frac{1}{4}Sp_{id})$. If the PD test *is applied*, then in the worst case $Sp_{at} = O(\frac{1}{5}Sp_{id})$ because the post execution analysis might add

another term of $\frac{a}{p}$ to T_a. Note that 20-25% of the ideal speedup could be an excellent performance — especially when compared to the alternative of sequential execution.

We remark that *in many cases the expected speedup of our techniques will be larger than the worst case estimates given above.* For example, if it is known that variables are not live after the loop, then often the overhead of time-stamping and restoring can be avoided. A case when all of the overhead can be avoided is when it is known that no overshooting will occur in the parallel execution (e.g., a linked list traversal with a RI termination condition). In this case, we would have $Sp_{id} = Sp_{at} = T_{work}/p + T_{rec}$. Another case in which our speedup estimates might prove overly conservative is when the iterations are initiated in order. In this case, on the average we would expect to undo $p/2$ iterations – not all of them as in our worst case estimate of T_a.

If the PD test is used on the loop, and it fails, then a *slowdown* will be incurred, i.e., the total execution time includes both the failed parallelization attempt and the sequential re-execution of the loop. Since, when using the PD test, in the worst case the failed attempt could require time $O(\frac{5}{p}T_{seq})$, the total execution time could be $O(T_{seq} + \frac{5}{p}T_{seq})$. Therefore, the slowdown incurred is proportional to T_{seq}/p.

From our analysis above, it is clear that as long as there is enough parallelism available in the loop, a speedup can be obtained by parallelizing it. This is true even if the dependence relations in the loop are not known, and the PD test must be applied — the speedup that would be obtained is significant and the potential slowdown is small. However, if it is known a priori with a high degree of confidence that the loop is not parallel, then parallelization should probably not be attempted. There are essentially two cases in which the maximum available parallelism in the loop may not be sufficient to justify parallelization. The first case is is when the dispatcher is not an induction and $T_{work} < T_{rec}$, i.e., the loop essentially consists of evaluating the dispatcher, which must be done sequentially. The second case in which speedups might not be obtained is if there are not enough iterations in the loop. For a `do` loop, it is sometimes possible to determine at compile-time whether there are enough iterations to justify parallelization. Unfortunately, this is not true for `while` loops. However, in this case the compiler could predict the number of iterations using branch statistics, where the branch is on the termination condition of the `while` loop. Although the application is different, this in not a new idea since branch speculation has been used effectively in superscalar compilers [18, 23, 24]. Since branch statistics have already been collected for many benchmarks, these collection mechanisms are available.

8 Strategies for Applying the Techniques

In the previous section we discussed the speedups and potential slowdowns that can be expected when using our techniques for parallelizing `while` loops. In this section, we discuss some strategies that might be used to help bias the results in our favor, i.e., to help to insure or improve speedups and reduce the possibility of slowdowns.

Thus far we have not addressed the fact that our techniques might cause an increase in the working set size which could lead to performance degradation. Motivated by this fact, most of the methods discussed in this section are targeted at managing the size of the working set.

8.1 Statistics Enhanced Strip-Mining

The additional memory required by our techniques is for the checkpointing prior to the parallel execution, and the time-stamps made during the parallel execution for the write accesses so that work done by iterations later found to be invalid can be undone.

Perhaps the simplest way to reduce the memory requirements is the *strip-mine* the loop, as discussed in Section 4. In this case, time-stamps would only need to be maintained for values written during the current strip. Therefore, only $O(sa)$ memory would be needed for these values, where s is the size of the strip and a is the number of writes per iteration. However, strip-mining introduces global synchronization points and might potentially reduce significantly the amount of obtainable parallelism.

Suppose that, as discussed in Section 7, the compiler has supplied an estimate n_i of the number of iterations in the loop. If this is a good estimate, then the time-stamps for values written in iterations smaller than n_i are not likely to be needed since these iterations will likely be valid. Therefore, in this case, a good strategy might be to only time-stamp the values written in iterations larger than n_i', for some value n_i' close to, but less than, n_i, e.g., $n_i' = .9n_i$. The value n_i' could be chosen based on the degree of confidence placed on the compiler's estimate of n_i, e.g., if the confidence in n_i is about x%, then n_i' is selected to be about x% of n_i.

