OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Parallelizing WHILE Loops

Youfeng Wu
T.G. Lewis
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

87-10-1

Parallelizing WHILE loops

Youfeng Wu
T.G. Lewis

Oregon State University
Corvallis, OR 97331
(503)754-3273

Abstract Two methods for parallelizing WHILE loops are presented. The first method converts a WHILE loop into a FORALL construct, and the second method pipelines a WHILE loop. Each of the methods is based on a transformation that makes explicit the loop counting. Also, we propose two parallel WHILE constructs.

1. INTRODUCTION

Automatically parallelizing a sequential program is desirable not only because of the need to restructure large amounts of existing sequential software, but also because programmers become less competent at optimization as computer complexity increases ([LAMPORT-75], [LUBECK-85]). It has long been recognized that designing parallel programs imposes a heavy burden on the programmers -- it can be extremely tedious for a programmer to ensure that the components of a parallel program do not interact unexpectedly. For example, to decide whether to use a FORALL i:= 1 TO n DO S ENDFORALL parallel construct or a FOR i:= 1 TO n DO S ENDFOR sequential construct, a programmer has to determine whether or not the executions of S in different iterations will interfere with one another. To discover such an interference, a programmer may have to mentally unwind the loop and imagine the many possibilities of interactions between different iterations. People tend to ignore such considerations, while a computer tool can analyze such loops faster and with greater precision.

The paradox is that there are cases beyond the capability of an automatic tool. One instance is very complex array subscripts, pointers. When this happens, human insight might be the only way to decide parallelism. Also, if an algorithm has already been designed, the compiler's analysis can be saved. In either case, an explicit parallelism construct is useful.

There are rather sophisticated techniques for parallelizing FORTRAN-like DO loops (for example, see [PADUA-86]). Since the DO loops are equivalent to the FOR loops of PASCAL, MODULA-2, etc., we do not distinguish them and refer to them all as FOR loops. Also, the parallel FORALL construct has been included in almost every parallel language [KARP-88]. However, to our knowledge, no technique for parallelizing WHILE loops has been proposed in literature (although several compilers, such as the Alliant FORTRAN compiler, actually performs certain while loop parallelization). In general there is the need for studying WHILE loop parallelism because most of the modern languages like Pascal, C, Modula-2, etc. have the WHILE construct, and the WHILE loop is a good source of parallelism.

Assume a WHILE loop has the following format:

```
(*)     WHILE b(D) DO
            S(D);
        ENDWHILE;
```

where D is a set of variables and S is performed on D only when *b* is true and repeats until *b* becomes false.

A straightforward method to parallelize a WHILE loop is to rewrite the WHILE loop as a FOR loop (with exit), and parallelizing the FOR loop using the known techniques. In this method, the given loop is converted to

```
FOR i:=1 TO N DO
    IF b(D) THEN S(D) ELSE exit ENDIF
ENDFOR;
```

where N is the estimated upper bound of the WHILE loop based on "worst case performance" ([LEE-85]). This method suffers from at least three drawbacks. First, it does not work in general because it is not always possible to estimate the upper bound N. Secondly, the estimation may be too conservative such that the parallelism obtained may be overweighted by the extra overhead introduced by the unreasonable N. Third, the method creates an IF construct and exit inside the FOR loop, which could make the FOR loop hard to be parallelized ([LEE-85]).

In this paper, we present two general methods to parallelize WHILE loops. The first method also converts a WHILE loop into a FOR construct, although we provide a method to determine the exact upper bound and introduce no IF construct. The second method pipelines a WHILE loop. Also, we propose parallel constructs to explicitly express parallelism in a WHILE loop like form.

2. DEFINITIONS AND ASSUMPTIONS

The concepts of *iteration vector* and *loop carried dependence* of [ALLEN-83] are introduced here.

Iteration vector. For a k-level nested loop (either FOR or WHILE), a vector (i1,i2,...ik) is called an iteration vector if the loop body can be executed when the j'th level loop is in the ij'th iteration for all j = 1,..,k.

We use S(I) to denote the execution of statement S during the iteration I= (i1,i2,...ik).

