Executing Loops on a Fine-Grained MIMD Architecture

Sunah Lee
slee@cs.pitt.edu

Rajiv Gupta [1]
gupta@cs.pitt.edu

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract - We present techniques for exploiting parallelism extracted from loops on an MIMD system. Parallelism is exploited through parallel execution of instructions on multiple processors as well as pipelined nature of individual processors. The processors based upon the load/store architecture read/write operands from/to private registers, shared registers, and channel queues. If the communication of a value from one processor to another requires synchronization then a channel is used otherwise a shared register is used to communicate the value. The receiving processor reads the values from a channel queue in the order they are written to the channel by the sending processor. The scheduling of operations is carried out in a manner that reduces interprocessor communication. Such schedules reduce the likelihood of one processor impeding the progress of other processors.

1. Introduction

Implicit parallelism present in loops is an important source of fine-grained parallelism. In this paper we present a tightly coupled fine-grained MIMD architecture whose processors can execute relatively independent streams of instructions as well as tightly synchronized instruction streams. The system is designed using a traditional RISC processor augmented with multiprocessor support. Fine-grained parallelism is exploited by executing multiple instructions in parallel on different processors as well as overlapped execution of instructions on pipelined processors. Globally shared registers and dedicated channel queues are provided which allow the processors to exchange data at high speed are provided. If no synchronization is required during the communication of a data value from one processor to another, then a globally shared register is used to communicate a data value, or else the channel queue from the sending processor to the receiving processor is used to communicate the data value. An MIMD system is tolerant of delays caused by unpredictable events as such memory access conflicts.

[1] Partially supported by an NSF Presidential Young Investigator Award CCR-9157371 to the University of Pittsburgh.

The Very Long Instruction Word (VLIW) architectures are a family of architectures that can effectively exploit fine-grained parallelism.[3,5] The processors in a VLIW machine operate in lockstep. The synchronization of the processors is guaranteed by the hardware on a per instruction basis. A value computed by a processor in one instruction is accessible to the other processors in the next instruction. The data values are communicated among the processors through shared registers. A VLIW machine is unsuitable for the execution of independent instruction streams. In addition, the lockstep operation of multiple processors makes the machine intolerant to run-time delays caused by unpredictable events. The delay in the completion of any one of the operations in a long instruction delays the completion of the entire instruction. Unlike a VLIW machine, the MIMD system is suitable for executing independent instruction streams and it is also tolerant of delays caused by unpredictable events. Also the MIMD system can be designed using an existing RISC design thus providing binary compatibility with existing machines. On the other hand the instruction set of a VLIW machine is not compatible with existing RISC processors.

The compilation techniques developed for VLIW machines, such as trace scheduling,[4] region scheduling,[8] percolation scheduling,[1] software pipelining,[12] and optimal loop parallelization[2] can also be used to generate code for the fine-grained architecture. However, the above techniques must be adapted to take advantage of the MIMD nature of the system. During the distribution of instructions among the processors an attempt should be made to minimize the synchronization of processors. This is because frequent processor synchronization can potentially result in run-time delays as well as reduce the effectiveness of a processor's pipeline. In addition, we present compilation techniques that are required to take advantage of channel queues to enforce data dependencies within loop iterations as well as data dependencies across loop iterations. Several values being sent from one processor to another can simultaneously reside in a channel queue. Thus, in the presence of an interprocessor loop carried dependency the values being communicated from one processor to another are written to the same channel. Thus, parallelism across loop iterations is exploited without unrolling the loop.

In the next section a brief description of the fine-grained MIMD architecture is presented. In subsequent sections techniques are presented for distinguishing between situations in which shared registers can be used for communicating data values among processors and situations in which channel queues must be used for the communication of data values. Results of some experiments that demonstrate the feasibility of

a channel based architecture are presented.

2. The Architecture

The pipelined RISC processors in the MIMD system are augmented with multiprocessor support. An operand involved in the execution of an instruction is read/written from/to the executing processor's private register, a register globally shared among all of the processors, or a channel queue to/from the executing processor and some other processor in the system. The operand specification in an instruction consists of a couple of bits to distinguish between the three types of operand sources/destinations. The remainder of the bits specify a particular private register, shared register, or a channel queue.