8.2 Resource Controlled Self-Scheduling

A way in which the memory requirements could be reduced without introducing rigid synchronization points is to maintain a *sliding window* of some predetermined size w: at any given time, the difference between the minimum iteration l that has not been completely executed and the maximum iteration h that has been, or is currently being, executed is at most w, i.e., iterations 1 through $l-1$ have been completely executed, and $h - l \leq w$. Similar to the strip mining solution, this method would bound the memory needed to maintain the time-stamps by the product of window size w and the number of write accesses performed in an iteration.

Now, suppose that the window size is dynamically determined at the application level based on the current memory usage: the window size is increased if more memory can be used without degrading performance, and is decreased if less memory should be used to improve performance. The window size can be dynamically adjusted by the program itself, since the program could easily monitor how much memory is used by its data structures. The working set size of the application can be user pre-set, or dynamically adjusted according to system usage information available through system calls. Note that we are suggesting that the application monitor its own memory usage and dynamically adjust its actions accordingly, which is is different from operating system monitors that watch such things as network traffic, i/o requests, or paging activity.

8.3 The One Processor/$(p-1)$ Processor Solution

As a final remark, we note a method that can be used to minimize the risks of parallelizing a `while` loop: one processor executes the loop sequentially, and the rest of the processors execute the loop in parallel. Of course, the sequential and the parallel executions would need separate copies of the output data for the loop. As long as the cost of creating these copies is not too great, this technique should maximize the potential gains attainable from parallel execution, while, at the same time, minimizing the costs.

9 Experimental Results

In this section we present experimental results obtained on a modestly parallel machine with 8 processors (Alliant FX/80 [1]) using a Fortran implementation of our methods. It should be pointed out that our results scale with the number of processors and the data size and that they should be extrapolated for MPPs, the actual target of our methods.

We considered five `while` loops that could not be parallelized by any compiler available to us; two loops are from the PERFECT Benchmarks [3], two loops are from MA28, a sparse non-symmetric linear solver [6], and one loop is extracted from MCSPARSE, a parallel version of a non-symmetric sparse linear systems solver [7, 8]. Our results are summarized in Table 2. For each method applied to a loop, we give the speedup that was obtained, and, mention whether backups and time-stamping were necessary. Whenever necessary, we performed a simple preventive backup of the variables potentially written in the loop. In some cases, the cost of saving/restoring might be significantly reduced by using another strategy. In addition to the summary of results given in Table 2, we show in Figures 6 through 14 the speedup measured for each loop as a function of the number of processors used.

Overall, our results show that significant speedups can be obtained by parallelizing `while` loops using our methods. We now make a few remarks about individual loops for which Table 2 does not give complete information.

Loop 40 in subroutine LOAD from SPICE loads the device models for capacitors. Since our interest is in measuring the performance of our linked list traversal techniques, the run-time overhead associated with run-time dependence testing has not been included in the reported results.

Even though the body in Loop 40 does little work, we obtained a very good speedup (Figure 6). Note that although each processor traversed the entire linked list, the General-3 method significantly outperformed the General-1 method, in which the processors cooperatively traversed the list (by placing the $next()$ operation in a critical section). Since the structure of Loop 40 is identical to those for the evaluation of transistor models (subroutines BJT and MOSFET), the same parallelization techniques can also be used on these loops. We remark that approximately 40% of the sequential execution time of SPICE is spent in subroutine LOAD, which calls subroutines BJT and MOSFET.

[2] All benchmarks are from the PERFECT Benchmark Suite, with the exception of MCSPARSE.

[3] The final paper will include experimental results for all loops on both machines.

