The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors*

Rajiv Gupta
Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor, NY 10510

Abstract - Parallel programs are commonly written using barriers to synchronize parallel processes. Upon reaching a barrier, a processor must stall until all participating processors reach the barrier. A software implementation of the barrier mechanism using shared variables has two major drawbacks. Firstly, the execution of the barrier may be slow as it may not only require execution of several instructions and but also result in hot-spot accesses. Secondly, processors that are stalled waiting for other processors to reach the barrier are essentially idling and cannot do any useful work. In this paper, the notion of the **fuzzy barrier** is presented, that avoids the above drawbacks. The first problem is avoided by implementing the mechanism in hardware. The second problem is solved by extending the barrier concept to include a region of statements that can be executed by a processor while it awaits synchronization. The barrier regions are constructed by a compiler and consist of several instructions such that a processor is ready to synchronize upon reaching the first instruction in this region and must synchronize before exiting the region. When synchronization does occur, the processors could be executing at any point in their respective barrier regions. The larger the barrier region, the more likely it is that none of the processors will have to stall. Preliminary investigations show that barrier regions can be large and the use of program transformations can significantly increase their size. Examples of situations where such a mechanism can result in improved performance are presented. Results based on a software implementation of the fuzzy barrier on the Encore multiprocessor indicate that the synchronization overhead can be greatly reduced using the mechanism.

Keywords - multiprocessor systems, barrier synchronization, parallelizing compilers.

1. Introduction

In order to achieve efficient parallel execution of tightly synchronizing streams of instructions, the development of fast synchronization mechanisms is essential. A commonly used mechanism for synchronizing the parallel execution of streams is the barrier[1]. An application that creates streams for exploiting fine-grained parallelism schedules the parallel execution of streams on processors. Upon reaching a barrier the processor must wait until all participating processors reach the barrier. Barriers may be automatically introduced by a parallelizing compiler[2] or may be introduced explicitly by the programmer[3]. Barriers can be easily implemented in software using one or more shared variables. However, such implementations entail significant run-time overhead as they require execution of several instructions in each stream in order to achieve synchronization. The synchronization overhead increases linearly, or for the best possible software implementation, logarithmically[4] with the number of processors synchronizing at the barrier. Furthermore, the techniques are known to cause hot-spot accesses[4]. A processor upon reaching a barrier is idle until other processors also reach the barrier[1]; thus no useful work is done by the processor while waiting to synchronize at the barrier. In this paper the notion of the fuzzy barrier, a mechanism that reduces both the run-time overhead and the idling of processors, is introduced.

In order to reduce the run-time overhead due to execution of additional instructions, barriers specified in instruction streams are detected by the hardware to ascertain when a processor is ready to synchronize. All participating processors are simultaneously informed of this event, and when all of the processors have reached the barrier, they simultaneously recognize that synchronization has taken place. This eliminates the run-time overhead caused by executing several instructions to achieve barrier synchronization. However, a single instruction is required to initialize a barrier. Once this has been done, the processors can repeatedly synchronize without executing any overhead instructions. Since the

* A patent application for the fuzzy barrier has already been filed.

cost of using a barrier mechanism is extremely low it can be used frequently, thus facilitating the exploitation of fine-grained parallelism. Hot-spot accesses are avoided as the mechanism does not rely upon shared memory to achieve synchronization. The above strategy applies only to situations in which the processes synchronizing at a barrier are simultaneously executing on different processors. Thus, the number of streams synchronizing at a barrier can at most equal the number of processors in the system.

In order to reduce the idling time of processors at barriers, estimates of the time taken to execute different parts of a program are first used by the compiler to schedule approximately equal amounts of work on each processor between successive barrier synchronizations. However, even if the compiler distributes the computation so that all processors execute identical code, they may not arrive at a barrier at the same time. The code being scheduled may contain conditional statements and different processors may follow different control paths and thus execute varying number of instructions. Furthermore, the times for memory accesses may vary for different processors. Due to a cache miss, a processor may fall behind in execution even if all processors are executing identical instructions. The barrier mechanism should be able to tolerate drift in the speed of execution of processors if idling at the barriers is to be reduced. The fuzzy barrier mechanism provides tolerance to this drift by specifying a range of instructions over which the synchronization is to take place rather than a specific point at which the processors must synchronize. Upon reaching the first instruction in this range, a processor is ready to synchronize. However, it can continue to execute the remaining instructions in the region even if synchronization has not yet occurred. The mechanism, though implemented in hardware, relies upon the compiler to discover this range of instructions. The processors may be executing different instructions from the specified range of instructions at the time of synchronization; hence the name fuzzy barrier.

A flexible barrier of the kind described has several advantages. If the processors in the system are pipelined, repeated synchronization is less likely to degrade the performance of the pipeline because the synchronization point is not exactly specified. Thus upon reaching a barrier, the processor may be able to issue instructions even if the synchronization has not taken place. Since there is almost no synchronization overhead, concurrentizable loops requiring barrier synchronization can be efficiently executed on multiple processors even if the size of the loop body is relatively small. Application of transformations such as cycle shrinking[5] depend heavily upon use of barriers. Availability of an efficient barrier mechanism makes their application practical. A parallelizing compiler can employ such a mechanism to exploit instruction level parallelism using techniques similar to those used in VLIW machines[6,7,8].

In subsequent sections the semantics of the fuzzy barrier is described in detail. An example showing the compilation process to exploit such a mechanism is presented. Code reorganization techniques to increase the range of instructions over which synchronization is to occur are described. Potential ways in which the mechanism can be used to achieve higher speed-ups are presented. An implementation of the mechanism in a prototype multiprocessor system based upon RISC[9] processors is currently in progress. The issues of using the barrier mechanism in presence of interrupts and subroutine calls are not addressed in this paper.