Loop carried dependence. Statements S1 and S2 have (true) loop carried dependence iff there exist I = (i1,i2,...ik) and $\mathcal{I}$ = (ι1, ι2,... ιk), I > $\mathcal{I}$, S1(I) uses the variables updated by S2($\mathcal{I}$). Further, S1 and S2 have level j loop carried dependence iff i1= ι1, i2 = ι2, ..., ij-1 = ιj-1 , ij ≠ ιj.

Example 2.1. Consider the following FOR loop,

```
FOR i:=1 TO 100 DO
    FOR j:=1 TO 100 DO
        S: x[i,j] := comp(x[i-2, j+3]);
    ENDFOR;
ENDFOR;
```

On iteration (i, j), S((i, j)) reads x[i-2, j+3]. The same memory location is written on iteration (i-2, j+3). Since iteration vectors (i, j) > (i-2, j+3), we conclude that S has a loop carried dependence on itself.

There are various techniques ([WOLFE-82] and [ALLEN-83]) that can be used to break up loop carried dependencies (especially those false dependencies: anti-dependence and output-dependence [KUCK-81]). In this paper, we assume that these techniques will be applied automatically whenever appropriate and the loop carried dependencies we addressed in the following are those that cannot be removed. Also, our context of loop parallelizability is constrained to the possibility of the parallel executions of all of the loop iterations, and we assume that any partial parallelism of a loop has been properly handled by other techniques such as expression tree height compression and loop splitting (see [PADUA-86]).

Based on these definitions and assumptions, we can state the following as a theorem, without giving its proof.

Theorem 1. A FOR loop at level j is parallelizable iff the loop body does not have level j loop carried dependence.

In the following, we consider non-nested loops. For simplicity, we use S(i) to indicate the execution of statement S in iteration i, instead of S((i)). The results can be easily extended to nested loops.

3. CHARACTERISTICS OF WHILE LOOPS

Trivially, the loop condition *b* in a while loop of form (*) must have true dependence on loop body S. Otherwise the condition will never change and the loop is either a empty loop of an infinite loop. Also, each iteration has control dependence on the loop condition. However, this two facts do not imply that S has loop carried dependence on itself. For example, in the following loop,

```
x := 1;
WHILE x > 0 DO x := -1 ENDWHILE;
```

the loop condition has true dependence on loop body, and the loop body has control dependence on the loop condition. However, the loop body has no loop carried dependence on itself.

In the following, we show that every iteration S(i) is truly dependent on S(i-1), if the loop is finite and executes more than once.

PROPOSITION 3.1. A WHILE loop body must have loop carried dependence on itself, or the WHILE loop is either an empty loop, being executed only once, or an infinite loop.

PROOF. Assume U_i be the set of variables that are actually used in iteration i. Since not all variables in D are used in every iterations, U_i may change from iteration to iteration. For the WHILE loop in (*), each iteration s(i) can be thought as performing two functions: r that selects a subset variables U_i from D, and f that maps U_i together with the values in these variables $V_i(U_i)$ to another set of variables M_i with new values $V_i(M_i)$:

$$S(i): U_i = r(U'_i, V_i(U'_i)), (M_i, V_i(M_i)) = f(U_i, V_i(U_i)),$$

where U'_i is a subset of U_i. The WHILE loop is neither an empty loop nor an infinite loop and is executed more than one iteration means that the condition *b* is true S(i-1) and is evaluated to false by S(i), for some i > 1. This implies that $(M_i, V_i(M_i)) \neq (M_{i-1}, V_i(M_{i-1}))$, since otherwise *b* will remain unchanged. This in turn requires $(U_{i-1}, V_{i-1}(U_{i-1})) \neq (U_i, V_i(U_i))$. If S(i) does not use the variables modified by

S(i-1), namely if M_i « U_i = F, then the selection function r in iteration i will select the same set of variables as in iteration i-1, so $U_i = U_{i-1}$. From $(U_{i-1}, V_{i-1}(U_{i-1})) \neq (U_i, V_i(U_i))$ and $U_{i-1} = U_i$, we have $V_{i-1}(U_i) \neq V_i(U_i)$. This can happen only if M_{i-1} « $U_i \neq$ F, a contradiction.
Q.E.D.

In the following, we will call a finite WHILE loop that can iterate more than once a **normal** WHILE loop. Obviously, any WHILE loop appearing in a meaningful program should be a normal WHILE loop.

