International Journal of Parallel Programming, Vol. 21, No. 3, 1992

Exploiting Parallelism on a Fine-Grained MIMD Architecture Based Upon Channel Queues[1]

Rajiv Gupta[2] and Sunah Lee[2]

Received August 1991; Revised January 1993

We present techniques for exploiting fine-grained parallelism extracted from sequential programs on a fine-grained MIMD system. The system exploits fine-grained parallelism through parallel execution of instructions on multiple processors as well as pipelined nature of individual processors. The processors can communicate data values via globally shared registers as well as dedicated channel queues. Compilation techniques are presented to utilize these mechanisms. A scheduling algorithm has been developed to distribute operations among the processors in a manner that reduces communication among the processors. The compiler identifies data dependencies which require synchronization and enforces them using channel queues. Delays that may result by attempting write operations to a full channel queue are avoided by spilling values from channels to local registers. If an interprocessor data dependency does not require synchronization, then the data value is passed through a shared register or shared memory.

KEY WORDS: Multiprocessor systems; parallelizing compilers; fine-grained parallelism; top-down scheduling; redundant synchronization; channel queues.

1. INTRODUCTION

Implicit parallelism present in sequential programs is an important source of fine-grained parallelism. This parallelism can be divided into two broad

[1] Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 (CCR-9249143) to the University of Pittsburgh.

[2] Department of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania 15260.

828/21/3-1

0885-7458/92/0600-0169$06.50/0 © 1992 Plenum Publishing Corporation

categories, namely loop level parallelism and extra-loop (or nonloop) 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) architectures are a family of architectures that can effectively exploit fine-grained parallelism present in sequential parts of a program.[1,2] 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. 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. The long instruction word allows initiation of several fine-grained operations in each instruction. The compilers for VLIW machines can detect and schedule extra-loop parallelism in sequential parts of the program and also exploit loop level parallelism through loop unrolling, software pipelining and other loop transformations.[3–7]

There are two main limitations of a VLIW machine. First it cannot be used as a multiprocessor since it executes a single stream of instructions. The second disadvantage arises due to events that are not predictable 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 presence of arrays and pointers. The lockstep operation of multiple processors makes the machine intolerant to runtime 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.

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 contains a small number of processors, possibly on the same chip, which allows exploitation of instruction level parallelism. 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. Unlike a VLIW machine, the MIMD system is tolerant of

delays caused by unpredictable events since the processors are not required to operate in lockstep.

The compilation techniques developed for VLIW machines, such as trace scheduling,[5] region scheduling,[6] software pipelining,[7] and optimal loop parallelization[4] can also be used to generate code for the fine-grained architecture. However, the above techniques must be adapted to take advantage of the data communication mechanisms supported by this 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 runtime delays as well as reduce the effectiveness of a processor's pipeline. A scheduling algorithm, namely top-down scheduling, that achieves the above goal is proposed in this paper. 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.

In the next section, a brief description of the fine-grained MIMD architecture is presented. In subsequent sections the compilation techniques for the architecture are discussed in detail. A scheduling algorithm which attempts to reduce interprocessor communication is presented. 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. We also present techniques by which anticipated delays during read/write operations on channels are avoided. Results of some experiments that demonstrate the effectiveness of the scheduling algorithm and the feasibility of a channel based architecture are presented.

2. THE ARCHITECTURE

In this section we discuss the primary features of the fine-grained MIMD architecture. The system is composed of four pipelined RISC processors augmented with multiprocessor support. The processors have a load/store architecture which is preserved by the multiprocessor features provided in the architecture. An operand involved in the execution of an instruction is read from, or written to, the executing processor's private register, a register globally shared among all of the processors, or a channel queue between the executing processor and any other processor in the system. The operand specification in an instruction includes 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. We envision that the four processor system will be

implemented on a single chip which enables the implementation of high speed synchronization and data communication mechanisms.

From each processor to every other processor a channel queue is provided (see Fig. 1). 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 attempting to read an empty queue or write to a full channel. The channels are organized as queues because, as we will demonstrate later in the paper, through appropriate compilation techniques we can 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 the value 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

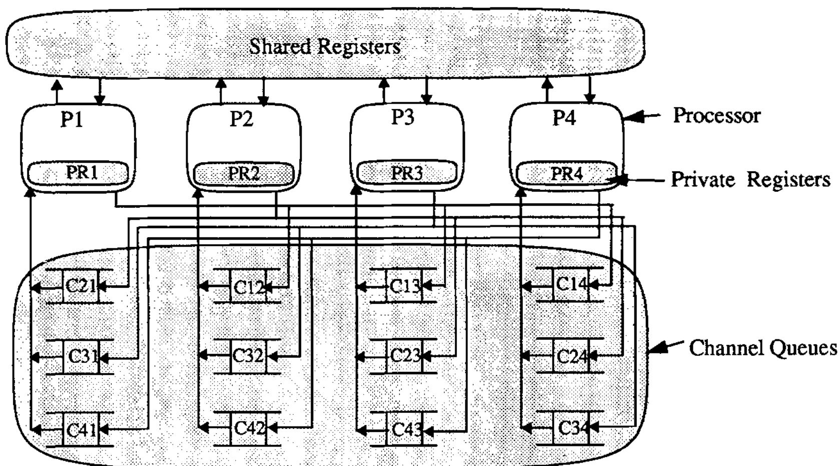


Fig. 1. The fine-grained MIMD architecture.

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 channels it is useful to provide barrier synchronization and an efficient collective branching mechanism in such a system. The design and use of these mechanisms was discussed in earlier work.[8]

3. COMPILER SUPPORT FOR UTILIZING CHANNEL QUEUES

Both extra-loop parallelism and loop level parallelism can be detected and exploited by the compiler. By constructing a directed acyclic graph (DAG) representing the data dependencies among the statements in a single basic block, or a sequence of basic blocks, extra-loop parallelism can be detected. Fine-grained parallelism present across loop iterations can be exposed by loop unrolling. Although loop unrolling is effective in exposing loop level parallelism, it causes a significant code growth. A technique developed by Aiken and Nicolau[4] transforms parallelism present across loop iterations to extra-loop parallelism with little code growth. Thus, after applying Aiken and Nicolau's transformation fine-grained parallelism can be detected by constructing a DAG for the loop body. The construction of a DAG requires data dependency information. There are three types of data dependencies flow, anti, and output. The flow dependencies represent true data dependencies and the other two types can be eliminated through renaming techniques. Furthermore we assume that control dependencies due to if-statements inside the loop body have been converted to data dependencies.[9] After the detection of fine-grained parallelism the compiler must perform the following steps to generate code for the fine-grained MIMD architecture.

- A parallel execution schedule is generated. By scheduling parallelism such that there are fewer interprocessor data dependencies, the top-down scheduling algorithm improves the performance of processor pipeline.
- The interprocessor data dependencies, including dependencies across loop iterations, are resolved through shared registers, channel queues, and shared memory.
- 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. By spilling a data value from a channel queue into a private register, the time that a data value spends in a channel is reduced and the delay associated with a write to a full channel is avoided.

3.1. Top-Down Instruction Scheduling

The need for the use of channels is dependent upon the parallel schedule. Thus, before the assignment of channels can be carried out a parallel schedule must be generated. The scheduling algorithm examines the DAG representing the data dependencies to generate an instruction schedule. A parent node in a DAG is data dependent upon its child nodes and a directed edge from a child node to a parent node indicates the direction in which data flows. A simple 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 along the longer unscheduled paths are scheduled first. The list scheduling algorithm does not make any attempt to reduce interprocessor communication. Next we discuss modifications to list scheduling to overcome this drawback.

- The list scheduler may assign different processors to a parent node and each of its children. In this case interprocessor communication is required to enforce data dependencies 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. The result is a reduction in the number of instances in which interprocessor communication takes place by one.
- Consider a DAG containing more parallelism than the processors in the system can exploit. In this situation the scheduling algorithm must selectively exploit parallelism in a manner that reduces interprocessor communication. The list scheduler is modified so that it identifies subDAGs that can be executed in parallel. These subDAGs are assigned to different processors. By choosing not to exploit the parallelism within a subDAG interprocessor communication is avoided. At the same time by exploiting the parallelism across the subDAGs all processors are kept busy. Since most operations require more than one operand it is often more likely that parallel subDAGs, each of which has a single root node, can be found at the top of the DAG. Thus, the scheduling is carried out in top-down fashion instead of the bottom-up fashion. As a result, the last instruction in the schedule is selected first and the first instruction to be executed is selected last.
- An advantage of list scheduling is that it tries to distribute the operations equally among the processors, which results in fast

schedules. It is therefore essential to maintain this characteristic during top-down scheduling. This goal is achieved by ensuring that the subDAGs that are simultaneously scheduled on the processors contain the same number of operations.

The identification of subDAGs for parallel execution is carried out as follows. Initially each subDAG contains simply their respective root nodes. The subDAGs are gradually expanded by including one node at a time to each of the subDAGs. If corresponding to each subDAG, a distinct node that is ready to be scheduled can be identified, then these nodes are added to the respective subDAGs. This step is carried out repeatedly as long as all subDAGs continue to grow. On the other hand if the search fails for any one of the subDAGs, none of the nodes identified in the current step are added to the subDAGs. This process ensures that the sizes of the subDAGs identified are equal. The nodes are examined in a breadth-first top-down fashion during this process.

The modifications to list scheduling give us a new scheduling algorithm that we refer to as the top-down scheduling algorithm. Top-down

TopDownScheduling {
  -- parent and child refer to nodes connected by a non-loop carried dependency

  Compute $\forall\, n_i$

$$height(n_i) = \begin{cases} 1 & \text{if } n_i \text{ has no child} \\ 1 + \max_{n_j,\ \text{child of } n_i} (height(n_j)) & \text{otherwise} \end{cases}$$

  Loop {
    Construct $S = n_1, n_2, \ldots, n_m$
      st $\forall\, n_i \in S$ the parents of n_i have been scheduled $\wedge$ height(n_i) $\geq$ height(n_{i+1})

    Let p be the number of processors available
    For i=1 to minimum(p, $|S|$) **Do**
      If processor p_i is available and a parent of node n_i is scheduled on p_i
      Then schedule n_i on processor p_i
      Else schedule n_i on any available processor
      Endif
    Endfor

    If $|S| \geq p$ **Then**
      -- nodes $n_1, n_2, \ldots, n_p$ have already been scheduled on the p processors
      On each processor p_i, schedule a set of operations S_i such that
        S_i is a subset of operations belonging to the subtree rooted at n_i and
          $|S_1| = |S_2| = \ldots = |S_p|$
    Endif
  } **Until** all operations have been scheduled
}

Fig. 2. Top-down scheduling algorithm.

scheduling reduces interprocessor communication due to data dependencies with iteration distance zero. However, there is additional interprocessor communication that may result from loop carried dependencies. The top-down scheduling algorithm is described in Fig. 2.

Runtime Complexity: Let E and V represent the sets of edges and nodes in the DAG. 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 overall runtime complexity of the top-down scheduling algorithm is $O(|V|^2)$.

An example which illustrates the effect of these modifications upon interprocessor communication is shown in Fig. 3. Both list scheduling resulted in equally fast schedules. However, the top-down schedule requires significantly less interprocessor communication. In Fig. 3, the shaded regions represent expected execution time delays for the depicted schedules. Since the code being generated is for a MIMD machine, no delay instructions are actually introduced in the code.

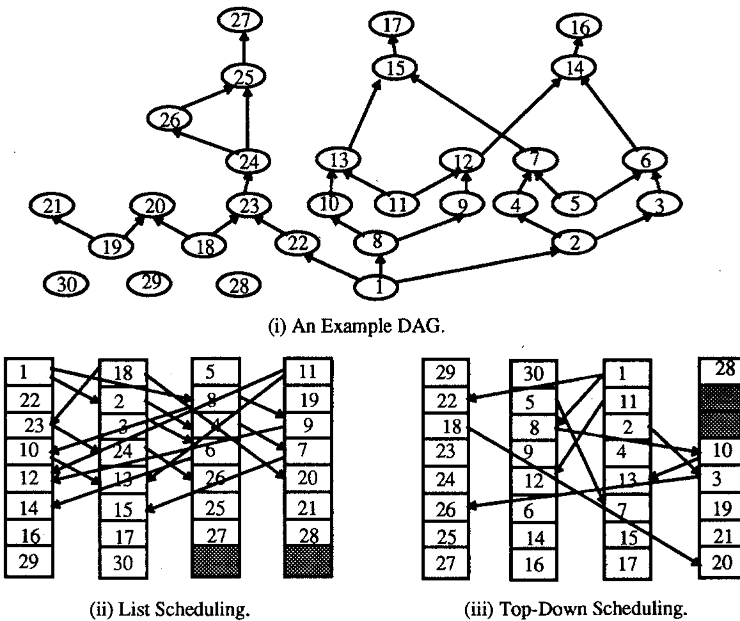


Fig. 3. A scheduling example.

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 channel queues, shared registers, or shared memory. In this section we present compile-time techniques that enable us to make an appropriate choice for each dependency. Depending upon the nature of dependency between pair of statements the following actions are performed at runtime.

Synchronization Only: The enforcement of output dependencies, anti dependencies, and indefinite dependencies (a dependence which is assumed to exist since the compiler cannot guarantee its absence) only requires synchronization of processors. In such a situation the channel is used to synchronize the processors. However, the value written to and read from the channel is of no interest.

Communication Only: The enforcement of a flow dependence requires the communication of a value from one processor to another. If the processor receiving the value is guaranteed to read the value after the value has been computed, then a shared register or shared memory is used to communicate the value. Note that a value communicated through a shared register or shared memory cannot be overwritten before it is read. This is because there will be an anti dependence between the statement that reads the value and the statement which performs the subseuent write. As discussed earlier, this dependence will be enforced through a synchronization edge using a channel to guarantee correct execution. Since there are limited number of shared registers available, a global register allocation is required to assign some communication only values to shared registers while others to shared memory. A global register allocation applicable to scalars and array references has been developed by Duesterwald *et al.*[10]

Communication and Synchronization: The enforcement of flow dependence which not only requires the communication of a value from one processor to another, but also requires explicit synchronization to force the receiving processor to perform the read after the write operation has been performed, is achieved using the channel queue from the sending processor to the receiving processor. A flow dependence across iterations can only be enforced through channel queues if the iteration distance is a compile-time constant. However, if this is not the case we must assume the minimum possible value for the iteration distance and treat the dependence as a *synchronization only* dependence.

The type of each dependence is known to the compiler. However, additional analysis is required to determine whether or not the enforcement

of a flow dependence requires synchronization. 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, i.e., makes the latter unnecessary. In the subsequent discussion, given an interprocessor data dependence edge e, $t_{src}(e)$ denotes the static estimate of time elapsed since the beginning of the loop's execution to the end of the execution of the source instruction corresponding to the dependence edge e. The time $t_{dest}(e)$ denotes the static estimate of time elapsed till the beginning of the destination instruction corresponding to e.

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 as shown in Fig. 4, 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: $t_{src}(e_1) < t_{src}(e_2) \wedge t_{dest}(e_1) \leqslant t_{dest}(e_2)$

If the two edges do not intersect then synchronization is clearly required to enforce the two dependencies (see Fig. 5). Consequently the result stated previously follows. □

In the subsequent results we will consider across iteration dependencies. In order to derive results regarding across iteration synchronizations we

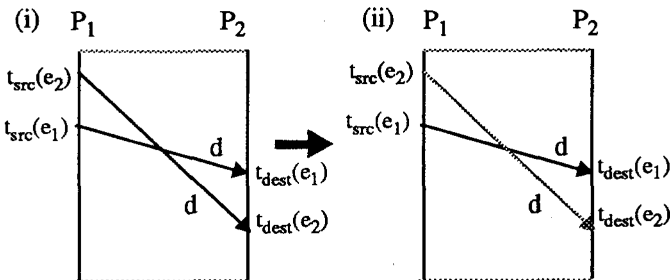


Fig. 4. Intersecting dependence edges with same iteration distance (d).

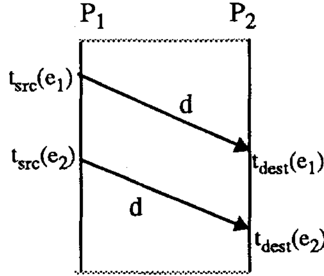


Fig. 5. Nonintersecting dependence edges with same iteration distance (d).

must examine successive loop iterations. The number of loop iterations that must be examined is one more than the difference between the iteration distances of the pair of dependencies being considered.

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 this result we consider the following cases which cover all possible relationships between dependence edges e_1 and e_2.

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

This situation is shown in Fig. 6(i). Instead of looking at a single loop iteration let us examine two successive loop iterations as shown in Fig. 6(ii). The dependency e_2 has an iteration distance of $d+1$. Thus, if we examine two successive loop iterations we can represent e_2 as a dependency between the loop iterations with iteration distance d. Now from Lemma 1 it follows that e_2 is subsumed by e_1.

Case II: $t_{src}(e_1) > t_{src}(e_2) \wedge t_{dest}(e_1) \geqslant t_{dest}(e_2)$

This situation is similar to the previous case. If we examine two loop iterations and transform e_2 into a dependency of distance d it is clear that e_2 is subsumed by e_1.

Case III: $t_{src}(e_1) > t_{src}(e_2) \wedge t_{dest}(e_1) \leqslant t_{dest}(e_2)$

In this case as in the two previous cases by applying Lemma 1 it is clear that e_2 is subsumed by e_1.

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

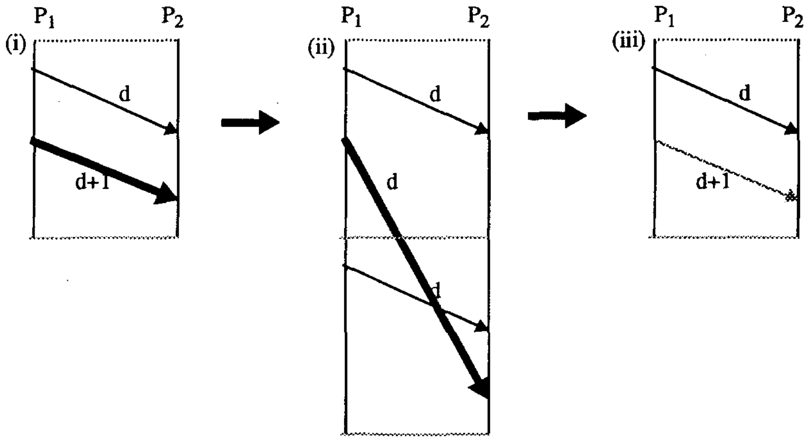


Fig. 6. Nonintersecting dependence edges of iteration distance $d+1$ and d.

Finally we consider the last possibility. As we can see from Fig. 7 the dependency represented by e_2 is not subsumed by e_1. Thus, the condition that embodies this case is the condition under which e_2 is not subsumed by e_1. Therefore, the result stated in Lemma 2 follows. □

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 $d+1$.

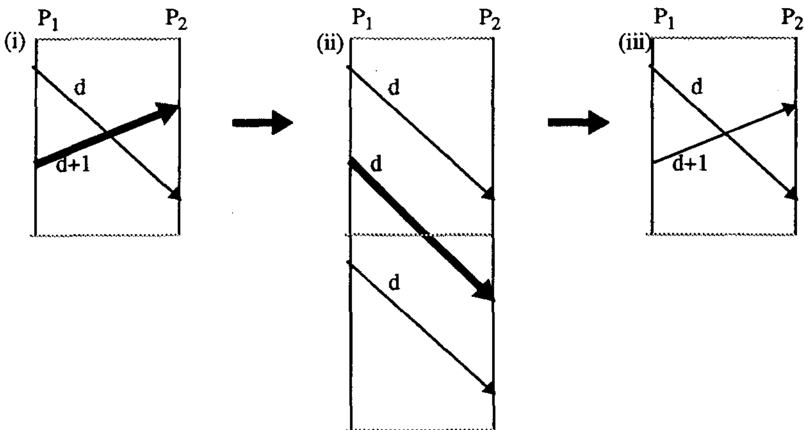


Fig. 7. Intersecting dependence edges of iteration distance $d+1$ and d.

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. In the first three cases the dependency e_2 is subsumed by the dependency e_1. This can be easily shown by first viewing e_2 as a dependency of iteration distance $d+1$ by considering an additional loop iteration and then applying the analysis of the first three cases in Lemma 2. The fourth situation is shown in Fig. 8(i). In this case we view e_2 as a dependency with iteration distance d by considering two additional loop iterations as shown in Fig. 8(ii). Next by applying Lemma 1 we conclude that e_2 is subsumed by e_1. Thus, we have shown that e_2 is subsumed by e_1 if the iteration distance of e_2 is $d+2$. It is obvious that this result will hold even if e_2 has an iteration distance which is greater than $d+2$. □

Theorem 1. Subsumption Theorem: 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:

(i) $d' = d \wedge t_{src}(e_i) > t_{src}(e_j) \wedge t_{dest}(e_i) < t_{dest}(e_j)$.

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

(iii) $d' > d+1$.

Proof: This theorem follows directly from Lemmas 1–3. □

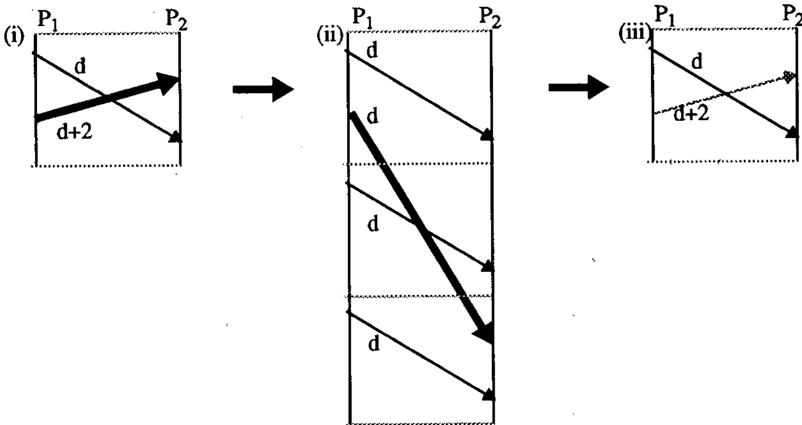


Fig. 8. Dependence edges of iteration distance d and $d+2$.

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 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 synchronizations.

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 $t_{dest}(e_i) \leqslant t_{src}(e_{i+1})$. 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 $d_1 + d_2 \cdots + d_n$ and $t_{src}(e) = t_{src}(e_1)$ and $t_{dest}(e) = t_{dest}(e_n)$.

Proof: The result can be inferred from Fig. 9 as follows.

p_1 cannot proceed beyond $t_{dest}(e_1)$ until d_1 iterations earlier p_0 has gone beyond $t_{src}(e_1)$

$\wedge$ p_2 cannot proceed beyond $t_{dest}(e_2)$ until d_2 iterations earlier p_1 has gone beyond $t_{src}(e_2)$ and $t_{dest}(e_1) < t_{src}(e_2)$

...

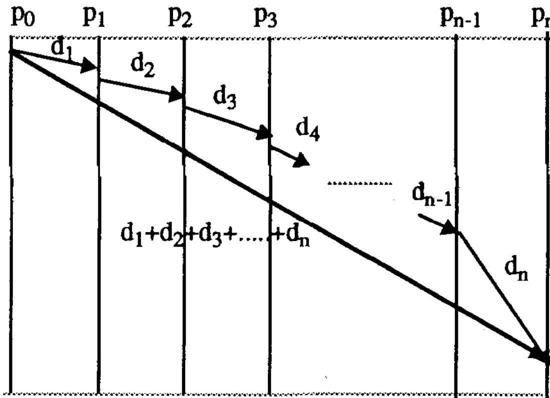


Fig. 9. Implied synchronizations.

$\wedge$ p_n cannot proceed beyond $t_{dest}(e_n)$ until d_n iterations earlier p_{n-1} has gone beyond $t_{src}(e_n)$ and $t_{dest}(e_{n-1}) < t_{src}(e_n)$

$\Rightarrow$

p_n cannot proceed beyond $t_{dest}(e_n)$ until $d_1 + d_2 \cdots + d_n$ iterations earlier p_0 has gone beyond $t_{src}(e_1)$

In other words there is an implied synchronization edge e from p_0 to p_n with iteration distance of $d_1 + d_2 \cdots + d_n$ and $t_{src}(e) = t_{src}(e_1)$ and $t_{dest}(e) = t_{dest}(e_n)$. □

Based upon these 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 nonloop carried dependencies and nonzero 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 in Fig. 10.

Step 2. Computation of Implied Synchronizations: In this step we compute the set of implied synchronizations between pairs of processors, see Fig. 11. The computation requires a single bottom up traversal of the graph constructed in step 1. In the algorithm I denotes 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^{srm} that will make use of shared registers or shared memory and, the set of edges E^{ch} will make use of channel queues. For each edge in E^{srm} it is guaranteed that the value will not be read till it has been computed. An edge in E^{srm} may represent the transmission of a value

$p(s)$ - 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 $p(s)$.

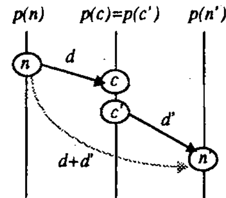
V = set of statements in the computation; and
E = set of edges in the graph which are determined as follows.
An edge is introduced from statement s_i to statement s_j if:
(i) $p(s_i)=p(s_j)$ and s_j is executed immediately after s_j; or
(ii) $p(s_i)\neq p(s_j)$ and there is a data dependency from s_i 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.

Fig. 10. Construction of execution order graph.

of a scalar variable or an array element. An existing global register allocation algorithm[10] that can handle scalars and array references can be employed for selecting the values that will be transmitted through shared registers. The remainder of the values are transmitted through shared memory. (See Fig. 12.)

MAXd - maximum iteration distance of a data dependency




```
Compute_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
}
```

Fig. 11. An algorithm for computing implied synchronization.

```
For each ordered processor pair (p_i, p_j) Do
    d = 0;
    E^srm = 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^srm = E^srm ∪ {e'}; E = E - {e'}
                Else E^ch = E^ch ∪ {e'}; E = E - {e'}
                Endif
            Endfor
        Endfor
        d = d + 1
    Endwhile
Endfor
```

Fig. 12. An algorithm for identifying the mode of interprocessor communication.

3.3. Handling Conditional Statements

So far we assumed that the loop being scheduled contains no conditional statements. We now briefly demonstrate the feasibility of using channel queues in the presence of conditional statements. Consider the situation in which a value computed by one processor may be needed by another processor in a subsequent iteration. The value can be passed through a channel queue as before. However, if the value is not required by the receiving processor, the processor still must read the value from the channel queue and then discard it. This is essential because a value written to a channel queue cannot be overwritten and they must be read if they are to be removed from the channel. Thus, conditional read operations on channel queues must be transformed in unconditional read operations. Similarly it can be shown that conditional write operations to channels must also be transformed to unconditional write operations. Consider the loop shown in Fig. 13. There is a conditional interprocessor data dependence between statements $S2$ and $S3$. As shown in Fig. 13 this dependence can be handled using the channel queue C_{12} if the write and read operations to C_{12} are performed unconditionally.

```
Do I = 1, N
  S1: -
  S2: If (..) Then A[I] = - Endif
  S3: If (..) Then - = A[I] Endif
  S4: -
EndDo
```

P_1	P_2
S_2	S_1
S_4	S_3

```
Do I = 1, N
  S1: -
  S2: If (..) Then A[I] = - Endif; C12 = A[I]
  S3: t = C12; If (..) Then - = t Endif
  S4: -
EndDo
```

Fig. 13. An example with conditional statements.

3.4. Reducing Delays Due to Bounded Channel Queues

The size of each channel queue is bounded. As a result delays may be caused when write operations are attempted on a full channel queue. Next, we present techniques for generating code so that delays that can be anticipated at compile-time are reduced. Consider the communication of values v_1 and v_2 from processor P_i to processor P_j as shown in Fig. 14(i). 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 (see Fig. 14(ii)) 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 (see Fig. 14(iii)). The delay can also be avoided by the combination of the two approaches.

Next we present an algorithm which modifies the code so that the delays can be avoided. For each interprocessor dependence edge we determine the amount of time by which the write should be delayed or by how much earlier the read should be performed. In order to ensure that the effects due to all interprocessor dependencies have been considered we must examine **d** successive iterations of the loop, where **d** is the maximum iteration distance of any interprocessor dependency. The algorithm in Fig. 15 repeatedly detects sequence of edges, from *first* to *last*, that are expected to experience delays. Then it removes the delays either by delaying the writes using function **DelayWrites** or by performing early reads using function **EarlyReads**. The availability of private registers on communicating processors is used to determine the approach to be applied to a sequence of edges. The delay in a channel write, or the amount by which the read is moved earlier, can be greater than a length of an iteration since **d** iterations are examined by this algorithm. 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 more than one *iteration.*[10,11] Furthermore, we also

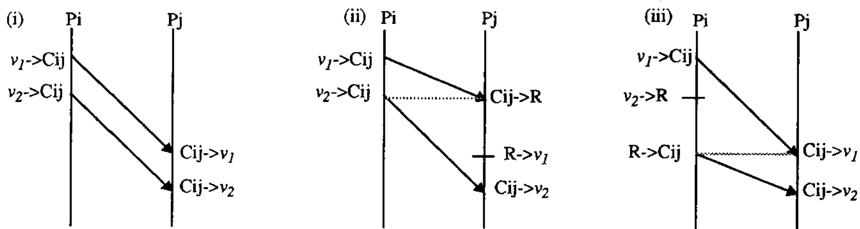


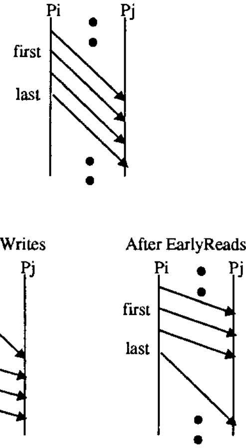
Fig. 14. Avoiding channel read/write delays.

Let E(p,q)={$e_1, e_2, \ldots, e_{max}$} denote the set of flow dependence edges from processor p to q that use channel queue C_{pq} and the edges are ordered according the order in which the writes to C_{pq} are performed.

$t_{src}(e_i)$ - expected time elapsed from the start of the first loop iteration to the time at which the write to channel is performed.

$t_{dest}(e_i)$ - expected time elapsed from the start of the first loop iteration to the time at which the read on the channel is performed.

current = 2;
first = last = 0;
While current ≤ max **Loop**
  If $t_{dest}(e_{current-1}) > t_{src}(e_{current})$ **Then**
    If first = 0 **Then** first = last = current **Else** last = current **Endif**
  Endif
  If (first ≠ 0) and (last ≠ current or current = max) **Then**
    If local registers available at p > local registers available at q
    Then DelayWrites(first,last) **Else** EarlyReads(last,first) **Endif**
    first = last = 0
  Endif
  current = current + 1
Endwhile



DelayWrites(first, last) {
  For i = first **To** ≤ last **Do**
    Shift_write(e_i) = $t_{dest}(e_{i-1})$ - $t_{src}(e_i)$
  Endfor
}

EarlyReads(last, first) {
  For i = last **Down To** ≤ first **Do**
    Shift_read(e_{i-1}) = $t_{dest}(e_{i-1})$ - $t_{src}(e_i)$
  Endfor
}

Fig. 15. An algorithm for reducing channel read/write delays.

know that an additional value will be spilled from the channel queue to a private register during each loop iteration. The number of private registers to hold the spilled values is equal to the dependence distance. If enough private registers are not available, the values must be spilled in memory. In Fig. 15, we assume that each channel queue can contain at most one data value. Thus, each edge is examined for potential delay. However, if the channel queue length is greater than one, then we need to examine every lth edge, where l is the length of a channel queue.

4. EXPERIMENTAL RESULTS

The techniques described in this paper were applied to code segments taken from scientific programs. We applied this approach to a sample of data dependency graphs to study the effectiveness of top-down scheduling and to determine the feasibility of providing channel queues. Some of the DAGs used in this study had been constructed by Rodeheffer.[12] Straight-

line code segments were constructed by converting control dependencies into data dependencies.[9,13] The test programs include the following:

EIGEN – The Generalized Eigenvalue Problem.
LINEAR – Solving Linear Equations Using Residue Arithmetic.
CURVEFIT – Discrete Chebychev Curve Fit.
TAYLOR – Evaluation of Normalized Taylor Coefficients.
FOURIER – Calculation of Fourier Integrals.
LIVERMORE LOOPS – Loops 2, 7, 9, 10, 21, and 23.

The results of the experiments conducted demonstrate the effectiveness of top-down scheduling. The results in Table I provide a comparison between list scheduling and top-down scheduling. In many cases the top-down scheduling approach results in almost half the number of inter-processor dependencies (#DEPS) as compared to the schedules generated using list scheduling. The length of schedules (LENGTH) generated by the two scheduling algorithms is almost the same in most cases. Thus, the results strongly indicate that top-down scheduling is as effective as list scheduling in exploiting parallelism and in addition it significantly reduces interprocessor communication. Assuming that each operation takes unit time, the fastest possible schedule for a DAG on a p processor system is equal to $\max(\lceil N_B/p \rceil, N_L)$, where N_B is the number of nodes in the DAG and N_L is the longest chain of dependencies in the DAG. The lengths of schedules are quite close to the fastest possible schedule (IDEAL) for most of the programs indicating that both list scheduling and top-down

Table I. Parallel Schedules on a Four Processor System

		LIST_SCHED		TOP_DOWN_SCHED	
	IDEAL $= \max(\lceil N_B/p \rceil, N_L)$	LENGTH	#DEPS	LENGTH	#DEPS
EIGEN	$\max(\lceil \frac{11}{4} \rceil, 7) = 7$	7	4	7	2
LINEAR	$\max(\lceil \frac{17}{4} \rceil, 6) = 6$	6	9	6	4
CURVEFIT	$\max(\lceil \frac{20}{4} \rceil, 7) = 7$	12	6	7	4
TAYLOR	$\max(\lceil \frac{41}{4} \rceil, 11) = 11$	11	19	11	7
FOURIER	$\max(\lceil \frac{30}{4} \rceil, 7) = 8$	8	17	8	5
Loop 2	$\max(\lceil \frac{14}{4} \rceil, 5) = 5$	5	8	5	5
Loop 7	$\max(\lceil \frac{33}{4} \rceil, 9) = 9$	10	13	11	7
Loop 9	$\max(\lceil \frac{17}{4} \rceil, 6) = 6$	6	9	6	7
Loop 10	$\max(\lceil \frac{22}{4} \rceil, 7) = 7$	11	11	11	8
Loop 21	$\max(\lceil \frac{12}{4} \rceil, 6) = 6$	6	5	6	3
Loop 23	$\max(\lceil \frac{38}{4} \rceil, 9) = 10$	11	22	11	12

scheduling algorithms exploit parallelism quite effectively. 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 illustrates that the top-down scheduling algorithm was effective in avoiding loop carried as well as nonloop carried interprocessor dependencies. 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

Following the success of VLIW systems there has been a significant interest in the development of fine-grained MIMD architectures. The architecture briefly described in this paper is one such architecture. Another architecture which falls in the same category is the OSCAR multiprocessor.(14) Although OSCAR is a tightly coupled MIMD system, unlike the architecture described in this paper, it does not provide fast data passing mechanisms such as shared registers and channel queues. The data transfer overhead although low is significant. Thus, OSCAR is can only effectively exploit *near* fine grained parallelism.

The architecture described in this paper provides a dedicated channel queue between every processor pair. 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.(15) 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 nontrivial.(15) On the other hand, no compiler algorithms are required for the allocation of channel queues since they are dedicated to a pair of procesors. The *HEP*(16) multiprocessor provides memory channels by adding a synchronization bit to every memory location in shared memory. Memory channels do not allow high speed communication among parallel streams. Thus, memory channels are inappropriate for the fine-grained MIMD architecture presented in this paper.

The top-down scheduling algorithm described in this paper attempts to generate schedules with low interprocessor communication to maximize

the likelihood of achieving all interprocessor data communication through shared registers and channel queues. Scheduling algorithm developed for OSCAR[14] also attempts to reduce interprocessor communication. The algorithm constructs tasks containing several instructions and schedules such tasks among the processors. The synchronization is performed at task level to limit synchronization overhead. This approach is not suitable for the fine-grained MIMD architecture described in this paper since it would not fully exploit the channels and shared registers for enforcing maximum number of data dependencies among the processors. Several coarse-grain scheduling algorithms which take communication costs into account during scheduling have also been developed.[17,18] However, these techniques are also inappropriate for the fine-grained MIMD system considered in this paper.

The problem of recognizing dependencies which can be enforced without explicit synchronization has been addressed in previous work by Li and Abu-Sufah,[19] Midkiff and Padua,[20] and Krothapalli and Sadayappan.[21] Li and Abu-Sufah,[19] identify certain situations where a dependence is rendered redundant while Midkiff and Padua[20] determine all redundant dependencies by performing transitive closures on subgraphs representing dependencies in a singly nested loop. The size of dependence graph increases linearly with the maximum dependence distance in the loop and the number of transitive closures performed equals the number of dependencies in the loop. Krothapalli and Sadayappan[21] develop an efficient algorithm by demonstrating that redundant dependencies can be identified without requiring a complete transitive closure of the data dependence graph. The algorithm presented in this paper also efficiently identifies all redundant dependencies. Like the algorithm by Krothapalli and Sadayappan[21] all implied synchronizations are computed in a single pass thus avoiding repeated transitive closures that are required by Midkiff and Padua's[20] approach. In addition, the algorithm in this paper differs from all of these algorithms in one important aspect. It takes advantage of the specific schedule generated by the top-down scheduling algorithm to further reduce the number of synchronizations that must be enforced. The dependencies which are implicitly enforced by the instruction schedule do not require the use of shared registers or channels.

6. CONCLUSION

This paper demonstrated the use of channel queues to exploit fine-grained parallelism in sequential programs. Compilation techniques for the exploitation of such a resource were presented. The use of channel

queues should provide an improvement in performance over VLIW machines as the multiple processors are no longer constrained to execute in lockstep. 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. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Massachusetts (1986).
2. R. Gupta and M. L. Soffa, Compilation Techniques for a Reconfigurable LIW Architecture, *The J. of Supercomputing* **3**:271–304 (1989).
3. A. Aiken and A. Nicolau, A Development Environment for Horizontal Microcode, *IEEE Trans. on Software Eng.* **14**(5):584–594 (1988).
4. A. Aiken and A. Nicolau, Optimal Loop Parallelization, *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pp. 308–317 (1988).
5. J. A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. on Computers* **C-30**(7):478–490 (July 1981).
6. R. Gupta and M. L. Soffa, Region Scheduling: An Approach for Detecting and Redistributing Parallelism, *IEEE Trans. on Software Eng.* **16**(4):421–431 (April 1990).
7. M. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Proc. of the SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pp. 318–328 (1988).
8. R. Gupta, The Fuzzy Barrier: A Mechanism for High-Speed Synchronization of Processors, *Proc. of the Third Int'l. Conf. on Arch. Support for Prog. Lang. and Oper. Syst.*, pp. 54–64 (April 1989).
9. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, Conversion of Control Dependence to Data Dependence, *Proc. of the Tenth Ann. Symp. on Principles of Prog. Lang.*, pp. 177–189 (1983).
10. E. Duesterwald, R. Gupta, and M. L. Soffa, Register Pipelining: An Integrated Approach to Register Allocation for Scalar and Subscripted Variables, *Int'l. Workshop on Compiler Construction*, Springer Verlag, *LNCS* **641**:192–206 (October 1992).
11. D. Callahan and B. Koblenz, Register Allocation via Hierarchical Graph Coloring, *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pp. 192–203 (1991).
12. T. L. Rodeheffer, Compiling Ordinary Programs for Execution on an Asynchronous Multiprocessor, Department of Computer Science; Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh (1985).
13. D. A. Padua, D. J. Kuck, and D. Lawrie, High-Speed Multiprocessors and Compilation Techniques, *IEEE Trans. on Computers* **29**(9):763–776 (1980).

14. H. Kasahara, H. Honda, and S. Narita, Parallel Processing of Near Fine Grain Tasks using Static Scheduling on OSCAR, *Proc. of Supercomputing*, pp. 856–864 (November 1990).
15. R. Gupta, Employing Register Channels for the Exploitation of Instruction Level Parallelism, *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 118–127 (March 1990).
16. B. J. Smith, Architecture and Applications of the HEP Multiprocessor Computer System, *Real-Time Signal Processing* **298**:241–248 (August 1981).
17. M. Girkar and C. Polychronopoulos, Partitioning Programs for Parallel Execution, *Proc. of the ACM Supercomputing Conf.*, pp. 216–229 (1988).
18. V. Sarkar and J. Hennessy, Compile Time Partitioning and Scheduling of Parallel Programs, *Proc. of the ACM SIGPLAN Symp. on Compiler Construction*, pp. 17–26 (1986).
19. Z. Li and Abu-Sufah, On Reducing Data Synchronization in Multiprocessed Loops, *IEEE Trans. on Computers* **C-36**(12):105–109 (December 1987).
20. S. P. Midkiff and D. A. Padua, Compiler Algorithms for Synchronization, *IEEE Trans. on Computers* **C-36**(12):1485–1495 (December 1987).
21. V. P. Krothapalli and P. Sadayappan, Removal of Redundant Dependences in DOACROSS Loops with Constant Dependences, *IEEE Trans. on Parallel and Distr. Syst.* **2**(3):281–289 (July 1991).