2. Semantics of the Fuzzy Barrier

Instruction streams are viewed as consisting of **barrier regions** and **non-barrier** regions. In Fig. 1 the shaded regions represent barrier regions and the unshaded regions are non-barrier regions. Streams with no barrier regions have no barrier synchronizations, while a shaded region extending across all or a subset of streams indicates a barrier and forces the processors to synchronize. The barrier regions for different streams may contain varying number of instructions. The functionality of the fuzzy barrier is briefly described as follows:

No processor can execute an instruction from its respective non-barrier region (UNSHADED2) following the barrier region (SHADED) until all processors have executed the instructions in their respective non-barrier regions (UNSHADED1) preceding the barrier region.

i.e., $\forall\ i$ $U_2^{P_i}$ can be executed iff $\forall\ j$ $U_1^{P_j}$ have been executed

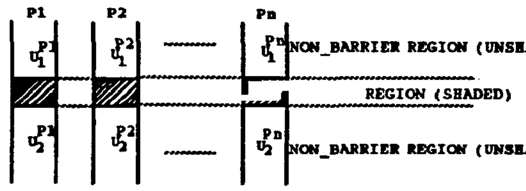

Fig. 1. Fuzzy Barrier

The semantics of the mechanism is described in detail below.

Definition: A processor is considered to have **exited** a region (barrier or non-barrier) of a stream if it has completed the execution of all the instructions in that region. It has **entered** a region if it has started the execution of an instruction from that region.

Definition: A processor is **ready to synchronize** if it has exited the non-barrier region preceding a barrier region. It should be noted that exiting this non-barrier region is not same as entering the barrier region for a pipelined machine, for a pipelined machine overlaps the execution of multiple instructions. Therefore, a processor may enter the barrier region before exiting the preceding non-barrier region.

Condition for Synchronization: Processors have

synchronized at a barrier if and only if they have all exited their respective non-barrier regions preceding the barrier region.

Condition for Stalling: A processor can enter a non-barrier region following a barrier region if and only if synchronization has occurred. Thus, if the synchronization is yet to occur when the processor exits the barrier region, it is not allowed to enter the non-barrier region and must idle. In other words, the execution of the stream is **stalled**.

From the above description it is clear that when the execution of a stream reaches the first instruction of a barrier region, it does not have to stop immediately but can continue to execute even if other streams haven't reached their corresponding barrier regions. Similarly upon reaching the last instruction in a barrier region, the processor can continue even if other processors have not reached the end of their corresponding barrier regions. If the barrier region for a stream consists of n instructions, then at the point of synchronization, the processor could have executed 0 to n instructions from the barrier region. The tolerance of the mechanism to the variation in the rate at which each stream progresses is limited by the number of instructions in the barrier regions. Thus, the larger the barrier regions, the less likely it is that the processors will stall.

3. Branch Instructions in Barrier Regions

The instructions that form a barrier region can contain unconditional as well as conditional branch instructions. Thus, any sequence of instructions that are consecutive along a control path in the program can form a barrier. Branches in the barrier region allow a barrier region to have multiple exits. Branches into a barrier region from non-barrier regions permit the barrier region to have multiple entry points. A processor enters the barrier when it executes an instruction from the barrier region and has crossed the barrier as soon as it executes the first instruction from a non-barrier region. The advantage of permitting branches in barrier code is that entire control structures, such as loops and if-statements, can be included in a barrier region. Furthermore, the sequence of instructions forming the barrier may not be physically contiguous. Thus, for a loop whose iterations are separated by a barrier, the barrier region can contain code not only from the end of one iteration but also from the start of the subsequent iteration. As will be demonstrated through an example later in the paper, typically the barrier region corresponding to a barrier at the end of a loop body will, in fact, extend across consecutive iterations.

The destination of a branch instruction in the barrier region should either be an instruction in the same barrier region or an instruction in a non-barrier region. If the destination is within the barrier region, the processor remains in the barrier region. On the other hand if the destination is in a non-barrier region, the processor exits the barrier region. The compiler should not generate code where control can be transferred directly from one barrier to another. Such branches can result in improper synchronization and deadlocks if the hardware cannot distinguish among different barriers. Consider the example in Fig. 2, where there are two barriers at which the processors must synchronize, and consider a branch instruction that transfers control of processor P_1 directly from $barrier_1$ to $barrier_2$. If this branch is taken, P_1 will cross both the barriers by synchronizing with P_2 only once when P_2 reaches $barrier_1$. Also P_2 will be deadlocked at $barrier_2$ waiting for a synchronization that will never take place. It should be noted that the above problem will not arise in an implementation which explicitly specifies unique identifiers for barriers in the code.

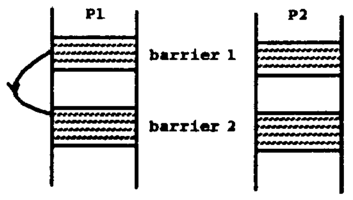

Fig. 2. Invalid Branch

4. Compiler Support

In this section the compilation process for constructing barrier and non-barrier regions is demonstrated using the Poisson solver[3] (Fig. 3(a)). The code shown in Fig. 3(b) is the non-deterministic parallel version of the algorithm. The example also demonstrates that code reordering can increase the size of the barrier region significantly.