To convince ourselves that there is potential parallelism in a WHILE loop, even with the establishment of the above proposition, we consider the following WHILE loop:

```
WHILE i<= n DO x[i]:= comp(i); i:=i+1 ENDWHILE;
```

where comp(i) is a function free of side effects. This WHILE loop is trivially equivalent to the following FOR loop:

```
FOR i:=1 TO n DO x[i]:= comp(i) ENDFOR;
```

and can be parallelized as

```
FORALL i:=1 TO n DO x[i]:= comp(i) ENDFORALL;
```

In summary, any normal WHILE loop has loop carried dependence, and thus should not be parallelizable according to Theorem 1 on parallelizing FOR loops. However, there do exist parallelizable WHILE loops. This phenomenon can be explained by the fact that the loop carried dependency in a WHILE loop is often caused by implicit loop counting, which is explicit in a FOR loop. If we can make the loop counting of a WHILE loop explicit, we can discover the parallelism inside a WHILE loop. This is discussed in the next section.

4. TRANSFORMATIONS OF WHILE LOOPS

We are interested in transforming a WHILE loop to an equivalent form in which its loop counting is explicit.

The first form is that in which the loop counting statements (LC) are at the end of the loop body as in (1) below, where D A » B and A « B = F.

(1)

```
D := D_0;
WHILE b(A) DO
      RS: B := h(D);
      LC: A := g(A);
ENDWHILE;
```

The second form is that in which the loop counting statements (LC) are in the beginning of the loop body, as in (2).

(2)

```
D := D_0;
WHILE b(A) DO
      LC: A := g(A);
      RS: B := h(D);
ENDWHILE;
```

In both form (1) and form (2), we require that A « B = F. This ensures that LC does not depend on RS, though RS is allowed to be dependent on LC. So, when RS is removed, the loop will iterate the same number of times as the original loop.

Trivially, a WHILE loop of form (*) can be transformed into form (1) (e.g. by assuming A=D). In general, we can transform a WHILE loop such that RS is not empty as follows:

TRANSFORMATION 1. Let A be the variables used by *b* and A' œ D be the duplication of A. Then RS consists of the assignment A' := A and those statements in S that are not related to the change of A with all occurrences of A replaced by A'. Similarly, LC consists of all of the statements of S that contribute to the change of A. That is,

```
RS:  A' := A;  S(A -> A') except the statements that are
     only used for updating A';
LC:  the statements in S that solely update A;
```

For this transformation, when removing RS, the WHILE loop will iterate the same number of times as the original loop because LC changes A the same way as the original loop, and RS does not change A at all.

Example 4.1. The following WHILE loop

```
WHILE i <= n DO
     x[i]:= comp(i);
     IF i < 5 THEN i:=i+1 ELSE i := i + 2 ENDIF;
     y[i] := comp(i)
ENDWHILE;
```

can be transformed into the form (1) WHILE loop as below:

```
WHILE i <= n DO
RS:  i' := i;  x[i']:= comp(i');
     IF i' < 5 THEN i':= i'+1 ELSE i' := i' + 2 ENDIF;
     y[i'] := comp(i');
LC:  IF i < 5 THEN i:=i+1 ELSE i := i + 2 ENDIF;
ENDWHILE;
```

Similarly, a WHILE loop of form (*) can be transformed into the form (2) as follows.

TRANSFORMATION 2. First, loop (*) is transformed to form (1) using transformation 1, and then LC is moved to before RS and all of the occurrences of variables in A are replaced by $g^{-1}(A)$.

Example 4.2, the WHILE loop in example 4.1, can be transformed to the following form (2) WHILE loop:

```
WHILE i <= n DO
LC:  IF i < 5 THEN i := i +1 ELSE i := i + 2 ENDIF;
RS:  IF i <= 5 THEN i":=i-1 ELSE i" := i - 2 ENDIF;
     i' := i";  x[i']:= comp(i');
     IF i' < 5 THEN i':= i'+1 ELSE i' := i' + 2 ENDIF;
     y[i'] := comp(i');
ENDWHILE;
```

We make no claim that the transformations proposed here are efficient (obviously some code in the above loop can be removed). In particular, since the transformations duplicate some amount of computation in both LC and RS, an algorithm that would minimize the duplication is an open problem. We expect that the compiler optimization methods like forward substitution and dead code deletion can be explored here ([AHO-87]).

