Employing Register Channels for the Exploitation of Instruction Level Parallelism

Rajiv Gupta
Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor, NY 10510
e-mail: gupta@philabs.Philips.com

Abstract - A multiprocessor system capable of exploiting fine-grained parallelism must support efficient synchronization and data passing mechanisms. This paper demonstrates the use of shared register channels as the communication mechanism among processors in a multiprocessor chip. A register channel is provided with a synchronization bit that is used to ensure that a processor succeeds in reading a channel only after the channel has been written to. In contrast to a VLIW machine a system with channels does not require strict lockstep operation of its processors. This reduces the delays caused by unpredictable events such as memory bank conflicts. Providing channels accessible at the speed of registers constrains the number of channels that can be supported in hardware. This paper presents compile-time techniques that efficiently allocate the channels and successfully exploit the fine-grained parallelism using a small number of channels. The scheduling of operations is carried out in a manner that reduces communication among the processors and hence the number of channels required. Redundant synchronizations subsumed by other synchronizations are eliminated and channels are reused whenever possible. Results of experiments demonstrating the effectiveness of the techniques in utilizing a small number of channels are presented.

Keywords - Fine-grained Parallelism, Instruction Scheduling, Channels, Multiprocessor System.

1. Introduction

Implicit parallelism present in sequential programs is an important source of fine-grained parallelism. This parallelism can be divided into two broad categories, namely loop level parallelism and extra-loop (or non-loop) parallelism. Commercially available multiprocessor systems, such as Encore and Alliant, can exploit loop level parallelism effectively. However, they are ineffective in exploiting extra-loop parallelism present in the sequential parts of a program. The Very Long Instruction Word (VLIW) [2,5] architectures are a family of architectures that can effectively exploit fine-grained parallelism present in sequential parts of a program. The TRACE[1] machine is an example of a commercially available VLIW machine. The compiler for this machine, based upon trace scheduling[3], can detect and schedule extra-loop parallelism in sequential parts of the program and also exploit loop level parallelism by unrolling the loops and converting loop level parallelism into extra-loop parallelism. A VLIW machine consists of multiple processors that operate in lockstep executing instructions fetched from a single stream of long instructions. The synchronization of the processors is guaranteed by the hardware on a per instruction basis. The long instruction word allows initiation of several fine-grained operations in each instruction.

A VLIW machine such as the TRACE has two major disadvantages. First it cannot be used as a multiprocessor as there is a single stream of instructions. The second disadvantage arises due to events unpredictable at compile-time. For example bank access conflicts cannot always be avoided since the operands required for an operation may not be known at compile-time due to the use of arrays and pointers. The lockstep operation of multiple processors makes the machine intolerant to delays caused by unpredictable run-time events. The delay in the completion of any one of the operations in a long instruction delays the completion of the entire instruction.

The Briarcliff Multiprocessor Project is developing a RISC based multiprocessor chip with a small number of processors[4]. The processors on this chip can execute relatively independent streams of instructions for exploiting loop level parallelism and also exploit extra-loop parallelism efficiently. The interconnection poten-

tial of a multiprocessor chip is being exploited to provide highly efficient hardware synchronization mechanisms. The goal of this research project is to develop a multiprocessor system tolerant of unpredictable delays in the progress of individual streams. A synchronization mechanism for achieving this goal during the execution of parallelized loops, named the **fuzzy barrier**[5], has been developed. This mechanism relies upon the compiler to find useful instructions that a processor can execute while it is waiting for other processors to arrive at the barrier. A combination of a mechanism for **collective branching**[6] of processors and data passing mechanism based upon **register channels** is being used to exploit extra-loop parallelism. This paper discusses the work done to ascertain the usefulness of a shared set of register channels [13,14]. A register channel is provided with a synchronization bit that is used to ensure that a processor reading a channel does so only after another processor has written into the channel. Using these registers the processors can communicate at high speed. The communication of values through channels enforces synchronization. The collective branching mechanism enables a single processor to control the execution paths taken by all the processors. The combination of register channels and collective branching allows exploitation of instruction level parallelism without strict lockstep operation of processors. Thus, delays caused by unpredictable operations in a VLIW machine can be potentially reduced.

Techniques such as trace scheduling[3] and region scheduling[8] developed for LIW architectures can also be used to generate code for the Briarcliff architecture. However, additional compile-time techniques to determine when channels should be used must also be developed. The channels are treated as a set of registers that are shared by all the processors. Since the channels are addressed as registers only a limited number of channels can be provided. The goal of this work is to determine whether a small number of channels is sufficient to exploit fine-grained parallelism in a system with a small number of processors. Compile-time techniques that make efficient use of channels by using them only when necessary are developed. These techniques have been applied to straight line code segments taken from a set of scientific programs. The study carried out shows that a small number of channels is sufficient, if used intelligently by the compiler.

The HEP [10,13] multiprocessor provides potentially infinite number of channels by adding a synchronization bit to every location in the shared memory and the register set. This is highly desirable in HEP as the channels are visible to the user at language level. The channels implemented in memory do not allow high speed communication among parallel streams. However, this is not a drawback in HEP as it achieves high throughput by creating a large number of streams and issuing instructions from streams that are ready to execute. The HEP approach is not effective for the Briarcliff architecture. The channels will be used to exploit fine-grained parallelism in a manner similar to VLIW machines; thus the number of streams will at most equal the number of processors in the system. In such a system it is essential to provide fast channels. Register channels are much more efficient than memory channels, especially in a load-store architecture. Another desirable result of using register channels, instead of memory channels, is that contention for shared memory is reduced instead of being increased. Although only a limited number of channels can be provided this is not a drawback for the Briarcliff architecture as the channels are allocated by a parallelizing compiler and are not visible at the language level.