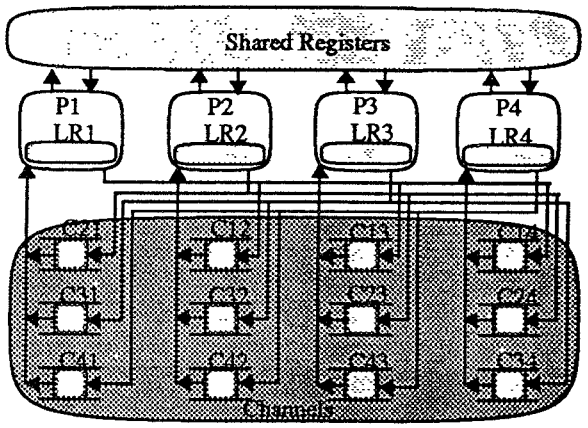

Figure 1: MIMD System based upon Channel Queues.

From each processor to every other processor a channel queue is provided. A receiving processor can read a channel queue only after the sending processor has written a value to the channel. A hardware counter associated with each queue indicates whether the queue is empty or not. The hardware stalls a processor if it is either attempting to read an empty queue or write to a full channel. The channels are organized as queues because through appropriate compilation techniques we ensure that the order in which the values are read by a receiving processor is the same as the order in which they are written to the channel by the sending processor.

The processors can communicate with each other through the shared registers and channel queues. When a value is communicated through a shared register, the synchronization of processors is not guaranteed by the hardware. Therefore, it is possible for a processor to incorrectly read a value from a shared register before it has been written to the register. On the other hand the hardware guarantees synchronization if a data value is communicated through a channel queue. The compiler, through its analysis of the parallel instruction schedules, distinguishes the situations in which channel queues must be used from the situations in which shared registers should be used. Since the synchronization of processors during the communication of values among processors is ensured by channel queues, unlike VLIW systems, the processors are no longer required to operate in strict lockstep fashion. Thus, the MIMD system is tolerant of delays that otherwise would be introduced by unpredictable events. In addition to channel queues the fuzzy barrier mechanism[6] can be provided for efficient synchronization of all processors and the collective branching mechanism[9] can be provided for efficient implementation of branches through sharing of condition code.

The channel queues can be used to enforce all types of interprocessor data dependencies. There are three types of data dependencies *flow*, *anti*, and *output*.[11] As shown below a flow dependency represents a true data dependency and an antidependence as well as an output dependence are storage related dependencies. In the case of a flow dependence the value is communicated through the channel queue. In the case of the other two types of dependencies the channel queue is simply used to synchronize the processors and no useful data value is communicated through the channel.

flow	*anti*	*output*
X = ..	.. = X	X = ..
.. = X	X = ..	X = ..

In some situations the presence of a dependency may not be predictable at compile-time. For example, consider the situation in which a definition of A[i] is followed by the use A[j] and the corresponding store and load operations are scheduled on different processors. The presence of a dependency depends upon whether the values of i and j are the same. The channel is used to synchronize the processors so that the store operation is performed before the load operation. However, no useful data value is communicated through the channel.

3. Compiler Support for Utilizing Channel Queues

Completely parallel doall loops can be executed on an MIMD system by distributing loop iterations among the processors. However, if the loop iterations can be simultaneously executed the fine-grained parallelism present within each loop iteration and across loop iterations can be exploited. A technique developed by Aiken and Nicolau[2] can be used to transform loops to expose parallelism present across loop iterations. After applying Aiken and Nicolau's transformation fine-grained parallelism can be detected by constructing a directed acyclic graph (DAG) representing the data dependencies among the statements in the loop body. Each processor is provided with a private copy of the loop variable. The branch corresponding to the loop back edge is executed independently by each processor and hence the loop predicate is tested by each processor.

After the detection of fine-grained parallelism the compiler must perform the following steps to generate code for the fine-grained MIMD architecture.