5. CONVERTING WHILE LOOPS TO FOR LOOPS

After transforming a WHILE loop into either form (1) or form (2), we can focus our attention on the parallelizability of RS. The main result in this section is that once a WHILE loop is transformed into either form (1) or form (2), it can be parallelized iff RS does not have loop carried dependency.

The approach used in this section is to convert a WHILE loop into a FOR loop, and use the result on parallelizing FOR loops to establish our result. Note that in a FOR loop, the upper bound is independent of the loop body. In the case of a while loop, the upper bound is often dependent on the execution of the loop body. However, once we have transformed a WHILE loop to either form (1) or (2), the upper bound of the loop count can be determined independently of RS.

PROPOSITION 5.1. Assume A_0 is the initial values of A. A form (1) WHILE loop can be converted to:

```
N = min ( n | b(g^n(A0)) = false )
FOR i:=1 TO N DO
    RS;
    LC;
ENDFOR;
```

PROOF. In a form (1) WHILE loop, A will not be modified by RS. So, if we let A^i be the values of A at the end of iteration i, we have

$$
\begin{aligned}
A^0 &= A_0, \\
A^1 &= g^1(A_0), \\
A^2 &= g^2(A_0), \\
&\ldots\ldots \\
A^N &= g^N(A_0),
\end{aligned}
$$

The total number of iterations of the loop is the smallest n such that

$$b(A^n) = \textbf{false}.$$

This is exactly the definition of N. This tells us that the code in the above FOR loop iterates the same number of times as the given WHILE loop. Also the FOR loop has the same effect on the variables modified by RS and on A.
Q.E.D.

In proposition 5.1, we still have the loop counting statements LC inside the FOR loop. We have to get rid of LC in order to parallelize the loop. We notice that in the FOR loop the purpose of LC is to update A, which might be used by RS and/or by the statements after the loop. As we know that the values of A at the end of iteration i is $A^i = g^i(A_0)$, we can let the RS(i) directly access the values $g^{i-1}(A_0)$ instead of referencing the variables in A. Also, we can let $A^N = g^N(A_0)$ be the values of A used by the statements after the loop.

PROPOSITION 5.2. A form (1) WHILE loop can be converted to the following FOR loop

$N = \min(\, n \mid b(g^n(A_0)) = \textbf{false})$
FOR i:=1 TO N DO
    RS(A-> $g^{i-1}(A_0)$);
ENDFOR;
$A = g^N(A_0)$;

PROOF. From the above discussion.
Q.E.D.

According to Theorem 1, the FOR loop above is parallelizable iff RS does not have loop carried dependence. Consequently, the given WHILE loop is parallelizable iff in the transformed form the N can be found and RS does not have loop carried dependence.

Consider the following form (1) WHILE loop, where $c>0$, $M > i_0$:

```
i := i0;
WHILE i <= M DO
      x[i]:= comp(i);
      i:=i+c;
ENDWHILE;
```

In this example,

$$
\begin{aligned}
g^1(i_0) &= i_0 + c, \\
g^n(i_0) &= i_0 + n*c, \\
b(i) &= (i \le M), \\
N &= \min(\, n \mid b(g^n(i_0)) = \textbf{false}) \\
&= \min(\, n \mid i_0 + n*c > M) \\
&= \min(\, n \mid i_0 + n*c \ge M + 1) \\
&= È(M - i_0 + 1)/c˘. \\
g^N(i_0) &= i_0 + N*c \\
&= i_0 + È(M - i_0 + 1)/c˘ * c.
\end{aligned}
$$

The corresponding FOR loop is:

```
i := i0;
FOR  j :=  1 TO È(M- i0 +1)/c˘ DO
      x [i0  + (j-1) * c]:= comp(i0  + (j-1) * c);
ENDWHILE;
i := i0 +  È(M- i0 +1)/c˘ * c;
```

This FOR loop is immediately parallelizable.