In subsequent sections the operations that can be performed on the channels and theoretical bounds on the number of channels required are presented. Next, instruction scheduling and channel allocation algorithms that attempt to reduce the number of channels required to exploit the parallelism in straight line code are presented. Experimental results demonstrating the effectiveness of the allocation strategies are also presented.

2. Channel Operations

The channels are globally shared among all the processors in the system. Typically at any given point in time a single pair of processors communicate using a channel. The operations that can be performed on the channels are as follows: (i) **Clear** - The channel is cleared by setting the synchronization bit to zero which indicates that the channel is empty. (ii) **Non-destructive Read** - If the channel is empty the reader is blocked till another processor writes to the channel. Once the channel is full the read can take place. The synchronization bit is left unchanged; thus the value can be read again from the channel. (iii) **Destructive Read** - If the channel is full the value is read and the synchronization bit is set to zero indicating that the channel is empty. If the channel is empty the reader blocks till another processor writes to the channel. (iv) **Non-destructive Write** - If the channel is empty the value is written and the synchronization bit is set to one indicating that the channel is full. If the channel is full the writer blocks till the channel becomes empty. (v) **Destructive Write** - The value is written and the synchronization bit is set to one indicating that the channel is full.

At the beginning of a program all channels are empty. To send an one word message from one processor to another the sender uses a non-destructive write and the reader uses a destructive read. However, if the value is to be read by multiple processors the readers use a non-destructive read. On the other hand if a value written to a channel is no longer useful it can be overwritten using a destructive write. By examining the dependency graph for a computation, the operations that can be executed in parallel are determined and then scheduled for execution on different processors. If an operand needed for an operation scheduled on processor p_i is computed by another processor p_j, a channel is used to send this

operand from p_j to p_i. In addition, the channels can also be used to signal the occurrence of events. Consider the situation in which an access to an array element *a[j]* must not precede the assignment to *a[i]* because *i* and *j* may have the same value. A channel can be used to signal the completion of the assignment as opposed to sending a data value. The channels are also useful for executing loop iterations in parallel. Across processor loop carried dependencies can be enforced through channels. In this paper the problem of allocating channels in parallelized loops is not addressed.

3. Bounding the Number of Channels

In this section theoretical lower and upper bounds on the number of channels that should be provided in a p processor system for exploiting parallelism in straight line code are derived. In this analysis N_C denotes the number of channels and N_B denotes the the total number of instructions in the straight line code segment being scheduled on the p processors in the system. If trace scheduling is used by the compiler then for all practical purposes it can be assumed that $N_B \gg p$.

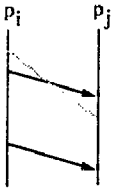

Fig. 1. Lower Bound

Lower Bound: The lower bound on the number of channels is the minimum number of channels needed to correctly execute an instruction schedule generated assuming the availability of channels. A processor in the system may need to send a value to any other processor in the system. Thus, a channel from each processor to every other processor must be provided. The minimum number of channels needed in the system is therefore given by the following:

$$N_C > p(p-1)$$

Next it is shown that a single channel from one processor to another is sufficient for correctly executing any instruction schedule. This can be easily demonstrated as follows. If the values generated by processor p_i for processor p_j are generated in the same order as they are used by p_j then a single channel is sufficient for correct execution. This is because the semantics of the channel guarantees that p_i will not be able to write a new value to the channel till the previous value generated by p_i has been consumed by p_j. If the order in which two values are produced is not same as the order in which they are consumed, then it is always the case that communication of one of the values does not require synchronization; hence the use of a channel. This value can be communicated through shared memory. This is illustrated in Fig. 1. The vertical lines represent the instruction schedules of individual processors and a directed edge from p_i to p_j is used to indicate that a value computed by p_i is used by p_j. The dotted line from p_i to p_j represents a communication that can be carried out without synchronization. Thus the lower bound on the number of channels that should be provided is as follows:

$$N_C \geq p(p-1)$$

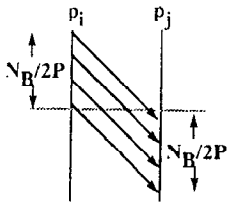

Fig. 2. Upper Bound

Upper Bound: The upper bound on the number of channels is the number of channels that must be provided to ensure that a processor never blocks upon a write to a channel because the channel is not free. To compute the upper bound assume that the operations in the code segment are distributed equally among the p processors. This is because distribution of operations among the processors creates the need for the use of channels. Thus, N_B/p operations are scheduled on each processor. Next we derive the maximum number of channels needed from processor p_i to processor p_j.