Benchmark[2] Subroutine Loop	Experimental Results			Description of Loop
	Technique	Input	Speedup	
SPICE LOAD Loop 40	General-1 (locks)	N/A	2.9	traverses linked list terminated by a NULL pointer loop counter: *recurrence* termination condition: *RI* no backups or time-stamps
	General-3 (no locks)	N/A	4.9	
TRACK FPTRAK Loop 300	Induction-1	N/A	5.8	accesses array indexed by run-time computed subscript array loop counter: *induction* termination condition: *RV* backups and time-stamps
MCSPARSE DFACT Loop 500	Induction-1	gematt11	7.0	processes an array loop counter: *induction* termination condition: *RV* no backups and no time-stamps
		gematt12	6.8	
		orsreg1	4.8	
		saylr4	5.7	
MA28 MA30AD Loop 270	Induction-1 *and* General-3 (no locks)	gematt11	3.5	processes an array loop counter: *induction* termination condition: *RV* backups and time-stamps
		gematt12	3.4	
		orsreg1	5.3	
MA28 MA30AD Loop 320	Induction-1 *and* General-3 (no locks)	gematt11	4.8	processes an array loop counter: *induction* termination condition: *RV* backups and time-stamps
		gematt12	4.5	
		orsreg1	2.8	

Table 2: Summary of Experimental Results.

Loop 300 in subroutine FPTRAK from TRACK is a `do` loop with a conditional exit which is taken if an error condition is detected. The speedup obtained is shown in Figure 7. For this loop we also show the ideal speedup, which was obtained from a hand parallelized version of the loop.

Loops 270 and 320 in subroutine MA30AD from MA28 cooperatively search for a pivot. Since MA28 is a sequential program, any parallelization must guarantee sequential consistency. In order to accomplish this we time-stamped the pivots found during the parallel execution. Then, after loop termination, we found the pivot with minimum cost by performing a time-stamp ordered reduction operation (minimum) on the (privatized) pivots selected by each processor. For each input set, the speedups for both Loop 270 and 320 are shown on the same graph (see Figures 12 through 14). We remark that the speedups shown for the loops from MA28 are not as big as for the other programs. This is largely due to the fact that there was less available parallelism in these loops.

`While-Doany`

MCSPARSE is, as mentioned before, a sparse solver that has been manually programmed as a parallel code. Loop 500 in subroutine DFACT from MCSPARSE searches for a pivot in a non-deterministic manner. In other words the program is designed to be insensitive to the order in which the columns and rows of the matrix are searched for the pivot. Originally, only the row search was parallelized by applying a technique equivalent to a `doany` construct [29], leaving the traversal of columns in a sequential `while` loop. We fused the two loops, effectively implementing a new `While Doany` parallel construct. Through this technique we were able to parallelize the pivot search across the whole matrix. Since the order of the searching iterations is not important, we did not need to perform backups or maintain time-stamps for back-tracking, even though the termination condition is RV and we do overshoot. We report speedups for four different input data sets from large sized Harwell-Boeing matrices (see Figures 8 through 11). Note that the available parallelism, and therefore our obtained speedup, is strongly dependent on the data input.

10 Related Work

We can find in the literature several efforts in improving the performance of the while loop execution. In [24] the authors have proposed some methods for achieving vector-like performance on multiple issue pipelined machines. They do not try to address the problem for large multiprocessors.

Some techniques for solving certain types of recurrences in parallel were proposed by Harrison in [27] for Lisp-like languages. His main goal was to parallelize list operations (e.g., traversing a linked lists). Generally, his methods assume that the terminator is RI and it is known that there are no cross-iteration dependences in the loop. In the context of his proposed framework ([9]), lists consist of linked chunks of contiguously allocated memory locations, and each chunk has a header that stores the number of memory locations in that chunk. In this way the evaluation of the dispatcher (i.e., the traversal of the list) can be optimized by using a *sequential* prefix computation (on the chunks) to assign portions of the recurrence (chunks) to processors for parallel evaluation. We note that this optimization requires the dynamic memory allocation scheme proposed by the author (in which list elements are allocated contiguously). Therefore, for languages such as FORTRAN which rely mainly on static memory allocation (i.e., each list element is contained in a separate chunk), this method could not be used to parallelize the evaluation of the dispatcher, i.e., it would degenerate to the naive loop distribution method mentioned for general recurrences in Section 3.3. In fact, the author mentions that if the chunk sizes become too small, then the result might be an "inefficient restructured version of the loop that contains too little parallelism to recover the expense [invested]" [9]. We note that when the entire list resides in a single chunk (i.e., an array), then this method is equivalent to the method we describe in Section 3.2 for associative recurrences, i.e., loop distribution together with a parallel prefix computation to evaluate the dispatcher in parallel.