This example shows that for certain functions g and *b*, N and $g^n(A_0)$, $0 < n < N$, can be statically defined as expressions. It is an interesting topic to characterize the class of functions which have the property and to study the complexity of deriving the expressions. For example, we can define the property "function f(n) can be statically defined as an expression" as that "f(n) can be computed without using GOTO and repetitions". Namely, f(n) is a primitive recursive function. Based on this definition, as long as g and *b* are primitive recursive functions and N is bounded, N and $g^n(A_0)$ can be statically defined as expressions ([BRAINERD-74], p64). Since the majority of practical g and *b* are primitive recursive, we claim that in practice, WHILE loops can be converted to WHILE loops as above.

Even in the most general case, when N cannot be calculated without using a WHILE loop (at runtime), we can calculate the N using the following WHILE loop (at runtime) which produces the values of A[n] $= g^n(A_0)$, $0 < n < N$, as side products.

```
A[0] := A0; N := 0;
WHILE b(A[N]) DO
      A[N+1] := g(A[N]);
      N := N + 1;
ENDWHILE;
```

This WHILE loop stands for the sequential portion of the original WHILE loop that has to be done sequentially. So whatever amount of parallelism we can obtain by parallelizing RS will speed up the program. The speed up will be analyzed in section 6.

We have similar results for the form (2) WHILE loops.

PROPOSITION 5.3. A form (2) WHILE loop can be converted to the following FOR loop

```
N = min ( n | b(g^n(A0)) = false )
FOR i:=1 TO N DO
    RS(A-> g^(i-1)(A0));
ENDFOR;
A = g^N(A0);
```

$N = \min(\, n \mid b(g^n(A_0)) = \textbf{false} \,)$
FOR i:=1 TO N DO
$\quad RS(A\text{->}\ g^{i-1}(A_0));$
ENDFOR;
$A = g^N(A_0);$

PROOF. Omitted.

6. PIPELINING WHILE LOOPS

In [CYTRON-86], DOACROSS was proposed to parallelize FOR loops that have loop carried dependence. A similar idea can be used to execute WHILE loops in the form of pipelining.

The power of pipelining lies in the overlapping of operations. This can be used to execute a loop by overlapping part of the iterations. Another feature of pipelining is that the operations do not have to be data independent as long as certain conditions hold. This latter feature makes it especially attractive for executing WHILE loops.

We first review the classic example of the two-stage instruction pipeline. The process of each instruction is decomposed into a fetch stage and an execution stage, and the instruction stream is fed into the two-stage pipeline as in figure 6.1 (a). The explicit condition that allows a stream to be executed in the pipeline is that the first stages should not depend on the immediately preceding second stage operations (In other words, the Fetch stage and Execution stage do not have loop carried dependence). The implicit conditions that are guaranteed by the single two-stage hardware is that no two first/second stage operations should be executed at the same time. For a steam of N instructions, the timing of the execution is shown in figure 6.1 (b). The total execution time is

$$T(fetch) + T(execution) + N * MAX(T(fetch), T(execution)).$$

When we have multiple processors, each capable of executing the two stages, we have a multiprocessor two-stage pipeline (see figure 6.2). The conditions to guarantee the proper pipelining is that (a) first stages should not depend on any of the second stages and (b) the second stages must be independent of any of the other second stage operations. Note that, these two conditions do not exclude the possibility that the first stages have loop carried dependence on themselves and that the second stages depend on the first stages.

Several timing patterns are possible depending on the patterns of dependence. The pattern in which the first stages are dependent is shown in figure 6.3. The timing of the pattern is shown in figure 6.4.