The number of channels needed is the maximum if processor p_i continues to produce values and p_j continues to delay their use. This is because each of the unused values will occupy a channel. Fig. 2 illustrates the situation in which maximum number of channels are needed. In each of the first $N_B/2p$ instructions processor p_i computes a value needed by processor p_j. Processor p_j delays using these values till after $N_B/2p$ instructions. Thus each of the values must be assigned a distinct channel. Any value generated by p_i after the first $N_B/2p$ instructions can be communicated without requiring any additional channels. This is because the synchronization constraint due to the communication of this value will cause at least one of the previously enforced synchronizations to become redundant as the edges from p_i to p_j will cross. This situation was shown earlier in Fig. 1. Thus, the upper bound on the number of channels that must be provided is as follows:

$$N_C \leq \frac{N_B}{2p} p(p-1) = \frac{N_B(p-1)}{2}$$

From the above analysis we can estimate the range for the number of channels that should be provided for a system with certain number of processors. The average size of basic blocks in programs is usually small. However, scientific programs tend to have larger basic blocks and techniques such as trace scheduling result in straight line code with higher number of instructions. For basic blocks of size 40 instructions and a four processor system, the number of channels needed lie in the following range:

$$p\ (p-1) \leq N_C \leq \frac{N_B(p-1)}{2}$$

$$12 \leq N_C \leq 60$$

The experimental results presented at the end of paper show that the minimum number 12 was more than adequate, for the sample of programs considered, if the techniques developed in this paper are used.

4. Channel Assignment

The need for the use of channels is dependent upon the instruction schedule. Thus before the assignment of channels can be carried out, the instruction schedule must be generated. A naive approach for generating schedules is list scheduling in which the operations ready to be scheduled are determined and one by one scheduled upon the processors. If the number of processors is greater than or equal to the number of ready operations then all of the ready operations are scheduled. On the other hand if the number of operations ready to be scheduled is higher, the operations that lie on the taller unscheduled paths are scheduled first. The directed acyclic graph (DAG) representing the data dependencies is examined to determine the operations ready to be scheduled and the next set of operations is scheduled. This process is repeated till all operations have been scheduled. The run-time complexity of the list scheduling algorithm is $O(|V|^2)$, where $|V|$ denotes the number of nodes in the dependence graph. Once a schedule has been generated a channel can be assigned every time a value computed by a processor is required by some other processor. This naive approach will serve as a basis for evaluating the performance of an intelligent approach presented next.

In the above approach no attempt was made to minimize the number of channels needed. Next a superior approach consisting of following steps is discussed: (i) **instruction schedule generation** - in a manner requiring fewer channels than list scheduling, without sacrificing the execution speed; (ii) **redundant synchronization elimination** - synchronizations subsumed by other synchronizations are eliminated thus further reducing channel usage; and (iii) **channel assignment** - in a manner that reuses channels whenever possible. The algorithms presented can be used to carry out channel assignment for instruction schedules generated for traces consisting of one or more basic blocks that lie along an execution path. At the points in the control flow graph where two traces meet, compensation code must be inserted to ensure that the channels are in proper state. This is analogous to introduction of code to carry out data movements at the beginning of a trace for a VLIW machine and is handled in a similar fashion[2].

4.1. Top Down Instruction Scheduling

In this section the drawbacks of the list scheduling algorithm are discussed and an alternative scheduling algorithm is developed. The first drawback is illustrated by the example in Fig. 3. The list scheduler may assign different processors to a parent node and each of its children (Fig. 3(i)). In this case a channel will be needed to enforce the data dependency due to each of the children. Without sacrificing any parallelism the parent node can be assigned to one of the processors assigned to its children. This will reduce the number of channels required by one channel (Fig. 3(ii)).

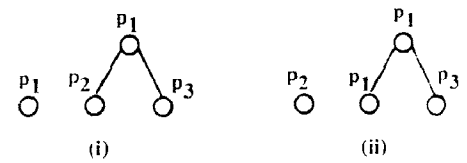

Fig. 3. Processor Assignment

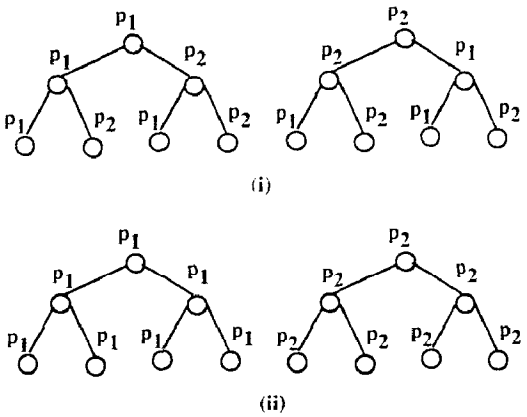

Fig. 4. Top-Down Assignment

For a computation containing more parallelism than the processors in the system can exploit, list scheduling in attempting to exploit this parallelism, may generate schedules requiring a larger number of channels. Consider the processor assignment in Fig. 4(i) found using list scheduling for a two processor system. This assignment requires the use of channels. An equally fast processor assignment shown in Fig. 4(ii) requires no channels at all. To generate assignments of this type the following approach is taken. First of all the scheduling is carried out in a top down fashion instead of the bottom up fashion used by list scheduling. As a result this strategy will generate the last instruction to be executed first and the first instruction to be executed last. An operation is ready to be scheduled if all its parents have been scheduled. Preference is given to nodes with maximum height, where the height of a node is the length of the longest path from the node to the bottom of the DAG. Next, if the number of operations ready to be scheduled is greater than or equal to the number of processors, then several nodes from the subgraphs rooted at these nodes are scheduled on each of the processors. Thus, for the DAG shown in Fig. 4, this will cause entire subgraphs to be scheduled on the same processor. When an entire subgraph is scheduled the operations are scheduled by traversing the graph in a top down and breadth first fashion. A node cannot be scheduled if at the time of scheduling one of its parent nodes has not

been scheduled yet then the node is not scheduled. This situation did not arise in the example presented in Fig. 4 because none of the nodes has more than one parent. By scheduling the operations in the above fashion the number of channels needed is reduced.