The iterations of the inner loops of the Poisson solver can be executed in parallel. Thus, M^2 processors can be used to execute a single iteration of the outermost loop. A processor ready to begin a new iteration of the outer loop has to be informed when all work from the previous iteration has been completed. This can be achieved by introducing a barrier at the end of each iteration. The code executed by each of the M^2 processors is shown in Fig. 3(b). Storage related dependences among the parallel iterations due to loop variables are eliminated by creating private copies of i, j and k for each subtask.

```
/* Boundary conditions are held in rows/columns 0 and M+1 */
 int P[M+1][M+1];

 for (k=1; k<=10*M; k++) do seq
   for (i=1; i<=M; i++) do par
     for (j=1; j<=M; j++) do par
         P[i][j] = (P[i][j+1] + P[i][j-1] + P[i+1][j] + P[i-1][j])/4;
```

Fig. 3(a). Poisson Solver

Processor $P_{l,m}$ *where* $1 \leq l,m \leq M$:

```
 Private i,j,k;
 i=l; j=m;
 for (k=1; k<=10*M; k++)
    {
      P[i][j] = (P[i][j+1] + P[i][j-1] + P[i+1][j] + P[i-1][j])/4;
      barrier;
    }
```

Fig. 3(b). Per Processor Tasks

The barrier region is constructed by examining instructions along the control flow path on which the barrier lies. The instructions preceding and following a barrier are candidates for inclusion in the barrier region. For the above example, since the barrier is at the end of a loop, these instructions include the instructions from two consecutive loop iterations. Our goal is to include as many instructions as possible in the barrier region.

In order to construct the barrier and non-barrier regions the instructions that must be in the non-barrier regions are identified. These instructions are referred to as the **marked instructions**. All instructions starting with the first marked instruction and ending at the last marked instruction are included in the non-barrier region. The remaining instructions form the barrier region. The marked instructions are those instructions which either access a value computed by another processor or compute a value that will be accessed by another processor. In order to ensure that a processor accesses a value after it has been computed by another processor, barrier synchronization is introduced. In the example presented, a barrier at the end of each iteration of the outer loop enforces loop carried dependences which are the data dependences among different iterations of a loop. Thus, by analyzing the loop carried dependences, the instructions that must be included in the non-barrier region can be identified.

The instructions that read/write array P are involved in the loop carried dependences and thus must be included in the non-barrier region, because the values computed by the processors during an iteration of the outer loop must be available during the next iteration. The intermediate code[10] for the Poisson solver is shown in Fig. 4(a). Instructions I_1, I_2, I_3 and I_4 are the four marked instructions that read/write elements of the array. Thus, the non-barrier region extends from I_1, the first instruction, to I_4, the last instruction that modifies the array. The remaining instructions are included in the barrier region. The code shown in Fig. 4(a) contains a fuzzy barrier that ensures that a processor does not execute any instruction from the non-barrier region during iteration $k+1$ until all processors have completed the execution of their respective non-barrier regions during iteration k.

As mentioned earlier, it is preferable if the non-barrier regions are small and barrier regions are large. *Code reordering*[11,12] can be performed to move instructions, other than the marked instructions, from the non-barrier region to the barrier region. The process of code reordering requires examining the dependences among the instructions to determine if they can be reordered and finding a suitable ordering. In the example, the instructions that compute the addresses of the array elements can be executed before any of the array elements are actually accessed and can be moved out of the non-barrier region. This leaves only a small number of instructions in the non-barrier region as shown in Fig. 4(b).

```
/* Let M = 2; int  P[3][3]; declaration of the array
  Let P be the base address of the array */
Non-barrier:
        ......
--------------------------------------------------------------------------------
Barrier:
        i = 1
        j = m
        k = 1
L1:     T1 = j + 1
        T2 = 12 * i
        T3 = T2 + P
        T4 = 4 * T1
        T5 = T3 + T4          /* T5 <- address of P[i][j+1] */
        T6 = j - 1
        T7 = 12 * i
        T8 = T7 + P
        T9 = 4 * T6
        T10 = T8 + T9         /* T10 <- address of P[i][j-1] */
--------------------------------------------------------------------------------
Non-barrier:
I1:     T11 = [T5] + [T10]  /* T11 = P[i][j+1] + P[i][j-1] */
        T12 = i + 1
        T13 = 12 * T12
        T14 = T13 + P
        T15 = 4 * j
        T16 = T14 + T15      /* T16 <- address of P[i+1][j] */
I2:     T17 = T11 + [T16]   /* T17 = T11 + P[i+1][j] */
        T18 = i - 1
        T19 = 12 * T18
        T20 = T19 + P
        T21 = 4 * j   .
        T22 = T20 + T21      /* T22 <- address of P[i-1][j] */
I3:     T23 = T17 + [T22]   /* T23 = T17 + P[i-1][j] */
        T24 = T23 / 4
        T25 = 12 * i
        T26 = T25 + P
        T27 = 4 * j
        T28 = T26 + T27      /* T28 <- address of P[i][j] */
I4:     [T28] = T24             /* P[i][j] = T24 */
--------------------------------------------------------------------------------
Barrier:
        k = k + 1
        if k<=20 go to L1
--------------------------------------------------------------------------------
Non-barrier:
        ......
```