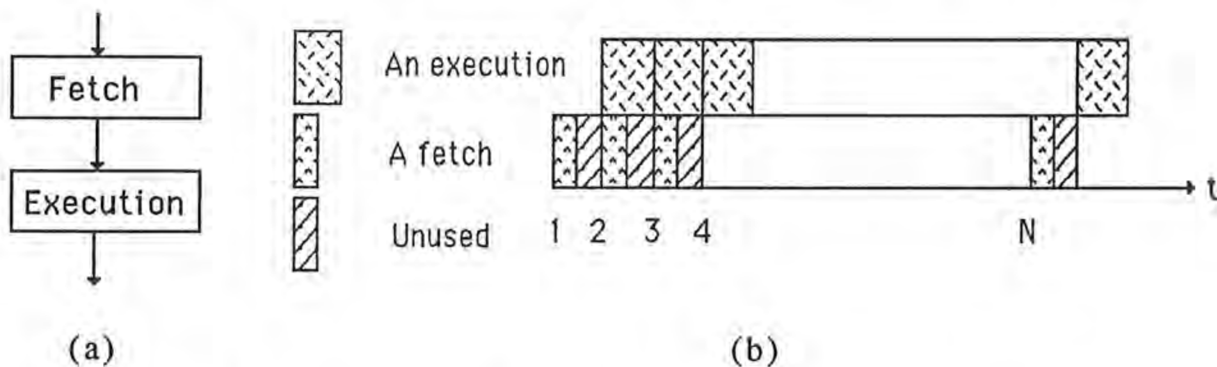

Figure 6.1. (a) two-stage pipeline, (b) timing of the pipeline.

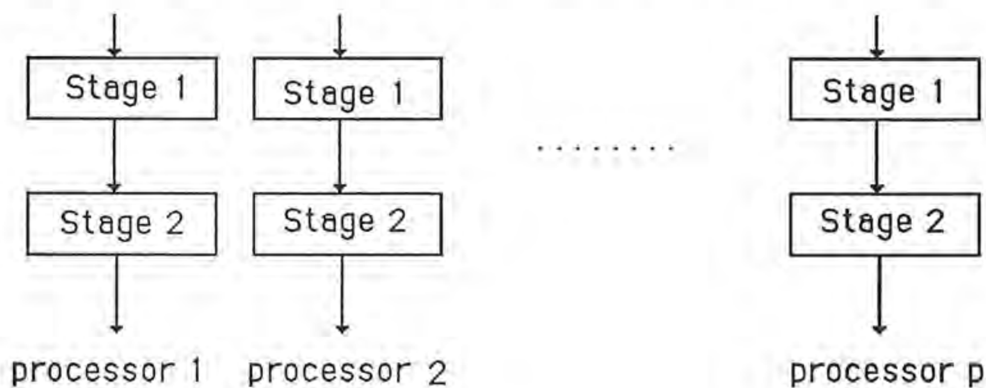

Figure 6.2. A multiprocessor two-stage pipeline.

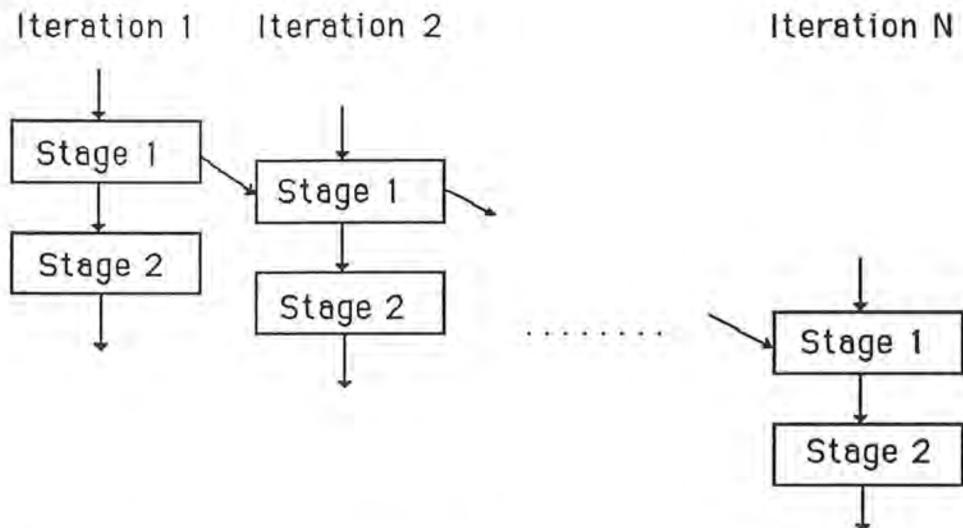

Figure 6.3. An execution of multiprocessor two stage pipeline.