One of the advantages of list scheduling is that it tries to distribute the work equally among the processors, which results in fast schedules. It is therefore desirable to incorporate this characteristic into the top down scheduling approach. This can easily be done by ensuring that when entire subgraphs of nodes are being scheduled on the processors, the number of nodes scheduled on each processor equals the number of nodes in the smallest subgraphs. This is illustrated by the example in Fig. 5. The processor assignment shown in Fig. 5(i) results in a poor schedule although it uses no channels. To obtain a faster schedule as shown in Fig. 5(ii), equal number of nodes are scheduled from each subgraph on the two processors. The remaining nodes in the bigger subgraph are then distributed among the two processors. Thus, this scheduling algorithm tries to minimize the number of channels needed without sacrificing the degree of parallelism exploited. The algorithm is summarized in Fig. 6.

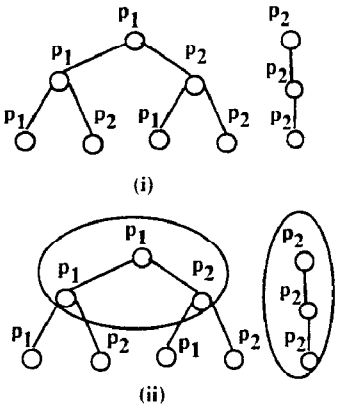

Fig. 5. Equal Distribution

So far it was assumed that all operations take the same amount of time. The algorithm can be easily modified to allow operations requiring variable amount of time. If the operations take varying amounts of time, the height of a node should be defined as the maximum of the number of time units needed to execute each of the paths from the node to the bottom of the DAG. Similarly when scheduling equal amounts of work on each of the processors, the work should be computed in terms of time units and not number of operations.

Run-time Complexity: Computing the heights of all nodes takes $O(|E|)$ time. Updating the status of the nodes to ready also takes $O(|E|)$ time. Maintaining the list of ready nodes sorted according to their heights takes $O(|V| log |V|)$ time. Before choosing a processor on which to schedule a node the algorithm must check if any of the processors on which its parents are scheduled are free or not. This will take at most $O(|E|)$ time. Thus, the run-time complexity of the top down scheduling algorithm is $O(|V|^2)$.

```
Top_down_scheduling
{
 Compute ∀ n_i

 height(n_i) = { 1                                   n_i has no child
               { 1 + max_{n_j, child of n_i} (height(n_j))   otherwise

 loop {
     Construct S = n_1, n_2, ..... , n_m
      st ∀ n_i ∈ S the parents of n_i have been
           scheduled ∧ height(n_i) > height(n_{i+1})

     Let p be the number of processors available
     for (i=1; i≤minimum(p,|S|); i++)
      if possible schedule n_i on processor p_i st p_i has
       one of the parent nodes of n_i scheduled on it
      else choose any available processor;

     if |S|≥p then {
      ∀ n_i, i=1..p schedule a set of operations S_i on p_i
        st ∀ n∈S_i, n∈ subtree rooted at n_i and
           |S_1|=|S_2|=.....=|S_p|
     }
 } until all operations have been scheduled
}
```

Fig. 6. Top-down Scheduling

4.2. Conditions for Reuse of Channels

The conditions under which the same channel may or may not be used for communicating values at different points in the schedule are described next. The use of a channel can be denoted as a pair of operations consisting of a write followed by a read (W_i, R_i), where the write and read operations are performed by different processors. The goal of the channel allocation algorithms is to assign a channel for each such pair of operations and minimize the number of channels used in the process. To minimize the number of channels used, several pairs of write-read's are mapped to the same channel. The following result specifies the condition under which the same channel cannot be used for different operations.

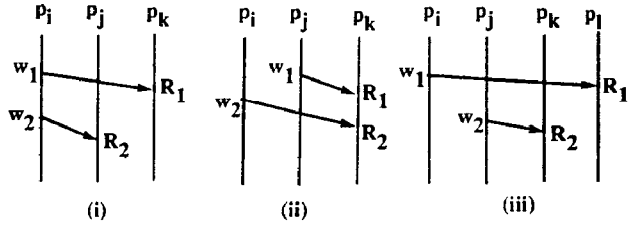

Fig. 7. Situations for Unsafe Sharing

Claim-1: Let (W_1, R_1) and (W_2, R_2) be pairs of operations to which channels have to be assigned. The same channel cannot be assigned to the two operations if the order in which the writes (W_1 and W_2) are performed is not known and/or the order in which the reads (R_1 and R_2)

are performed is not known at compile-time.