Fig. 4(a). Barrier Region

Given a piece of code that forms the non-barrier region, code reordering to move instructions to the barrier region can be carried out as follows. First a directed acyclic graph (DAG)[10] representing the data dependences for the code in the non-barrier region is built. Since a DAG represents the dependences among the intermediate code statements, it can be used to find another legal ordering of instructions that results in smaller non-barrier regions. First we consider for scheduling only the instructions from the non-barrier region that are not marked (i.e., instructions other than I_1, I_2, I_3 and I_4 in the example). All instructions scheduled during this phase are essentially moved into the barrier region preceding the non-barrier region. Next, the scheduling of instructions is carried out in manner that tries to schedule the marked instructions as early as possible. This process continues until all marked instructions have been scheduled. In the example, in addition to instructions I_1, I_2, I_3 and I_4 only one more instruction is scheduled during this phase. The instructions scheduled during this phase form the non-barrier region. After the last non-barrier instruction has been scheduled, the final phase generates an ordering for the remaining instructions. These instructions are included in the barrier region following the non-barrier region and hence are moved out of the non-barrier region. In the example presented, there are no instructions left to be scheduled during this phase.

```
Non-barrier:
        .....
-------------------------------------------------------------------------
Barrier:
        i = 1
        j = m
        k = 1
L1:     T1 = j + 1
        T2 = 12 * i
        T3 = T2 + P
        T4 = 4 * T1
        T5 = T3 + T4        /* T5 <- address of P[i][j+1] */
        T6 = j - 1
        T7 = 12 * i
        T8 = T7 + P
        T9 = 4 * T6
        T10 = T8 + T9       /* T10 <- address of P[i][j-1] */
        T12 = i + 1
        T13 = 12 * T12
        T14 = T13 + P
        T15 = 4 * j
        T16 = T14 + T15     /* T16 <- address of P[i+1][j] */
        T18 = i - 1
        T19 = 12 * T18
        T20 = T19 + P
        T21 = 4 * j
        T22 = T20 + T21     /* T22 <- address of P[i-1][j] */
        T25 = 12 * i
        T26 = T25 + P
        T27 = 4 * j
        T28 = T26 + T27     /* T28 <- address of P[i][j] */
-------------------------------------------------------------------------
Non-barrier:
I1:     T11 = [T5] + [T10]  /* T11 = P[i][j+1] + P[i][j-1] */
I2:     T17 = T11 + [T16]   /* T17 = T11 + P[i+1][j] */
I3:     T23 = T17 + [T22]   /* T23 = T17 + P[i-1][j] */
        T24 = T23 / 4
I4:     [T28] = T24           /* P[i][j] = T24 */
-------------------------------------------------------------------------
Barrier:
        k = k + 1
        if k<=20 go to L1
-------------------------------------------------------------------------
Non-barrier:
        ......
```

Fig. 4(b). Barrier Region After Code Reordering

Since the barrier region in Fig. 4(b) is large, a processor can execute the majority of instructions from the next iteration even if synchronization at the end of the current iteration has not yet taken place. In the example presented, the reordering was performed at intermediate code level as this is more effective than reordering machine code. After machine code has been generated, the opportunities for reordering are restricted due to dependences introduced from register or other resource usages.

In addition to reordering at the intermediate code level, statement level transformations may be useful in increasing the size of the barrier region. The example shown in Fig. 5 illustrates the use of loop distribution[13] to increase the size of the barrier region. Loop distribution is a transformation that takes a loop with several statements and divides it into multiple loops, each of which contains only a subset of statements from the loop body. For example the loop in Fig. 5(a) has two statements S_1 and S_2. Application of loop distribution results in two loops with S_1 and S_2 as their respective loop bodies (see Fig. 5(c)). In this example, the barrier synchronization is required between consecutive iterations of the outer loop, because the values computed by S_1 during iteration i of the outer loop are needed during the execution of S_1 in iteration $i+1$ and the processor using the value may not be same as the processor computing the value. Since the outer loop must be sequentialized to enforce the dependences due to statement S_1, the execution of statement S_2 can be performed as part of the barrier region. If loop distribution is not applied, the barrier region includes a single execution of statement S_2 (Fig. 5(b)), which is the last execution of S_2 by a processor in an iteration of the outer loop. On the other hand if loop distribution is applied, the barrier region consists of an entire loop that includes all executions of the statement S_2 as shown in Fig. 5(c).

```
for (i=1; i<N; i++) do seq
  for (j=1; j<=M; j++) do par
    {
      S1: a[j,i] = a[j+1, i-1] + 2;
      S2: b[j,i] = b[j,i] + c[j,i];
    }
```

(a) Original Code

```
Task_p, where 0 ≤ p ≤ S−1:
for (i=1; i<N; i++) {
    for (j=p*⌈M/S⌉+1; j<min(M,(p+1)*⌈M/S⌉); j++) {
       S1; S2;
     }
    S1;
    -------
    | S2; | barrier region
    -------
  }
```

(b) Without Loop Distribution

```
Task_p, where 0 ≤ p ≤ S−1:
for (i=1; i<N; i++) {
      for (j=p*⌈M/S⌉+1; j<=min(M,(p+1)*⌈M/S⌉); j++) S1;
      /* start of barrier region */
      for (j=p*⌈M/S⌉+1; j<=min(M,(p+1)*⌈M/S⌉); j++) S2;
      /* end of barrier region */
  }
```

(c) After Loop Distribution

Fig. 5. Enlarging Barrier Regions

In the example presented above, a significant amount of source level code was included in the barrier region. At source level a programmer may be able to construct barrier regions while coding an application. This indicates it may be possible for both the compiler and the programmers to exploit the semantics of the fuzzy barrier. Commercial multiprocessor systems, such as Encore[14] and Sequent[15], support the barrier mechanism as part of their parallel programming library which is available to application programmers. By supporting the fuzzy barrier in software, the performance of the multiprocessor system may be further enhanced.