The only previous work of which we are aware (except some early work by [28]) for parallelizing while loops in languages such as FORTRAN for multiprocessors is due to Wu and Lewis [30]. One method they propose is to pipeline the loop by executing it in doacross fashion, and to enforce any cross-iteration data dependences with explicit synchronization operations. When the terminator is RI and it is known that there are no cross-iteration data dependences in the loop, they suggest using the naive form of loop distribution mentioned in Section 3.3 (also implicit in [9]), i.e., first a *sequential* while loop evaluates the dispatcher and stores its values in an array, and then the loop iterations are performed in parallel using this array.

For the case of RV termination conditions no methods have been proposed in the past. Also, the problem of testing for cross-iteration data dependences has not been addressed before.

11 Conclusion

In this paper, we have shown that lack of knowledge about the iteration space of a loop does not preclude parallelization. We have demonstrated this by giving techniques for concurrently executing while loops and do loops with conditional exits. Our methods can even be used to obtain significant speedups for loops that involve linked list traversals without using global synchronization or explicitly sequential code — something that has not been done before. This is an important problem since linked list traversals arise frequently in loops with irregular access patterns, such as sparse matrix computations. In many cases, these methods have no associated overhead or side effects. Our techniques can be applied even when the dependence relations between the iterations of the loop cannot be analyzed at compile-time. In this case, an efficient run-time test for cross-iteration dependences is inserted into the parallel version of the loop, and the outcome of the test determines whether the parallel execution was valid, or if the loop must be re-executed sequentially.

We feel our framework for while loop parallelization represents a step toward providing while loop counterparts for the existing constructs for parallel execution of do loops: While-Doall, While-Doacross, and While-Doany. Based on our experience, these new constructs would be useful extensions to present parallel languages.

Our experimental results show that our techniques yield significant speedups for real loops — even on a modestly parallel machine like the Alliant FX/80. However, we believe that the true significance of these methods will be the increase in real speedup obtainable on massively parallel processors (MPPs). The performance gain (speedup) from our techniques ranges from a minimum of $20 - 25\%$ of the ideal speedup to nearly 100% of the ideal speedup. If the target architecture is an MPP with *hundreds* or, in the future *thousands*, of processors, then even the minimum expected speedup could easily reach into the hundreds. We have also shown that the potential payoffs remain large

when the cross-iteration dependences are analyzed at run-time. Therefore, our conclusion is that as long as there is enough parallelism available in the `while` loop, our techniques should be applied.

To bias the results even more in our favor, we would like to avoid parallelizing loops that do not have enough available parallelism. For this reason it would be useful to estimate the number of iterations in the loop using information such as branch statistics – data which can easily be obtained for any program. Also, in order to decrease the probability of attempting to parallelize a loop that is in fact sequential, our methods should make use of run-time collected information about the parallel/not parallel nature of the loop. In all cases, specialized hardware features could greatly reduce the overhead introduced by the methods.

References

[1] Alliant Computer Systems Corporation, 42 Nagog Park, Acton, Massachusetts 01720. *FX/Series Architecture Manual*, 1986. Part Number: 300-00001-B.

[2] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer. Boston, MA., 1988.

[3] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[4] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, pages 71–88, 1989.

[5] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical parallel band triangular solvers. *ACM Transaction on Mathematical Software*, 4(1):270–277, September 1978.

[6] I. S. Duff. Ma28– a set of fortran subroutines for sparse unsymmetric linear equations. Technical Report AERE R8730, HMSO, London, 1977.

[7] K. Gallivan, B. Marsolf, and H. Wijshoff. A large-grain parallel sparse system solver. In *Proc. Fourth SIAM Conf. on Parallel Proc. for Scient. Comp.*, pages 23–28, Chicago, IL, 1989.

[8] K. A. Gallivan, B. A. Marsolf, and H. A. G. Wijshoff. MCSPARSE: A parallel sparse unsymmetric linear system solver. Technical Report 1142, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1991.

[9] W. L. Harrison, III. Compiling lisp for evaluation on a tightly coupled multiprocessor. Technical Report 565, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., March, 1986.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantatative Approach*. Morgan Kauffman, San Mateo,CA, 1990.