Proof: The result is obvious from the cases considered in Fig. 7. In the first case the order of writes is known but the order of reads is not known. If the same channel is used it is possible that after p_i performs W_1 processor p_j may perform R_2 before p_k performs read R_1. Thus, the value meant for processor p_k will be consumed by p_j. Similarly by examining the other two cases one can see that the same channel cannot be used for both (W_1,R_1) and (W_2,R_2). □

Claim-2: If the order in which the reads (R_1 and R_2) are performed and the order in which the writes (W_1 and W_2) are performed is known precisely, then either the same channel can be assigned for both (W_1,R_1) and (W_2,R_2) or one of the operations does not require synchronization and hence the use of channel.

Proof: There are two possible orderings for the operations: (i) W_1 precedes W_2 and R_1 precedes R_2; or (ii) W_1 precedes W_2 and R_2 precedes R_1. In the first case the same channel can be used for both (W_1,R_1) and (W_2,R_2), since this will guarantee that the order in which the operations will occur is $W_1R_1W_2R_2$. In the second case channels need not be assigned for both (W_1,R_1) and (W_2,R_2). The order in which the operations should occur is $W_1W_2R_2R_1$. Thus, if a channel is assigned to guarantee the order W_2R_2 the ordering W_1R_1 is automatically guaranteed. □

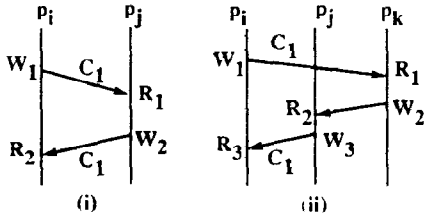

Fig. 8. Conditions for Safe Reuse

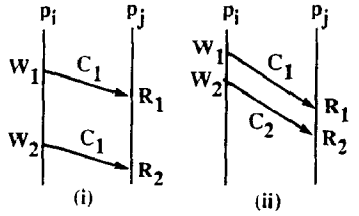

Fig. 9. Efficient Execution Time

Although the same channel can be reused if the order in which reads and writes are performed is known precisely, it may not always be advisable to do so. This is illustrated by the examples in Fig. 9. Although the semantics of the channel will guarantee that R_1 is performed before W_2 it is desirable for processor p_j to have performed R_1 when processor p_i performs W_2 so that the execution of instructions by p_i is not blocked. If the example in Fig. 9(i) is compared with the example in Fig. 9(ii) it can see that it is highly likely for processor p_i to block in the second case if the same channel is reused. Thus, for fast execution two channels should be used in the second case.

4.3. Eliminating Redundant Synchronizations

As mentioned earlier in the proof for *Claim-2*, every time a processor generates a value for another processor a channel may not be needed. If the processor using the value is guaranteed to read the value after it has been generated by the other processor then the value can be transmitted through shared memory without explicitly synchronizing the two processors. This is illustrated by the examples presented in Fig. 10. In both cases R_1 is guaranteed to occur after W_1 if the orderings for the other write and read operations are enforced. Before channels are actually assigned, the instruction schedules can be examined to eliminate those cross-processor dependencies that are automatically ensured if the remaining dependencies are enforced using channels.

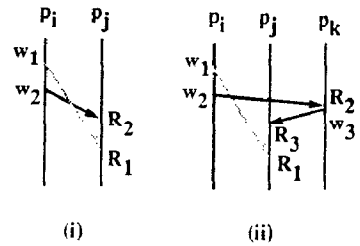

Fig. 10. Redundant Synchronizations

The elimination of the redundant synchronizations can be carried out in any order. This is due to the following result. Let $(W_i,R_i) \longrightarrow (W_j,R_j)$ denote that guaranteeing the write before read order for (W_i,R_i) automatically guarantees the write before read order for (W_j,R_j). The relation $\longrightarrow$ is transitive i.e.,

$$((W_1,R_1) \longrightarrow (W_2,R_2)) \wedge ((W_2,R_2) \longrightarrow (W_3,R_3)) \Rightarrow ((W_1,R_1) \longrightarrow (W_3,R_3)).$$

Thus, the order in which (W_2,R_2) and (W_3,R_3) are eliminated has no bearing on the final outcome.

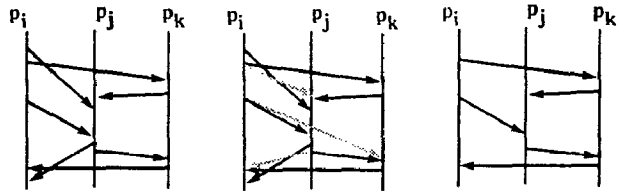

Fig. 11. Removal of Redundant Synchronizations

The algorithm for the removal of redundant synchronizations consists of three steps. In the first step a graph is constructed, the nodes of which are the nodes from the DAG. The edges in the graph represent the order in which the operations must be performed to ensure cross-processor dependencies. In addition the nodes scheduled on the same processor are also connected by edges to indicate the order in which they will be executed. In the second step the graph constructed is traversed to determine for each operation node n

scheduled on a processor, the earliest instructions in the schedules for the other processors that must wait for the completion of n. This information essentially represents additional synchronizations referred to as implied synchronizations, that are guaranteed if the cross-processor dependencies are enforced. Finally the above information is used to eliminate the redundant synchronizations. This is achieved by inspecting a cross-processor dependency and determining if it is automatically enforced by another dependency in which case it can be eliminated. The result of its application to an example is illustrated in Fig. 11. The run-time complexity of this algorithm is $O(|E_C|+|V|)$, where $|E_C|$ is the number of cross-processor dependencies and $|V|$ is the number of nodes in the graph.