(i) A parallel execution schedule is generated. The schedule should exploit the MIMD nature of the system. By scheduling parallelism such that there are fewer interprocessor data dependencies we improve the performance of the processor pipeline.

(ii) The interprocessor data dependencies, including dependencies across loop iterations, must be resolved through the shared registers and channel queues. We assume that the iteration distances of all dependencies are constants which are known at compile-time.

(iii) The size of each channel queue is fixed. Thus, delays can be caused if an attempt is made to write to a channel that is full. Techniques are required to anticipate and avoid such delays.

3.1. Instruction Scheduling

The top-down instruction scheduling algorithm that generates schedules for exploiting parallelism with low interprocessor communication was developed in previous work.[7] Consider an operation in a DAG that receives its two operands from two other operations in the same DAG. The operation requiring the two operands can be assigned to one of the processors assigned to the operations that compute the operands. This will reduce interprocessor dependencies. For a computation containing more parallelism than the processors in the system are able to exploit, schedules involving fewer interprocessor data dependencies can be generated by carrying out scheduling in a top down fashion. 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. The number of nodes scheduled on each processor equals the number of nodes in the smallest subgraphs. By attempting to schedule the same number of operations on each processor good load balancing and hence better speedups can be expected. Thus, this scheduling algorithm tries to minimize the number of channels needed without sacrificing the degree of parallelism exploited. By reducing interprocessor dependencies, opportunities for reordering the code scheduled on a processor to reduce pipeline delays are created. Instruction reordering techniques developed by Gross and Hennessy[10] can be used for reordering the group of statements that are simultaneously assigned to a given processor at the same time.

In Figure 2 we show a loop which is first transformed using Aiken and Nicolau's algorithm to expose parallelism. The schedule *Sched1* is generated using top-down scheduling for the execution of the loop on two processors. If we examine the schedule *Sched1* we can see that there is no interprocessor data dependency with iteration distance zero. This will avoid the processor pipelines from being underutilized due to interprocessor communication. In addition we can transform *Sched1* to *Sched2* which separates intraprocessor dependencies apart which further reduces the likelihood of pipeline delays.

3.2. Selecting the Mode of Interprocessor Communication

After a schedule has been generated the compiler must identify all situations requiring interprocessor communication and then generate code to establish communication through either channel queues or shared registers. In this section we present compile-time techniques that enable us to make the appropriate choice. If the processor receiving the value is guaranteed to read the value after the value has been computed, then a shared register is used. If this is not the case and synchronization is required, then the appropriate channel queue is used since it guarantees read after write. Next we derive results that enable us to ascertain the need for synchronization. These results essentially identify conditions under which one synchronization subsumes another synchronization,

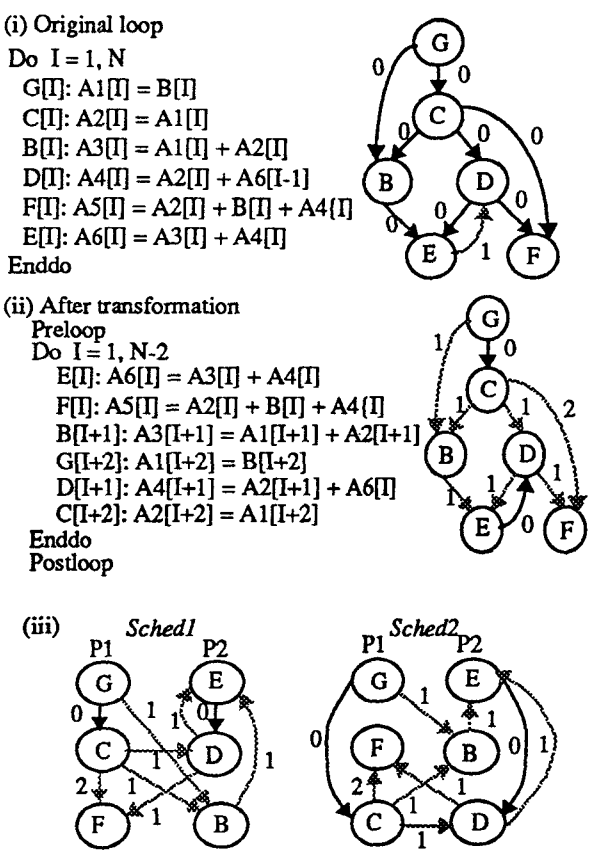

Figure 2: Instruction Reordering.

i.e., makes the latter unnecessary. We assume that the iteration distances of all interprocessor dependencies are constants that are known at compile-time.