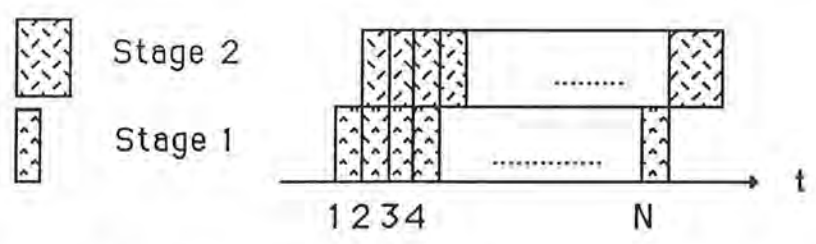

Figure 6.4. Timing of the multiprocessor two stage pipeline.

The multiprocessor pipeline is a good device to execute the form (2) WHILE loop. The conditional checking (b(D)) and the statements in LC can be thought of as the first stage operation, while the statements in RS can be considered as the second stage operations. The pipelining is safe as long as RS does not have loop carried dependence on itself, since this and the properties of form (2), namely LC does not depend on RS, are exactly the conditions required to execute the loop on the multiprocessor pipeline. Because we allow LC to have loop carried dependence, the WHILE loop can be executed as in figure 6.4.

The advantage of pipelining a WHILE loop over parallelizing a WHILE loop is that we do not have to calculate an upper bound on the number of loop iterations. Another advantage is that the first stage of the pipeline does not need to be used only for loop counting. Even if RS has loop carried dependence, we may be able to isolate the part

of RS that has loop carried dependence and put it into the first stage. This suggests a generalization of the form (2) WHILE loop as stated in the following proposition.

PROPOSITION 6.1. Any WHILE loop of the following form can be pipelined as long as S1 does not depend on S2 and S2 does not have loop carried dependence on itself.

```
(3)   D := D0;
      WHILE b(D) DO
            S1;
            S2;
      ENDWHILE;
```

PROOF. This is obvious from the conditions of the multiprocessor pipeline. Q.E.D.

7. SPEEDUP ANALYSIS.

As the discussion in section 4 showed, transforming a loop to either form (1) or form (2) will duplicate a certain amount of code in the original loop body. It is a challenging task to devise a transformation algorithm to minimize the duplications. For now, we assume the time taken by the sequential execution of is t units, and the execution time of LC is $t_1 = at$ units and RS is $t_2 = bt$ units, $1 < a + b < 2$, $0 < a, b < 1$. Then, the execution time of the original loop is $N^* t$. The time taken by one iteration of the transformed loop is:

$$(t_1 + t_2) = (a+ b) * t.$$

For the parallelizing schema of section 4, the calculation of N takes N^* t_1 time units when N has to be calculated at runtime using a WHILE loop, or takes a constant time if a formula for N can be found at compilation time. Since the statement RS can be done in parallel (assuming we have sufficiently many processors), the total execution time of the transformed loop is $mNt_1 + t_2$ where m is either 0 (the calculation of N can be done completely at compilation time) or 1 (the calculation of N has to be done completely at runtime by a

WHILE loop). The speed up is

$$SP = Nt / (mNt_1 + t_2) = N / (mNa + b).$$

When $m = 0$,

$$SP_1 = N / b .$$

When $m = 1$,

$$SP_2 = N / (Na + b) = 1/a - b / (Na^2 + ba).$$

Since $b < 1$, SP_1 may be greater than N. This is achieved by performing the runtime loop counting computation at compilation time.

On the other hand, SP_2 is dominated by $1/a = Nt / Nt_1$, which is the ratio of the execution time of the original loop to its sequential portion that calculates N. Since the parallelized loop cannot be executed faster than the sequential portion, $1/a$ is the best speedup for this case.

For the pipelining schema of section 5, the N LC's are executed sequentially. This is exactly the case of the parallelizing schema when $m = 1$. So the speedup is SP_2.

In summary, the speedups of both parallelizing schema and the pipelining schema are dependent on how much of the loop body is loop independent. The parallelizing schema can further speed up the loop if the calculation of N can be done statically.