4.4. Channel Assignment Strategy-1

In this section and the next, two algorithms for channel allocation are presented. These algorithms try to minimize the number of channels used by reusing the channels and assume that additional channels are always available if required. After presenting these algorithms it is shown how the algorithms can be adapted to function for a fixed number of channels.

```
Channel_Assignment
{
 Assign(n,channel_num)
 {
 if ∃ n_i st edge n→n_i is yet to be assigned a channel {
     assign channel_num to edge n→n_i; let n_j be the next
     operation executed by the processor that executes n_i
     Assign(n_j,channel_num);
   }
 else { let n_j be the next operation node to be executed
        by the processor that executes node n
       Assign(n_j,channel_num);
     }
 }

 channel_num = 0;
 ∀ operation nodes n
 if ∃ an edge that from node n that should be assigned a
   channel and has not been assigned yet {
    channel_num = channel_num + 1;
    Assign(n,channel_num);
  }
}
```

Fig. 12. Channel Assignment Algorithm

The algorithm presented here allocates channels in such a way that a channel is reused only if it can be guaranteed that at the point of reuse the channel will be free. Thus, it is guaranteed that at run-time a processor writing to a channel never blocks due to the channel being full. To ensure this, the same channel is allocated for (W_i,R_i) and (W_j,R_j) if and only if, the precise orderings for the reads and writes are known and the writes are not performed by the same processor. The algorithm takes one channel at a time and tries to resolve as many non-redundant cross-processor dependencies as possible. This process is repeatedly employed using additional channels till all dependencies have been enforced. The algorithm is summarized in Fig. 12. The procedure *Assign* assigns a given channel to enforce as many dependencies as possible.

Run-time Complexity of Strategy-1: Assume that the $|V|$ operations in the DAG are evenly distributed among p processors. Let $|E_C^R|$ denote the number of cross-processor dependencies, after removal of redundant synchronizations, for which channels have to be assigned. In each step of the algorithm a channel is assigned to enforce as many cross-processor dependencies as possible and in the process $|V|/p$ instructions are examined. In the worst case only single dependency will be resolved in each step and thus the overall run-time complexity of the algorithm will be $O(|E_C^R||V|/p)$.

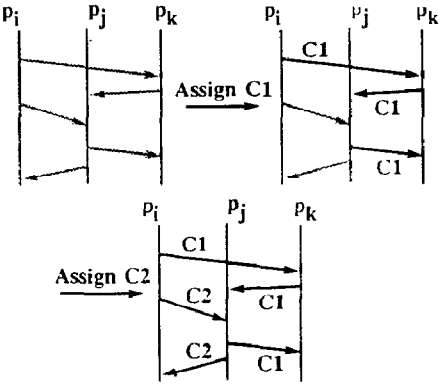

Fig. 13. Channel Assignment - An Example

4.5. Channel Assignment Strategy-2

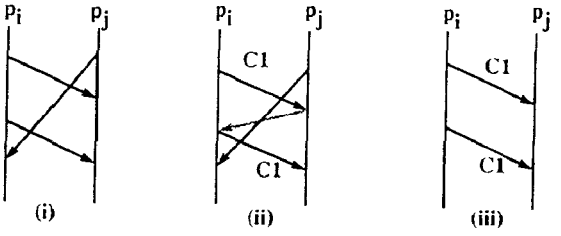

Fig. 14. Implied Synchronizations

As channels are assigned additional synchronizations due to resource usages are introduced. These additional synchronizations may make some of the other synchronizations for cross-processor dependencies redundant. In Fig. 14(i) there are three dependencies that must be enforced through channels. If channel $C1$ is assigned to enforce the two dependencies from p_i to p_j, as shown in Fig. 14(ii), an additional synchronization shown by a dotted edge is implied. This causes the synchronization due to dependency from p_j to p_i to become redundant.

The algorithm presented in this section takes advantage of the above observation. The allocation of channels to enforce dependencies for each ordered pair of processors is carried out one at a time. After channels have been assigned to an ordered processor pair (p_i,p_j),

synchronizations implied by resource usage are computed. Next the synchronizations that have not been assigned channels yet and are now redundant are removed. The algorithm is summarized in Fig. 15. It should be noted that the order in which channel allocation is carried out influences the number of channels used. This is because the synchronizations made redundant due to reuse of channels cannot be known till some amount of allocation has already been done. It is also the case that as channels are allocated to enforce dependencies, some of the dependencies for which channels had already been assigned may become redundant. However, it may not be possible to deallocate channels at this stage.