Lemma 1: Given two flow dependence edges e_1 and e_2 from processor p_1 to processor p_2 with the same iteration distances. The synchronization for the flow dependence e_1 subsumes the dependence e_2 if and only if $t_{src}(e_1) > t_{src}(e_2)$ and $t_{dest}(e_1) < t_{dest}(e_2)$.

Proof: There are only two possibilities to consider here. Either the two dependence edges e_1 and e_2 intersect or they do not intersect.

Case I: $t_{src}(e_1) > t_{src}(e_2) \wedge t_{dest}(e_1) < t_{dest}(e_2)$

If the edges intersect (see Figure 3(i)) then it is clear that enforcing the dependence e_1 guarantees that e_2 is also enforced. Therefore, e_2 is subsumed by e_1.

Case II: $t_{src}(e_1) < t_{src}(e_2) \wedge t_{dest}(e_1) \leq t_{dest}(e_2)$

If the two edges do not intersect (see Figure 3(ii)) then the synchronization is clearly required to enforce the two dependencies. Consequently the result stated above follows. □

Lemma 2: Given a flow dependence edge e_1 with iteration distance d and another flow dependence edge e_2 with iteration distance $d+1$ from processor p_1 to processor p_2. The synchronization for flow dependence e_1 subsumes the synchronization requirement for the dependence e_2 unless the condition $(t_{src}(e_1) < t_{src}(e_2) \wedge t_{dest}(e_1) > t_{dest}(e_2))$ is true.

Proof: In order to derive the above result we consider following cases which arise from the relationships between t_{src} and

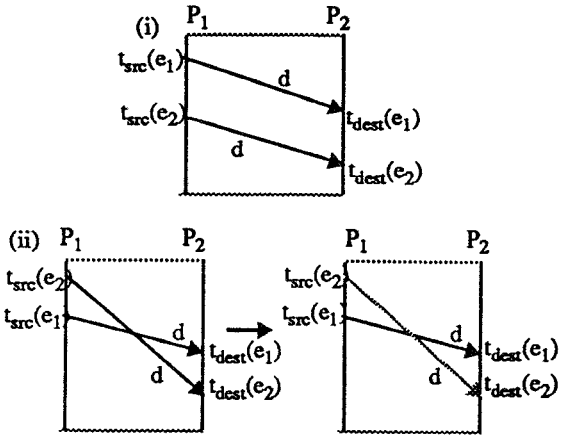

Figure 3: (i) Non-intersecting Dependence Edges; (ii) Intersecting Dependence Edges.

t_{dest} values for edges e_1 and e_2.

Case I: $t_{src}(e_1)<t_{src}(e_2) \wedge t_{dest}(e_1)<t_{dest}(e_2)$ (Figure 4).

Case II: $t_{src}(e_1)>t_{src}(e_2) \wedge t_{dest}(e_1)\geq t_{dest}(e_2)$ (Figure 5).

Case III: $t_{src}(e_1)>t_{src}(e_2) \wedge t_{dest}(e_1)\leq t_{dest}(e_2)$ (Figure 6).

Case IV: $t_{src}(e_1)<t_{src}(e_2) \wedge t_{dest}(e_1)>t_{dest}(e_2)$ (Figure 7).