5. Multiple Barriers

All of the processors in the system are not forced to synchronize every time a barrier is used. Disjoint subsets of processors can independently synchronize among themselves. A *mask* is provided in each processor for specifying particular processors participating in a barrier synchronization. If it is known at compile-time that the streams would definitely be created and interact in a precisely predictable fashion, the synchronizations can be achieved using a single barrier. The masks for each of the processors can be set to either synchronize with or ignore other streams. But if the streams are created dynamically or are conditionally created, their existence is not known until run-time. In this situation multiple barriers are used. Logically distinct barriers are assigned to different subsets of streams that do not know of each others existence. In addition to the mask a *tag* is provided to indicate the identity of a barrier. Two processors can only synchronize at a barrier if their tags match. Both the mask and the tag are set by the processors under software control. Barriers are allocated when the streams are created. The creation of the first stream does not require allocation of a barrier as there is no other stream with which it can synchronize. Subsequently, creation of every stream requires allocation of at most one barrier which may be used by the newly created stream to synchronize with its parent. Thus, in a N processor system which allows creation of at most N streams, a maximum of $N-1$ barriers is needed. Different subsets of streams must synchronize using logically different barriers. In other words, the processors must know the identity of a barrier to achieve correct synchronization.

Consider the example shown in Fig. 6 where the barriers are essentially being used to merge streams. Different subsets of processors synchronize at different barriers. Note that processor P_3 engages in barriers B_1 and B_2, processor P_2 engages in barriers B_2 and B_3 and finally P_1 engages in barrier B_3. Processor P_1 upon reaching barrier B_3 may incorrectly synchronize with processor P_2, when P_2 reaches barrier B_2, if the barriers are not given different identities. From this example it is clear that in a N processor system which allows creation of at most N streams, a maximum of $N-1$ barriers is needed. The streams that need to synchronize repeatedly can reuse the barrier shared by them. Disjoint subsets of a group of streams that share the same barrier can synchronize by manipulating their masks.

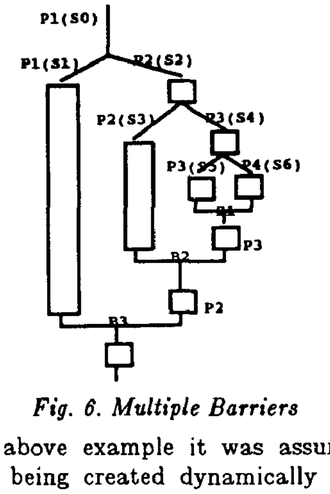

Fig. 6. Multiple Barriers

In the above example it was assumed that the streams were being created dynamically or are conditionally created. For the same set of streams, if it was known at compile-time that the streams would definitely be created and interact precisely in the manner specified in Fig. 6, the synchronizations can be achieved using a single barrier. By forcing all processors to synchronize each time any two processors need to synchronize, a correct schedule that uses a single barrier can be generated. However, the disadvantage of such an approach is that redundant synchronizations are introduced in the streams. Having multiple barriers eliminates redundant synchronizations and enables decisions regarding creation and destruction of streams to be dynamic. Although static schedules have the advantages of simplicity and low run-time overhead, they lack the capability to spawn a variable number of instruction streams based upon run-time information such as the amount of computation to be performed and the availability of processors. A dynamic schedule can do a better job in allocation of resources based upon the run-time information.

6. Implementation

The fuzzy barrier mechanism is being implemented in a multiprocessor system that uses RISC processors. In this section the implementation is described briefly (for detailed description see [16]). In order to distinguish between instructions from non-barrier and barrier regions, a single bit in each instruction is used. The bit is one if the instruction is from a barrier region and zero otherwise. If there are no instructions that can be included in the barrier region, a null operation is introduced to create a barrier region. An alternative and less expensive approach is to use special instructions that when executed, indicate an entry or exit from a barrier region. If special instructions are used to mark the boundaries of a barrier region then the null operation is no longer needed to represent a null barrier region.

In a non-pipelined machine a processor enters a region at the same time it exits the preceding region. Thus, determining whether a processor is in a barrier region or a non-barrier region can be done simply by examining the current instruction. In a pipelined machine, exiting the non-barrier region and entering the barrier region are not equivalent. A processor will typically enter the barrier region before exiting the non-barrier region because multiple instructions are being executed simultaneously. Thus, checking whether synchronization has occurred or not requires information about all the instructions in the pipeline.

It is assumed that all processors use a common clock and are reset simultaneously. The hardware detects when a processor enters a barrier region, and a signal indicating that the processor is ready to synchronize is broadcast to all other processors. When a processor is ready to synchronize and has received similar signals from the participating processors, it knows that synchronization has taken place. Since the signals are being broadcast and monitored by each processor independently, all processors simultaneously discover the occurrence of synchronization. If a processor reaches the end of the barrier region and tries to execute a non-barrier instruction before synchronization has not taken place, the processor is stalled.

Each processor contains an identical copy of the fuzzy barrier hardware. This consists of a state machine that determines the status of the barrier for the processor, an internal register that contains the current tag and mask for the processor, and some combinational logic which determines whether the processor's tag matches the tags of processors with which it wishes to synchronize. A processor's state machine can be in one of the following states: (i) the processor is executing instructions from a non-barrier region; (ii) the processor is in the barrier region and has not synchronized; (iii) the processor is in the barrier region and has synchronized; and (iv) synchronization has not taken place and the processor is stalled as it has completed the execution of instructions from the barrier region. No explicit reset is required as the state machine returns to the start state when a processor is ready to synchronize again.

In an n processor system, the mask for each processor consists of $n-1$ bits, one bit corresponding to each of the other processors. By setting the mask bits, a processor specifies the processors with which it wishes to synchronize. The tag identifies the current barrier for the processor, and two processors can synchronize only if their tags match. A system with an m bit tag supports 2^m-1 logical barriers, where a combination of all zeros is used to indicate that the processor is not participating in barrier synchronization. The internal register containing the tag and the mask is set under software control. The mask and tag for a processor are determined by the compiler for static scheduling and by the run-time system for a dynamic schedule. All of the processors in the system are not forced to synchronize. Disjoint subsets of processors can be made to independently synchronize among themselves by setting the masks appropriately, without one subset interfering with the other.

Although the fuzzy barrier can be implemented in a system with any number of processors, the number of interconnections among the processors increases with the number of processors. Each processor must broadcast its tag to the other processors in the system. The extensibility of the mechanism is further restricted by the fact that all of the processors share the same clock.

7. Other Applications of the Fuzzy Barrier

In this section, situations in which barrier synchronization can be used are presented. Possible advantages of using a fuzzy barrier extending across several instructions in each of these cases are discussed.

7.1. Variable Length Streams

The advantage of allowing conditional and unconditional branches is the possible inclusion of if-statements in barrier regions. As a result, the time spent in barrier regions can vary from one instruction stream to another. If a single instruction barrier region is used, all processors that execute the lesser number of instructions have to wait. If the entire statement is part of the barrier region then there are situations where the variation in the number of instructions will not result in a stall. This is demonstrated by the example in Fig. 7. If a single instruction barrier is introduced at the end of the loop, the processor that takes the path along S_2, which involves less work, must wait for the other processor before it can continue execution (Fig. 7(b)(i)). On the other hand if the entire if-statement is part of the barrier then even if the two processors take different paths they may not have to stall (Fig. 7(b)(ii)).

```
for (i=1; i<=N; i++) do seq
 for (j=1; j<=2; j++) do par
   {
     S1;
     if cond then S2 else S3;
   }

Task_j:
 for (i=1; i<=N; i++) {
     S1;
   -----------------------------------
   | if cond then S2 else S3; |  barrier region
   -----------------------------------
   }
```

Fig. 7(a). Parallel Loop with if-statements

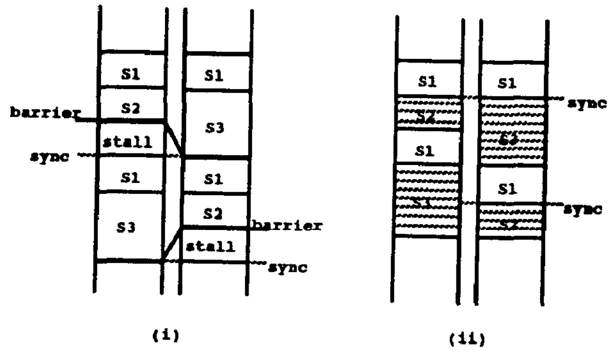

Fig. 7(b). If Statements in Barrier Regions

7.2. Lexically Forward Dependences in Loops

Consider the schedules for processors containing dependences of the type shown in Fig. 8. These dependences point forward in the program source and are called *lexically forward dependences*[17]. Assuming that all processors proceed at the same rate, these dependences are satisfied by the time they are needed. Although the dependences are not likely to cause any delay, in an architecture where processors execute asynchronously, a barrier synchronization is required to guarantee these dependences.

The example in Fig. 9 demonstrates the use of a barrier to enforce lexically forward dependences. In this example the iterations of the inner loop can be executed in parallel and are distributed among four processors. Further assume that the outer loop has been unrolled once to create the tasks shown for each processor. Upon examining the schedules for the processors, dependences between statements $S_{j,l}$ and $S_{j+1,l+1}$ that belong to the corresponding iterations of the loop for processor l and

$l+1$ are found. The outer loop contains loop carried data dependences; therefore a second barrier is introduced at the end of the loop to enforce these dependences.

The intermediate code after code reordering for the example presented is shown in Fig. 10. Upon examining this code, it can be seen that the barrier regions for the loop contain a substantial number of instructions and hence the code is *tolerant of significant drift* in execution of different streams. The code contains two distinct barrier regions, one of which extends across loop iterations and the other is entirely included in a single iteration.

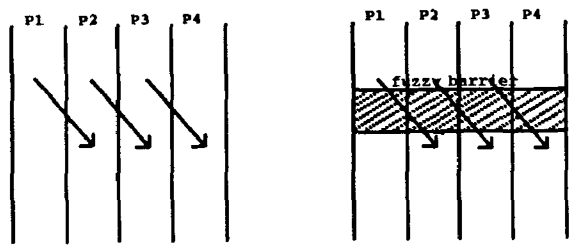

Fig. 8. Lexically Forward Dependences

```
for (j=1; j<10; j++) do seq
  for (i=1; i<5; i++) do par {
      S_{j,i}: a[j][i] = a[j-1][i-1] + i * j;
   }
```

```
Task T_l, where 1 ≤ l ≤ 4:
  Private i;
  i = l;
  for (j=1; j<10; j+=2) do seq
    {
     S_{j,i}: a[j][i]=a[j-1][i-1] + i*j;
     barrier; /* due to lexically forward dependences */
     S_{j+1,i}: a[j+1][i]=a[j][i-1] + i*(j+1);
     barrier; /* due to loop carried dependences */
    }
```