Run-time Complexity of Strategy-2: In this algorithm, as the allocation of channels is carried out additional edges are added to the graph constructed for removing redundant synchronizations. Let $|E_C^R|$ be the number of cross-processor edges. Since an edge is added every time two cross edges are assigned to the same channel, at most $|E_C^R|-1$ additional edges may be added. Updating the graph constructed for the elimination of redundant synchronizations takes $O(p|E_C^R|)$ time, $O(p)$ time for each edge. Removing redundant synchronizations in all takes $O(|E_C^R|+|V|)$ time. Finally, the assignment of channels to cross-processor edges takes $O(|E_C^R|)$ time. Thus, the overall run-time complexity of the algorithm is $O(|V|+p|E_C^R|)$.

```
Strategy-2
{
 for each ordered pair of processors (p_i,p_j) st instructions
 executed by p_j are dependent upon instructions executed
 by p_i and enforcing these dependencies requires channels
  {
   Remove all redundant dependencies of p_j on p_i.
   ∀ dependence edges from p_i to p_j assign
    channels chosen as follows:
    {
     Let n_i ⟶ n_j be the edge under consideration
      If one of the instructions preceding instruction n_i
      in the schedule for p_i reads from a channel and
      since then it has not been allocated to be written by
      either processor p_i or p_j then it can be
      used to enforce the dependency n_i ⟶ n_j.
     If none of the already used channels can be assigned
      allocate a new channel.
    }
   Introduce additional synchronizations guaranteed by the
    assignment of channels just carried out.
  }
}
```

Fig. 15. Strategy-2

4.6. Allocating a Fixed Number of Channels

The algorithms presented in the preceding sections assumed that there is an unlimited number of channels available. However, in practice the number of channels will be fixed by a specific hardware implementation. Next it is shown how the above algorithms can be applied even if the number of channels is fixed. In deriving the lower bound for the number of channels that a system must support, it was shown that as long as there is a single channel dedicated from each processor to every other processor, any schedule can be correctly executed. The same idea is used to ensure that all dependencies can be enforced using a fixed number of channels.

The total number of channels is divided into two groups *Unconstrained* and *Constrained*. The number of channels in the *Constrained* set is the number of ordered pairs of processors that require the use of a channel due to cross-processor dependencies. This is the minimum number of channels needed to enforce all dependencies. The remaining channels are put in the *Unconstrained* set. The channel assignment algorithm allocates channels from the *Unconstrained* set and attempts to resolve as many dependencies as possible. During this process, if all dependencies for an ordered processor pair get resolved then the channel reserved for this pair in the *Constrained* set can be moved to the *Unconstrained* set. The channels are allocated until either all dependencies have been resolved or the *Unconstrained* set is empty. In the latter case it is guaranteed that the *Constrained* set will have enough channels to resolve the remaining dependencies. The algorithm is summarized in Fig. 16.

```
Allocate_Fixed
{
 Constrained = Set of channels st one channel is reserved for
  every ordered pair (p_i,p_j) of processors st there are
  cross-processor dependencies from p_i to p_j. This can
  be at most p(p-1), where p is the number of processors.

 Unconstrained = rest of the channels

 Applying the Channel Assignment algorithm continue to
 allocate channels from the Unconstrained set. If
 all channel requirements for an ordered processor pair
 have been fulfilled then remove a channel from the
 Unconstrained set and include it in the
 Constrained set.

 If Unconstrained set is empty and all edges have
 not been allocated channels then assign the channels
 from the Constrained set as follows:
  Choose a channel from Constrained set and assign
  it to all the edges of an ordered processor pair.
  The number of channels should be exactly the number of
  ordered processor pairs that still have an edge requiring
  a channel.
}
```

Fig. 16. Allocating Limited Number of Channels

Assignment of the same channel to enforce all remaining dependencies from one processor to another in the final step of the above algorithm will result in schedules that may execute slower as a processor may have to wait between performing successive writes to the channel. Another approach for the allocation of a fixed number of channels is to modify the schedule so that it requires no more than the available number of channels.

The latter approach is unlikely to perform better than the suggested approach because to generate a schedule that requires fewer channels some parallelism will be left unexploited. Scheduling of operations on the same processor instead of different processors not only reduces the number of channels required but also the degree of parallelism exploited.

5. Experimental Results

Instruction schedules and channel allocations were carried out for a sample of data dependency graphs obtained from real programs. These graphs had been constructed by Rodeheffer[12] for the inner loops of a set of scientific programs. Use of scientific programs is more appropriate for obtaining data dependency graphs because in general they tend to have longer basic blocks and hence are likely to require greater number of channels. Furthermore, these dependence graphs were constructed by coverting control dependencies into data dependencies to construct longer sequences of unconditional code. Techniques to do so were developed by Kuck *et al.*[11] The test programs include the following:

ALG1 - The Generalized Eigenvalue Problem.
ALG2 - Solving Linear Equations Using Residue Arithmetic.
ALG3 - Solution of Ordinary Differential Equations.
ALG4 - Discrete Chebychev Curve Fit.
ALG5 - Evaluation of Normalized Taylor Coefficients.
ALG6 - Calculation of Fourier Integrals.
ALG7 - Exact Cumulative Distribution of the Kolmogorov-Smirnov Statistic for Small Samples.

The results of the experiments conducted demonstrate the effectiveness of the instruction scheduling and channel allocation strategies presented in this paper. The results in Table 1 provide a comparison between the naive approach and the efficient approach. In the naive approach the number of cross-processor dependencies is large and equal to the number of channels used. On the other hand the use of top-down scheduling results in a significantly smaller number of cross-processor dependencies (#DEPS). The use of efficient channel assignment strategy (strategy-1) further reduces the number of channels (#CHAN) needed by reusing them whenever possible. The naive approach used up to 19 channels while the efficient approach employed a maximum of 7 channels. The number of channels used is fairly small which makes it feasible to implement channels that are addressed and accessed as registers.