8. PARALLEL WHILE CONSTRUCTS

We propose form (1) and (2) WHILE loops discussed in section 4 as the parallel constructs to explicitly express parallelism. Although it seems a little counterintuitive that in form (2) the loop counting statements for the next iteration are executed upon entry to the loop, sophisticated users may find it useful to specify parallel WHILE loop to by-pass compiler's transformation. The abstract constructs can be specified as:

```
(1')    PARWHILE b(i1,i2,...ik) DO
                S;
        NEXT (i1,i2,...ik) := exp
        ENDWHILE;

(2')    PARWHILE b(i1,i2,...ik) NEXT (i1,i2,...ik) := exp DO
                S;
        ENDWHILE;
```

As with the FORALL construct, the user must decide that (a) S has no loop carried dependence on itself, and (b) *b* does not depend on S, before coding in either form (1') or (2'). The compiler can translate these constructs into either a parallelized or a pipelined execution without performing data dependency analysis and loop transformations. For example, the loop in Figure 8.1(a) can be coded as in Figure 8.1(b) or Figure 8.1(c).

```
(a)     WHILE i <= n DO
                x[i]:= comp(i);
                IF i < 5 THEN i+1 ELSE i + 2 ENDIF
                y[i] := comp(i)
        ENDWHILE;

(b)     PARWHILE i <= n DO
                x[i]:= comp(i);
                IF i < 5 THEN i' := i+1 ELSE i' := i + 2 ENDIF;
                y[i'] := comp(i');
                NEXT IF i < 5 THEN i+1 ELSE i + 2 ENDIF;
        ENDPARWHILE;
```

```
(c)     PARWHILE i <= n DO NEXT IF i < 5 THEN i+1 ELSE i + 2 ENDIF;
            IF i <= 5 THEN i':=i-1 ELSE i' := i - 2 ENDIF;
            x[i'] := comp(i');
            y[i] := comp(i);
        ENDPARWHILE;
```

Figure 8.1. Explicitly Expressing Parallel WHILE loops.

9. CONCLUSIONS.

Though a WHILE loop looks inherently sequential, as Proposition 3.1. suggested, a WHILE loop may be parallelized automatically by first transforming it into either form (1) or form (2) and then either converting the transformed loop into a parallel FOR loop or pipelining the loop. This method is general in the sense that it works for any given WHILE loop and it can be used equally well to parallelize other iterative control structures such as REPEAT loops. To save the transformation work, a programmer can also use the parallel WHILE constructs to directly code in either form (1') or form (2').

10. REFERENCES

[AHO-87] Aho, Alfred, U. Ravisethe, and Jeffres D. Ullman, "Compilers, Principles, Techniques, and Tools," Addison-Wesley, 1987.

[ALLEN-83] Allen, J. R., Dependence Analysis for Subscripted Variables and Its Applications to Program Transformations, Ph.D. Thesis, Rice University, Houston, April 1983.

[BRAINERD-74] Brainerd, W.S. and L.H. Landweber, Theory of Computation, John Wiley & Sons, 1974.

[CYTRON-86] Cytron, Ron, "Doacross: Beyond Vectorization for Multiprocessors," Proceedings of the 1986 International Conference on Parallel Processing.

[KARP-88] Karp, Alan H. and Robert G. Babb II, "A Comparison of 12 Parallel Fortran Dialects," IEEE Software, vol. 5, no. 5, Sept 1988.

[KUCK-81] Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," Proc. 8th ACM Symp. Principles Programming Languages, Jan. 1981, pp. 207-218.

[LAMPORT-75] Lamport, L., On Programming Parallel Computers, in Proc. of a Conf. on Prog. Lang. and Compilers for Parallel and Vector Machines, New York, March 18-19, 1975.

[LEE-85] Lee, Gyungho, Clyde P. Kruskal, and David J. Kuck, An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors, IEEE Trans. on Computers, Vol. c-34, No. 10, October 1985.

[LUBECK-85] Lubeck, O. M., P. O. Frederickson, R. E. Hiromoto, and J. W. Moore, "Los Alamos Experiences with the HEP Computer," in Parallel MIMD Computation: HEP Supercomputer and Its Applications (MIT Press, 1985).

[PADUA-86] Padua, D.A., and M.J. Wolfe, Advanced Compiler Optimizations for Supercomputers, CACM, 29(12), Dec. 1986.

[WOLFE-82] Wolfe, M.J., Optimizing Supercompiler for Supercomputers, Ph.D. Thesis, University of Illinois, Urbana-Champaign, 1982.