By considering two successive loop iterations we can view e_2, originally with iteration distance of $d+1$, as a dependency between the loop iterations with iteration distance d (see Figures 4(ii)-7(ii)). Now the dependencies e_1 and e_2 both have the same iteration distance and therefore lemma 1 can be used to determine the condition under which e_1 subsumes e_2. This analysis yields us to the conclusion that in the first three cases e_1 subsumes e_2. Thus, e_1 does not subsume e_2 if the condition $t_{src}(e_1)<t_{src}(e_2) \wedge t_{dest}(e_1)>t_{dest}(e_2)$ is true. □

Lemma 3: Given a flow dependency from processor p_1 to p_2 with iteration distance d. The synchronization that enforces the given dependency also enforces (i.e., subsumes) any synchronization from p_1 to p_2 with iteration distance greater than $d+1$.

Proof: In order to prove this result we consider two dependencies e_1 and e_2 of iteration distance d and $d+2$. There are four possible relationships between the two dependencies as mentioned in lemma 2. By considering two successive iterations we can view e_2 as a dependence with iteration distance $d+1$. Next by applying lemma 2 we can easily show that in all four cases e_1 subsumes e_2. □

Theorem 1: A synchronization introduced for enforcing a flow dependency e_i with iteration distance d from processor p_1 to p_2 subsumes the synchronization required for enforcing a flow dependency e_j of iteration distance d', also from p_1 to p_2, if and only if one of the following conditions is true:

- $d'=d \wedge t_{src}(e_i)>t_{src}(e_j) \wedge t_{dest}(e_i)<t_{dest}(e_j)$
- $d'=d+1 \wedge not\,(t_{src}(e_i)<t_{src}(e_j) \wedge t_{dest}(e_i)>t_{dest}(e_j))$
- $d'>d+1$

Proof: This theorem follows directly from lemmas 1, 2, and 3. □

So far we have only considered dependencies between pairs of processors. Introduction of synchronizations between pairs of processors creates additional synchronizations between other pairs of processors. Such a synchronization is

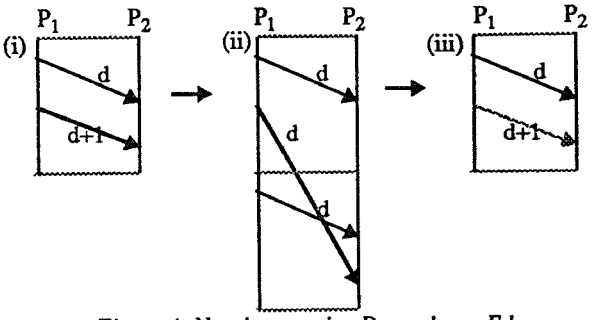

Figure 4: Non-intersecting Dependence Edges of Iteration Distances d+1 and d.

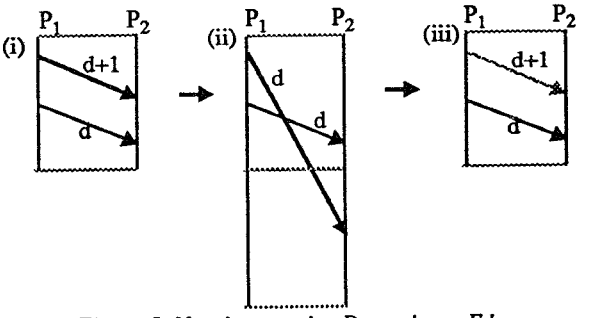

Figure 5: Non-intersecting Dependence Edges of Iteration Distances d and d+1.

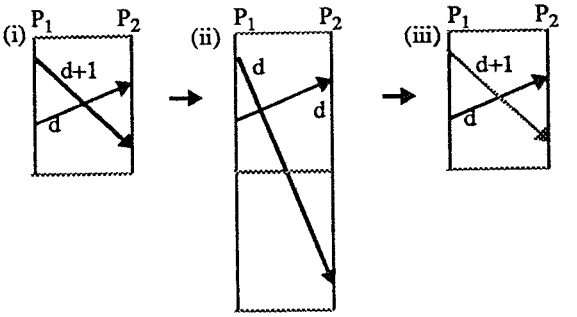

Figure 6: Intersecting Dependence Edges of Iteration Distances d and d+1.

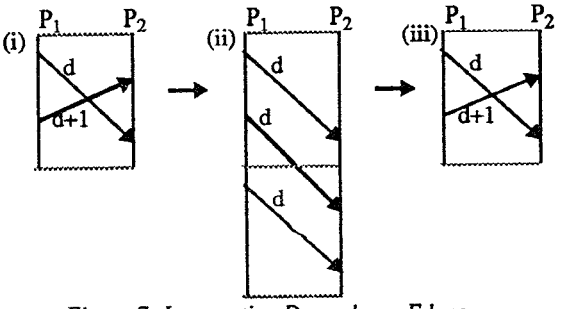

Figure 7: Intersecting Dependence Edges of Iteration Distances d+1 and d.

called an *implied synchronization* since it is not explicitly introduced in the code. For example, introduction of synchronization from p_1 to p_2 and p_2 to p_3 implies a synchronization between p_1 and p_3. The computation of implied synchronizations is necessary to determine all interprocessor data dependence edges which do not require explicit synchronization. The following result specifies the computation of implied syn-

chronizations.

Theorem 2: Given a sequence of flow dependence edges e_1, e_2, ..., e_n with iteration distances of d_1, d_2, ..., d_n respectively. An edge e_i represents a flow dependency from processor p_{i-1} to processor p_i and $t_{dest}(e_i) \leq t_{src}(e_{i+1})$. The introduction of synchronization instructions to enforce the sequence of dependencies e_1, e_2, ..., e_n creates an *implied synchronization e* between processors p_0 and p_n. This synchronization has an iteration distance of $d_1+d_2....+d_n$ and $t_{src}(e)=t_{src}(e_1)$ and $t_{dest}(e)=t_{dest}(e_n)$.

Proof: This result is obvious from Figure 8. □

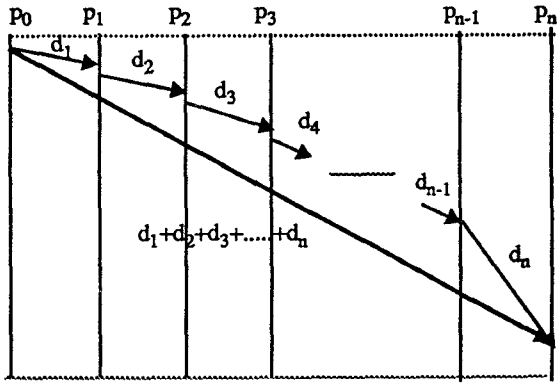

Figure 8: Implied Synchronizations.

Based upon the above results we develop an algorithm for distinguishing situations in which shared registers should be used from situations in which channels must be used for communicating values between processors. There are three major steps in this algorithm. In the first step we construct a graph representing the parallel schedule and interprocessor data dependencies. The dependencies also include loop carried dependencies if the code segment represents a loop body. Associated with each dependency is the iteration distance which is zero for non-loop carried dependencies and non-zero for loop-carried dependencies. Next we compute all implied synchronizations using theorem 2. In the final step we classify each real dependence edge as either requiring a shared register or a channel using theorem 1. The algorithm guarantees that the order in which a receiving processor reads data values from a channel queue is exactly the same as the order in which the data values are written to the channel queue by the sending processor. Thus, the implementation of channels as queues is an appropriate choice.

Step 1: Construction of a Graph Representing the Correct Execution Order

We construct a directed graph G=(V,E), from the parallel schedule and data dependency information, representing the constraints on the execution order of the statements as described below.

$p(s)$ - the processor on which the statement s has been scheduled for execution.

$t_{start}(s, p(s))$ - the expected time elapsed from the beginning of a loop iteration to the beginning of s's execution on processor $p(s)$.

$t_{end}(s, p(s))$ - the expected time elapsed from the beginning of a loop iteration to the end of the execution of statement s on processor $p(s)$.

V - set of statements in the computation; and

E - set of edges in the graph which are determined as follows.

An edge is introduced from statement s_i to statement s_j if:
(i) $p(s_i)=p(s_j)$ and s_j is executed immediately after s_j; or
(ii) $p(s_i)\neq p(s_j)$ and there is a data dependency from s_i to s_j.
An edge from statement s_i to statement s_j is denoted as $[t_{end}(s_i, p(s_i)), t_{start}(s_j, p(s_j)), d]$, where d is the iteration distance of the dependency known at compile-time.

Step 2: Computation of Implied Synchronizations

In this step we compute the set of implied synchronizations between pairs of processors in accordance with theorem 2. The computation requires a single bottom up traversal of the graph constructed in step 1. In the algorithm in Figure 9 the set I is the set of implied synchronization edges.

Step 3: Identify the Mode of Communication for Interprocessor Data Dependencies

The set of flow dependence edges E is partitioned into the set of edges E^{sr} which will make use of shared registers and the set of edges E^{ch} which will make use of channel queues using the algorithm in Figure 10.

3.3. Reducing Delays due to Bounded Channels

If an interprocessor data dependence with iteration distance of d is enforced using a channel queue, d values are accumulated in the channel queue since a value produced is consumed d iterations later. If the size of the channel queue is less than d, the writes to be performed by the sending processors will be delayed till the receiving processor reads the values from the channel queue. Code can be generated so that the delays that can be anticipated at compile-time are reduced.

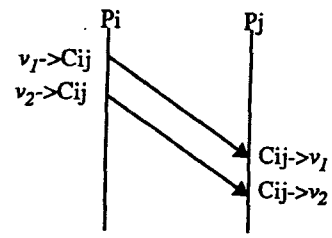

Figure 11: Delays due to Bounded Queues.

Consider the communication of values v_1 and v_2 from processor P_i to processor P_j as shown in Figure 11. If we assume that the channel C_{ij} can hold a single data value, then a delay can be expected during the execution of code assigned to processor P_i. This delay can be avoided either by modifying the code so that processor P_j reads the value v_1 from the channel C_{ij} early and saves it in a private register R or by delaying the write performed by processor P_i to channel C_{ij} by computing the value v_2 into a private register R. The delay can also be avoided by the combination of the two approaches. When the shift in the channel read/write operation is more than a single iteration, then the value will be held in a private register for

```
ComputeImpliedSynchronizations {
    /* MAXd - maximum iteration distance of a data dependency */
    /* I - set of implied synchronizations */
    I = φ
    mark all nodes in G as unvisited
    For each processor p Do
        find the earliest unvisited node n in processor p 's schedule and
        If one is found Then Traverse(n)
    Endfor
}
Traverse(n) {
    mark n as visited
    For each child c of n Do
        If c is unvisited Then Traverse(c) Endif
        If p(c) ≠ p(n) Then
            Generate implied synchronizations involving edge e = [t_end(n, p(n)), t_start(c, p(c)), d] as follows:
            For each edge e' = [t_end(c', p(c)), t_start(n', p(n')), d'] ε E ∪ I st p(n') ≠ p(n) and c' is after c Do
                If d + d' ≤ MAXd Then -- compute implied synchronization using theorem 2
                    I = I ∪ { [t_end(n, p(n)), t_start(n', p(n')), d + d'] }
                    -- implied synchronization with iteration distance > MAXd cannot
                       subsume a synchronization required for a true data dependency.
                Endif
            Endfor
        Endif
    Endfor
}
```

Figure 9. Computing Implied Synchronizations.

```
For each ordered processor pair (p_i, p_j) Do
    d = 0; E^sr = E^ch = φ
    While d ≤ MAXd Loop
        For each edge e = [t_end(s_i, p_i), t_start(s_j, p_j), d] ε E ∪ I Do
            Identify edges from p_i to p_j that are subsumed by e as follows:
            For each edge e' = [t_end(s'_i, p_i)), t_start(s'_j, p_j), d'] ε E such that d' ≥ d Do
                If -- conditions from theorem 1
                    ((d' = d) ∧ (t_end(s_i, p_i) > t_end(s'_i, p_i)) ∧ (t_start(s_j, p_j) < t_start(s'_j p_j)))
                    ∨ ((d' = d+1) ∧ not (t_end(s_i, p_i) < t_end(s'_i, p_i) ∧ t_start(s_j, p_j) > t_start(s'_j, p_j)))
                    ∨ (d' > d+1)
                Then E^sr = E^sr ∪ {e'}; E = E - {e'}
                Else E^ch = E^ch ∪ {e'}; E = E - {e'}
                Endif
            Endfor
        Endfor
        d = d + 1
    Endwhile
Endfor
```

Figure 10. Identifying the Mode of Communication.

more than one iteration.

4. Implementation and Experimental Results

The techniques described in this paper have been implemented and they were applied to some of the Livermore loops. The results of the experiments conducted demonstrate the effectiveness of top-down scheduling. The results indicate that in most cases the top-down scheduling approach results in almost half the number of interprocessor dependencies as compared to the schedules generated using list scheduling. The length of schedules generated by the two scheduling algorithms is almost the same in most cases. The schedules were also examined to determine the queue length that would guarantee no delays upon writes to channel queues. It was found that a queue length of less than four was sufficient for this purpose. This leads us to conclude that channel queues with small lengths form an effective mechanism for achieving interprocessor communication in a fine-grained MIMD system provided that appropriate compilation techniques are used.

5. Related Work

An alternative approach for implementing channels is to provide globally shared channels each with a full/empty synchronization bit. This approach has been studied in earlier work.[7,9] The channels must be addressable as registers to achieve high execution speeds which limits the number of globally shared channels that can be provided. On the other hand the number of bits needed to address dedicated channel queues is limited by the number of processors. An increase in the channel queue length does not increase the number of bits required to address the channel queues. The channel queues are also easier to implement in hardware. The compilation techniques for the allocation of global channels are also quite complicated.[7] In order to enforce a loop carried dependency multiple global channels are required. Thus, the loop must be sufficiently unrolled so that different global channels can be used during different iterations of the loop. On the other hand this can be achieved without unrolling if channel queues are used.

6. Conclusion

This paper demonstrated the use of channel queues to exploit fine-grained parallelism present in sequential programs. Compilation techniques for the exploitation of such a resource were presented. The experimental results demonstrate that a small queue length (four) is sufficient to exploit parallelism in several applications.

The compilation techniques developed in this paper are also applicable to other parallel architectures. The top-down scheduling algorithm can be used to schedule tasks on shared-memory machines as well as distributed memory machines since the reduction of interprocessor communication and processor synchronization is essential for obtaining good performance. Elimination of redundant synchronizations on a shared-memory machine can also reduce synchronization overhead.

References

1. A. Aiken and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Trans. on Software Eng.*, vol. 14, no. 5, pp. 584-594, 1988.
2. A. Aiken and A. Nicolau, "Optimal Loop Parallelization," *Proc. of the SIGPLAN Conf. on Prog. Lang. Design and Implementation, Atlanta, Georgia*, pp. 308-317, 1988.
3. 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," *IEEE Transactions on Computers*, vol. 37, pp. 967-979, August, 1988.
4. 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.
5. R. Gupta and M.L. Soffa, "Compilation Techniques for a Reconfigurable LIW Architecture," *The Journal of Supercomputing*, vol. 3, pp. 271-304, 1989.
6. 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.
7. R. Gupta, "Employing Register Channels for the Exploitation of Instruction Level Parallelism," *Proc. 2nd ACM Sigplan Symp. on Principles and Practice of Parallel Programming*, pp. 118-127, 1990.
8. R. Gupta and M.L. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 421-431, April 1990.
9. R. Gupta, "A Fine-Grained MIMD Architecture Based Upon Register Channels," *Proceedings of the 23rd Annual Workshop on Microprogramming and Microarchitecture*, pp. 28-37, Nov., 1990.
10. 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.
11. D.J Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proc. of the 8th Annual ACM Symp. on Principles of Programming Languages*, pp. 207-218, 1981.
12. M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. of the SIGPLAN'88 Conf. on Prog. Lang. Design and Implementation*, pp. 318-328, 1988.