——→ lexically forward dependences
——→ loop carried dependences

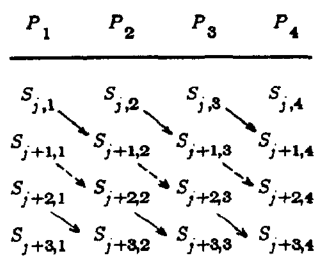

Fig. 9. Enforcing Lexically Forward Dependences

```
/* Let int a[10][4]; be the the declaration of
   the array and "a" be its base address  */

Non-barrier:
        .......
--------------------------------------------------------------------
Barrier due to loop carried dependences:
        i = l
        j = 1
L1:     T1 = i - 1
        T2 = j - 1
        T3 = 16 * T2
        T4 = T3 + a
        T5 = 4 * T1
        T6 = T4 + T5        /* T6 <- address of a[j-1][i-1] */
        T7 = i * j
        T8 = 16 * j
        T9 = T8 + a
        T10 = 4 * i
        T11 = T9 + T10      /* T11 <- address of a[j][i]  */
--------------------------------------------------------------------
Non-barrier:
        T12 = [T6] + T7     /* T12 = a[j-1][i-1] + i*j  */
        [T11] = T12         /* a[j][i] = T12  */
--------------------------------------------------------------------
Barrier due to lexically forward dependences:
        T13 = i - 1
        T14 = 16 * j
        T15 = T14 + a
        T16 = 4 * T13
        T17 = T15 + T16     /* T17 <- address of a[j][i-1] */
        T18 = j + 1
        T19 = i * T18       /* T19 = i*(j+1) */
        T20 j + 1
        T21 = 16 * T20
        T22 = T21 + a
        T23 = 4 * i
        T24 = T22 + T23     /* T24 <- address of a[j+1][i] */
--------------------------------------------------------------------
Non-barrier:
        T25 = [T17] + T19   /* T25 = a[j][i-1] + i*(j+1) */
        [T24] = T25         /* a[j+1][i] = T25  */
--------------------------------------------------------------------
Barrier due to loop carried dependences:
        j = j + 2
        if j<10 go to L1
--------------------------------------------------------------------
Non-barrier:
        .......
```

Fig. 10

7.3. Static Scheduling of Parallel Loops

The schedule for execution of a parallel loop can be statically specified at compile-time if the number of loop iterations and the number of available processors are known at compile-time. It is not possible to distribute the iterations of a loop equally among the processors if the number of iterations is not divisible by the number of processors available. For example, if there are only three processors available to execute the code segment in Fig. 11(a), one of the processors would have to execute two iterations of the inner loop. As a result the other two processors may have to be kept idle for periods of time.

The idling times of processors can be potentially reduced in the following manner. As demonstrated in Fig. 11(b), instead of scheduling the extra iteration on the same processor every time, the processors can take turns in executing the extra iteration. The result of such scheduling strategy is that over multiple iterations of the outer loop, the processors do equal amount of work. Once the work has been equally divided among the processors, an attempt to reorder code to create large barrier regions can be made. In the best possible case, barrier regions large enough to potentially eliminate idling may be found (Fig. 11(c)). In order to achieve the above effect, not only must the inner loop unrolled completely but the outer loop must be unrolled as well. The outer loop is unrolled until the total number of loop iterations available becomes divisible by the number of processors. In the example presented unrolling the outer loop twice results in 12 iterations which is divisible by three, the number of processors. It should be noted that if the barrier mechanism was not flexible in nature, the compiler would have to decide how much code should be moved across the barrier in each of the streams. However, in this example all it had to do was to try and include as many instructions as possible in the barrier region.

```
for (j=1; j<10; j++) do seq
  for (i=1; i<5; i++) do par
  {
     loop body;
  }
```

(a) Original Code

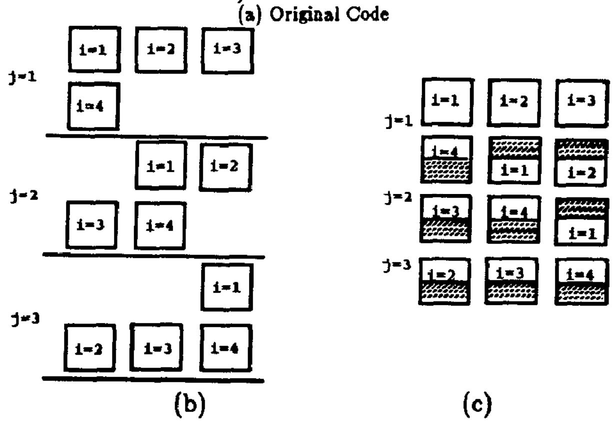

Fig. 11. Static Scheduling

7.4. Run-time Scheduling of Loop Iterations

In situations where the number of loop iterations and/or the number of processors available are not known at compile-time, compiler assisted run-time scheduling techniques can be used. Consider the code segment in Fig. 12. The number of iterations in the inner loop is not known at compile-time. Thus one must wait until run-time to schedule these iterations among the available processors. Several self-scheduling techniques have been developed to distribute the iterations at run-time[18]. Guided Self Scheduling (GSS) is one technique that attempts to distribute the work among the processors so that they complete execution at about the same time[19]. This is desirable as it will reduce the idling of processors at the barrier that must be introduced between iterations of the outer loop.

Idling of processors can also be reduced using the fuzzy barrier. At run-time the iterations can be distributed among the available processors. Next, depending upon the iteration being executed, the processor can choose to execute a version of the loop body compiled with or without a barrier. The successive iterations of the outer loop must be separated by a barrier for correct execution. To achieve this, the first iteration of the inner loop that a processor executes should start with a barrier, the last iteration should be followed by a barrier and the intervening iterations should have no barriers at all. If the processor is allocated only a single iteration, the loop body should be compiled such that it is both preceded and followed by a barrier region. The four versions of the loop are shown in Fig. 12. Compiling multiple versions of code and selecting the appropriate one at run-time is a common practice in parallelizing compilers[20].

```
for (i=1; i<N; i++) do seq
  for (j=1; j<M; j++) do par
  {
    loop body;
  }
```

```
Task_p where 1 ≤ p ≤ S:
  for (i=1; i<N; i++)
    for (j=p; j<M; j+=S) {
       if j is (first and (not last)) iteration   { version_1 }
     else if j is ((not first) and last) iteration { version_2 }
     else if j is not(first or last)     iteration  { version_3 }
     else if j is (first and last)       iteration  { version_4 }
    }
```

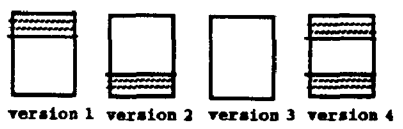

Fig. 12. Run-time Scheduling of Parallel Loops

8. Preliminary Results

Commercial multiprocessor systems, such as Encore[14] and Sequent[15], support the barrier mechanism as part of their parallel programming library which is available to application programmers. By supporting the fuzzy barrier in software, the performance of the system can be further enhanced. A software implementation of the fuzzy barrier on a four processor Encore Multimax has been carried out. For nested loops, similar to those in Fig. 9, the cost of synchronizing four processors was reduced from 10,000μsec to 300μsec as the size of the barrier region was increased from zero instructions to half of the total instructions in the loop body. The cost of barrier synchronization is mainly due to context saves and restores for the tasks that must be stalled. As the size of the barrier region increases, the likelihood of a processor stalling decreases. Therefore, the expensive context saves and restores may be avoided. At source level a programmer may be able to construct barrier regions while coding an application. Thus, it is possible for both the compiler and the programmers to exploit the semantics of the fuzzy barrier.

9. Current Status and Future Work

The fuzzy barrier mechanism is being implemented in a prototype system using RISC processors. It will be used for executing code in VLIW mode as well as code generated by concurrentization of loops. Currently the possibilities of allowing procedure calls from barrier regions are being investigated. This is important because allowing parallel procedure calls can significantly increase the amount of parallelism. The issue of interrupts and traps in a barrier region is also being investigated. Traps are useful as they are often used in RISC based systems to implement floating point operations.

Acknowledgements - I am grateful to Mike Epstein for developing the first hardware realization of the fuzzy barrier and to Carlo Bronco for assisting him. I thank Mike Whelan, Charlie Hill, Mark Tucker, Jim Wendorf and H-Y. Wang for their help and discussions during this work. I am grateful to Horst Mauersberg and Loek Nijman for their encouragement and support. I appreciate and thank the referees and Prof. M.L. Soffa for their suggestions.

References

1. P. Tang and P.C. Yew, "Processor Self-Scheduling for Multiple-Nested Parallel Loops," *Proc. International Conf. on Parallel Processing,* pp. 528-535, August, 1986.

2. R. Gupta, "Synchronization and Communication Costs of Loop Partitioning on Shared-Memory Multiprocessor Systems," *Tech. Report TR-88-019, Philips Laboratories, Briarcliff Manor, NY,* 1988.

3. H.S. Stone, *High-Performance Computer Architecture,* Addison-Wesley Publishing Company, 1987.

4. P.C. Yew, N.F. Tzeng, and D.H. Lawrie, "Distributing Hot-Spot Addressing in Large Scale Multiprocessors," *IEEE Trans. on Computers,* vol. C-36, no. 4, April, 1987.

5. C.D. Polychronopoulos, "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design," *IEEE Trans. on Computers,* vol. 37, no. 8, pp. 991-1004, August, 1988.

6. J.R. Ellis, *Bulldog: A Compiler for VLIW Architectures,* MIT Press, 1986.

7. R. Gupta, "A Reconfigurable LIW Architecture and its Compiler," Dept. of Computer Science; Ph.D. dissertation, Tech. Report 87-3, University of Pittsburgh , August, 1987.

8. R. Gupta and M.L. Soffa, "A Reconfigurable LIW Architecture," *Proc. of the International Conf. on Parallel Processing,* pp. 893-900, August, 1987.

9. D.A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM,* vol. 28, no. 1, pp. 8-21, Jan., 1985.

10. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1986.

11. J. Hennessy and T. Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Trans. on Programming Languages and Systems,* vol. 3, no. 5, pp. 422-448, 1983.

12. W.C. Hsu, "Register Allocation and Code Scheduling for Load/Store Architectures," Dept. of Computer Science; Ph.D. dissertation, University of Wisconsin, Madison, 1987.

13. D.J Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *8th Annual ACM Symp. on Principles of Programming Languages,* pp. 207-218, 1981.

14. "Multimax Technical Summary," Encore Computer Corporation, Marlboro MA, 1987.

15. A. Osterhaug, "Guide to Parallel Programming on Sequent Computer Systems," Sequent Computer Systems, Inc., Beaverton, Oregan, 1987.

16. R. Gupta and M. Epstein, "Achieving Low Cost Synchronization in a Multiprocessor System," Philips Laboratories; Tech. Note TN-88-140, Briarcliff Manor, NY, October, 1988.

17. R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors," *Proc. International Conf. on Parallel Processing,* pp. 836-844, August, 1986.

18. C.D. Polychronopoulos, D.J. Kuck, and D.A. Padua, "Execution of Parallel Loops on Parallel Processor Systems," *Proc. International Conf. on Parallel Processing,* pp. 235-242, August, 1986.

19. C.D. Polychronopoulos and D.J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. on Computers,* vol. C-36, no. 12, pp. 1425-1439, Dec., 1987.

20. M. Byler, J.R.B. Davies, C. Huson, B. Leasure, and M. Wolfe, "Multiple Version Loops," *International Conf. on Parallel Processing,* pp. 312-318, August, 1987.