Next the performance of the top-down scheduling is compared with that of list scheduling. First let us compare the quality of schedules generated by the two scheduling techniques in terms of the parallelism exploited. In Table 2 the total number of nodes (N_B) and the number of nodes along the longest path (N_L) in each DAG are given. Assuming that each operation takes unit time, the fastest possible schedule for a DAG on a p processor system is equal to $\max(\lceil N_B/p \rceil, N_L)$. By examining the execution times for the schedules generated by both the scheduling strategies it can be seen that not

Table 1: Number of Channels

	NAIVE	EFFICIENT	
	#CHAN	#DEPS	#CHAN
ALG1BB1	1	3	2
ALG1BB2	4	2	2
ALG2BB1	2	1	1
ALG2BB2	9	5	4
ALG2BB3	5	3	3
ALG3BB1	3	3	2
ALG4BB1	8	6	5
ALG4BB2	12	5	4
ALG5BB1	19	10	7
ALG6BB1	17	9	5
ALG6BB2	3	3	3
ALG6BB3	1	1	1
ALG6BB4	0	0	0
ALG6BB5	6	5	3
ALG7BB1	4	3	2

only do both strategies perform equally well, but they also generated the fastest possible schedules for a four processor system. Next the channel assignment algorithm was applied to the schedules generated by both the scheduling strategies. A maximum of ten channels were used for the schedules generated by list scheduling which is higher than the maximum number of seven channels required for the schedules generated using top top-down scheduling. Thus, top-down scheduling generates schedules that are not only as fast as the schedules generated using list scheduling but also require fewer channels.

Table 2: Expected Execution Times (p=4)

	N_B	N_L	LIST_SCHED		TOP_DOWN_SCHED	
			TIME	#CHAN	TIME	#CHAN
ALG1BB1	13	7	7	2	7	2
ALG1BB2	11	7	7	3	7	2
ALG2BB1	18	12	12	1	12	1
ALG2BB2	17	6	6	5	6	4
ALG2BB3	15	7	7	4	7	3
ALG3BB1	26	9	9	2	9	2
ALG4BB1	12	5	5	5	5	5
ALG4BB2	20	7	7	6	7	4
ALG5BB1	41	11	11	10	11	7
ALG6BB1	30	7	8	8	8	5
ALG6BB2	13	7	7	3	7	3
ALG6BB3	8	6	6	1	6	1
ALG6BB4	30	12	12	0	12	0
ALG6BB5	14	7	7	3	7	3
ALG7BB1	15	5	5	6	5	2

An alternative approach for implementing channels is to provide dedicated channels from each processor to every other processor. This is easier to implement in hardware because a channel is no longer globally accessible to all processors. By introducing a queue of fixed length, effectively multiple channels can be provided between a pair of processors. The allocation of such channels is a trivial task. The schedules for the test cases were analyzed and it was found that a queue

length of four channels would have been sufficient. Further reduction in the number of channels of this kind may be possible because the top-down scheduling algorithm tries to reduce the overall number of channels and not the number of channels between a pair of processors.

6. Conclusion

This paper explored the possibility of employing channels implemented as registers with synchronization bits, to exploit fine-grained parallelism in sequential programs. Compile-time techniques for allocation of such a resource were developed. The results of experiments performed show that a small number of channels are sufficient to exploit parallelism in code segments of significant size. The use of channels will provide improvement in performance over VLIW machines as the multiple processors are no longer constrained to execute in lockstep.

References

1. R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, and P.K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Proc. Second International Conf. on Architectural Support for Programming languages and Operating Systems*, pp. 180-192, 1987.
2. J.R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1986.
3. J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. on Computers*, vol. 7, no. C-30, pp. 478-490, July, 1981.
4. R. Gupta, M. Epstein, and M. Whelan, "The Design of a RISC based Multiprocessor Chip," *Technical Note TN-89-151, Philips Laboratories, Briarcliff Manor, NY*, 1989.
5. R. Gupta, "The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors," *Proceedings of the Third International Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 54-64, April, 1989.
6. R. Gupta and M. Epstein, "Collective Branching in a MIMD System," *Technical Note TN-89-013, Philips Laboratories, Briarcliff Manor, NY*, 1989.
7. R. Gupta and M.L. Soffa, "A Reconfigurable LIW Architecture," *Proc. of the International Conf. on Parallel Processing*, pp. 893-900, August, 1987.
8. R. Gupta and M.L. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *to appear IEEE Transactions on Software Engineering.*
9. R. Gupta and M.L. Soffa, "Compilation Techniques for a Reconfigurable LIW Architecture," *The Journal of Supercomputing*, vol. 3, pp. 271-304, 1989.
10. J.S. Kowalik, Editor, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, MIT Press, 1985.
11. D.A. Padua, D.J. Kuck, and D. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. on Computers*, vol. 29, no. 9, pp. 763-776, 1980.
12. T.L. Rodeheffer, "Compiling Ordinary Programs for Execution on an Asynchronous Multiprocessor," Ph.D. Dissertation, Carnegie-Mellon University, 1985.
13. B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Real-Time Signal Processing*, vol. 298, pp. 241-248, August, 1981.
14. J.A. Solworth, "The Microflow Architecture," *Proc. of the International Conference on Parallel Processing*, vol. I, pp. 113-117, August, 1988.