[11] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing*, pages 407–416, November 1990.

[12] C. Kruskal. Efficient parallel algorithms for graph problems. August 1985.

[13] C. Kruskal. Efficient parallel algorithms for graph problems. pages 869–876, August 1986.

[14] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[15] R. Ladner and M. Fisher. Parallel prefix computation. *J. ACM*, pages 831–838, 1980.

[16] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[17] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 313–322, 1992.

[18] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[19] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. In *Proceedings 5th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[20] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, December 1986.

[21] L. Rauchwerger and D. Padua. The privatizing doall test: A run-time technique for doall loop identifi cation and array privatization. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 33–43, July 1994.

[22] Lawrence Rauchwerger and David A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, June 1995.

[23] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 344–354, May 1990.

[24] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of loops with exits on pipelined architectures. In *Supercomputing*, November 1990.

[25] P. Tu and D. Padua. Array privatization for shared and distributed memory machines. In *Proceedings 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, September 1992.

[26] P. Tu and D. Padua. Automatic array privatization. In *Proceedings 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[27] III W. L. Harrison. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. Technical Report 860, Univ. of Illinois at Urbana-Champaign, Center for Superco mputing Res. & Dev., Feb., 1989.

[28] M. Wolfe. *Optimizing Compilers for Supercomputers*. The MIT Press, Boston, MA, 1989.

[29] M. Wolfe. Doany: Not just another parallel loop. In *Proceedings 5th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 757. Springer-Verlag, 1992.

[30] Youfeng Wu and Ted G. Lewis. Parallelizing while loops. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, Software, pages 1–8, 1990.

[31] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, New York, 1991.

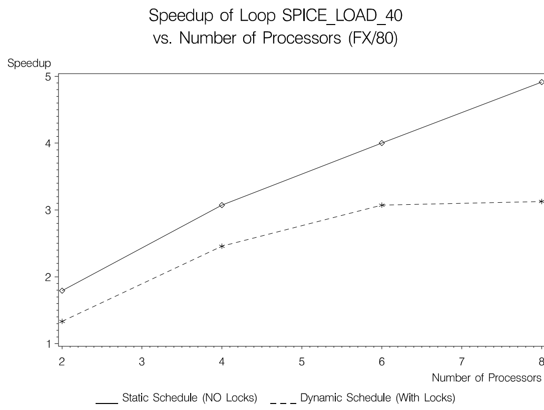

Figure 6:

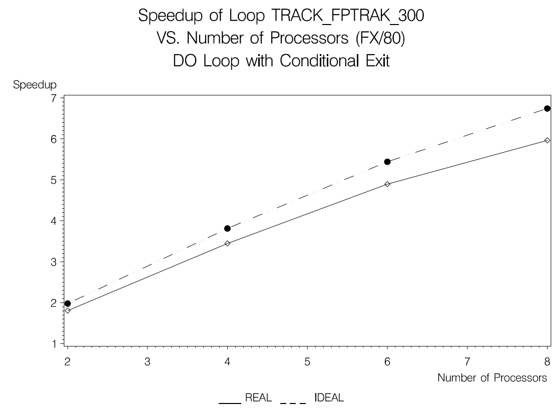

Figure 7:

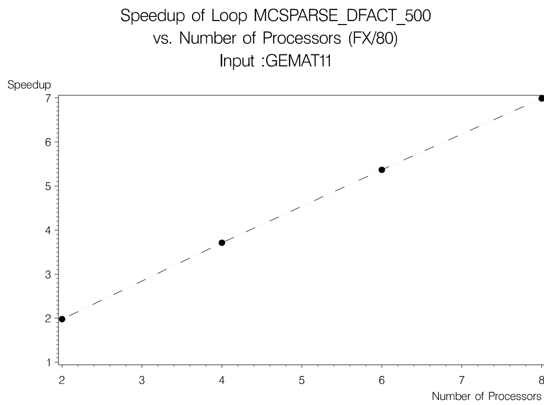

Figure 8:

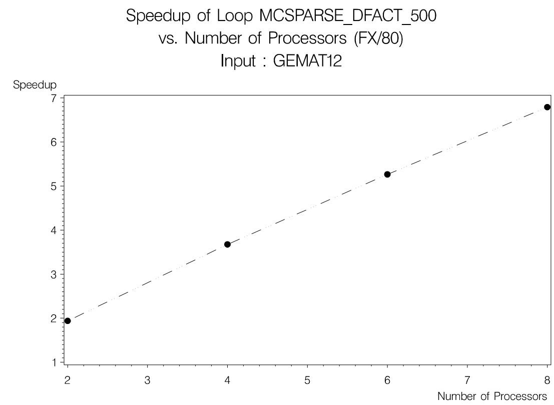

Figure 9:

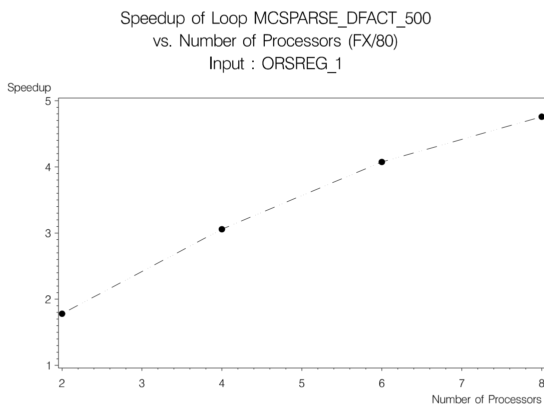

Figure 10:

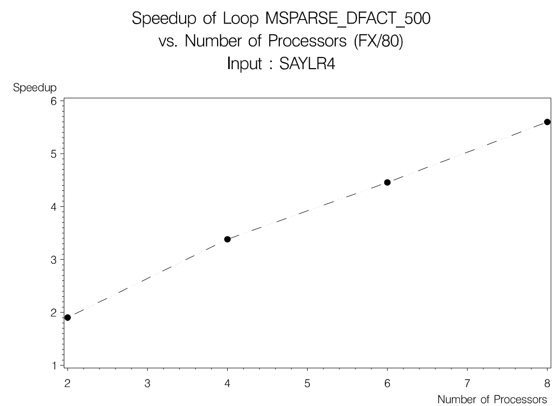

Figure 11:

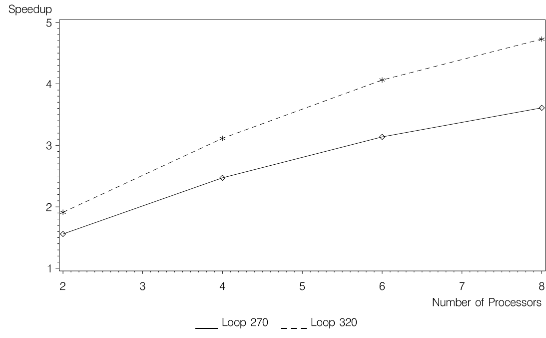

Figure 12:

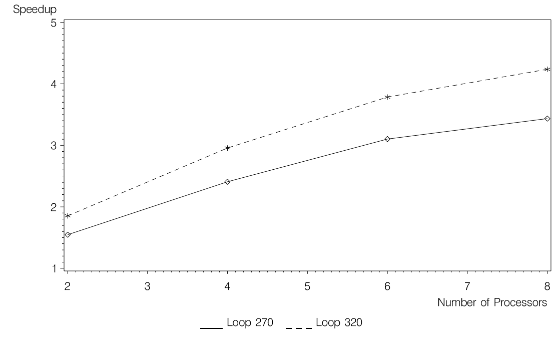

Figure 13:

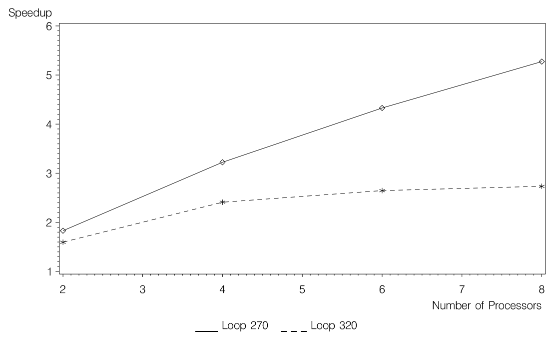

Figure 14: