A Fine-Grained MIMD Architecture based upon Register Channels

Rajiv Gupta

Department of Computer Science
University of Pittsburgh
220 Alumni Hall
Pittsburgh, PA 15260
E-mail: gupta@cs.pitt.edu

Abstract: This paper discusses the use of shared register channels as a data exchange mechanism among processors in a fine-grained MIMD system with a load/store architecture. A register channel is provided with a synchronization bit that is used to ensure that a processor succeeds in reading a channel only after a value has been written to the channel. The instructions supported by this load/store architecture allow both registers and register channels to be used as operand sources and result destinations. Conditional load, store, and move instructions are provided to allow processors to exchange values through channels in presence of aliasing caused by array references. Compiler support required to take proper advantage of channels is briefly discussed. 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.

Keywords: Parallelizing Compilers, Fine-grained Parallelism, Instruction Scheduling, Aliasing, Channels, Multiprocessor System.

1. Introduction

Compiler detected 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 non-loop parallelism. Shared memory multiprocessor systems, such as the Encore, can exploit loop level parallelism effectively. However, they cannot exploit non-loop parallelism present in the sequential parts of a program. The Very Long Instruction Word (VLIW)[2,3,5] architectures are a family of architectures that can effectively exploit fine-grained parallelism present in sequential parts of a program. Compilers for VLIW machines can detect and schedule non-loop parallelism in sequential parts of the program and also exploit loop level parallelism through loop unrolling and software pipelining. 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 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 has two main 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[8]. The processors on this chip can execute relatively independent streams of instructions for exploiting loop level parallelism and also exploit non-loop parallelism efficiently. The interconnection potential of a multiprocessor chip is being exploited to provide highly efficient register channels. A combination of a mechanism for **collective branching**[8] of processors and data passing mechanism based upon **register channels** is being used to exploit non-loop parallelism. A register channel is provided with a synchronization bit which enforces proper synchronization during the sending of a value from one processor to another processor. 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

0194-1895/90/0000/0028/$01.00 © IEEE

operation of processors. Thus, delays caused by unpredictable operations in a VLIW machine can be potentially reduced. The MIMD nature of the architecture also allows loop level parallelism to be exploited effectively. The fuzzy barrier mechanism is suported in hardware to allow synchronization of processors during the execution of parallel loops. This mechanism provides tolerance to unpredictable delays in the progress of different streams. Compile-time techniques are used to find useful instructions that a processor can execute while it is waiting for other processors to arrive at the barrier.

Techniques such as trace scheduling[4], region scheduling[9], and software pipelining[1] developed for LIW architectures will 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 a set of registers that are shared by all the processors. In absence of aliasing the compiler can precisely determine when a value computed by one processor will be required by another processor. Thus, it can use a register channel for the communication of the value. This avoids memory load/store instructions that would have been introduced if the values were communicated through shared memory. In presence of aliasing due to array accesses memory load/store operations can still be avoided using conditional channel load/store operations. Since the channels are addressed as registers only a fixed number of channels can be provided. This work demonstrates the use of a fixed number of channels for exploiting fine-grained parallelism.

The HEP[10, 11] 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 equal the number of processors in the system. In such a system it is essential to provide fast channels as memory latency cannot be hidden by switching among different streams. 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 overview of the channel based architecture and the instructions supported by the processors to allow the use of register channels are discussed. We demonstrate the usefulness of these instructions in reducing memory load/store operations even in the presence of aliasing. Finally, a brief overview of instruction scheduling and channel allocation process is provided.

2. Overview of the Architecture

The multiprocessor chip being developed at Briarcliff contains a small number of RISC processors[8]. The interconnection potential of a multiprocessor chip is being exploited to provide highly efficient hardware synchronization mechanisms. The multiprocessor support provided preserves the load/store nature of the processor architecture. The system is tolerant of unpredictable delays in the progress of individual instruction streams. The combination of data passing mechanism based upon **register channels** and a mechanism for **collective branching**[8] of processors is being used to exploit non-loop parallelism without strict lockstep operation of processors. Thus, delays caused by unpredictable operations in a VLIW machine can be potentially reduced.

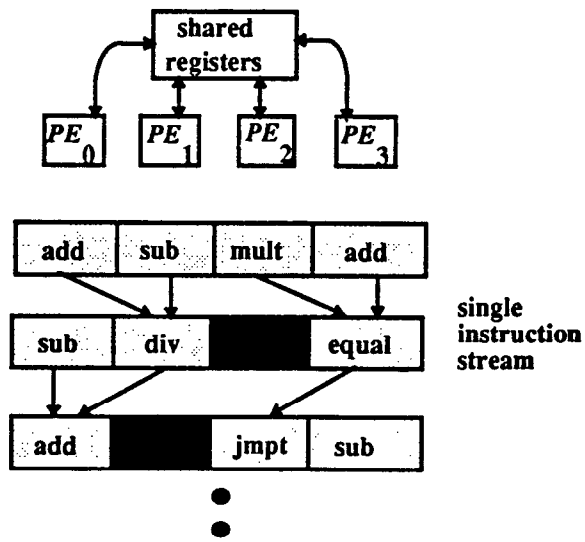

Figure 1: VLIW Architecture

Each processor is provided with a local set of registers and is also allowed access to a set of register channels that it shares with all the processors in the sys-

tem. In Fig. 1 it is shown how processing elements in a VLIW architecture communicate through a shared set of registers. In Fig. 2 a MIMD architecture based upon register channels is shown. The shared set of registers in Fig. 1 have been replaced by a shared set of register channels. The processors in Fig. 2 communicate through the register channels but they no longer have to operate in strict lockstep fashion. 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 to the channel. The register channels are addressed and accessed in the same manner as local registers available to each processor. Thus, using register channels the processors can communicate at high speed. The communication of values through channels enforces synchronization.

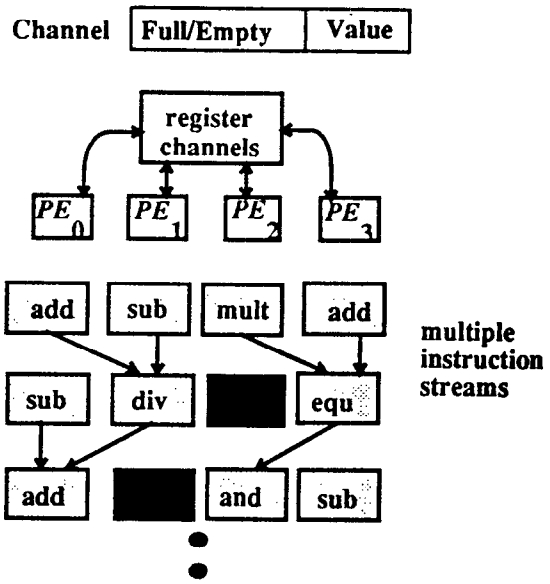

Figure 2: MIMD Architecture with Register Channels

By examining the data dependency graph for a trace, which is a sequence of basic blocks along an execution path in a program, the compiler determines which operations can be executed in parallel and then schedules them 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 through a pointer p must not precede an assignment through pointer q because p and q may be aliases for the same data element. A channel can be used to signal the completion of the assignment as opposed to sending a data value. A processor can read a value that it itself wrote to a channel because the hardware does not keep track as to who is reading or writing to a channel. Thus, the channels can also be used as a substitute for local registers in situations where a processor requires more registers than the number of local registers and channels are not being used. The channels are also useful for executing loop iterations in parallel. Across processor loop carried dependencies can also be enforced through channels although this is not discussed in this paper.

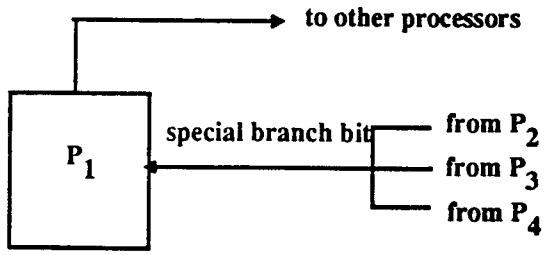

Figure 3: Collective Branching Mechanism

The collective branching mechanism allows a single processor to control the branching of all the processors through a global condition code that is replicated at each processor. Modification of this global condition code by any one processor causes the broadcast of the global condition code which changes the condition code at all the processors. The compiler schedules the evaluation and testing of the condition on any one processor. The result of this test is then broadcast to the remaining processors in the system through the modification of the global condition code. Hardware support is provided to broadcast the branch bit. New instructions for setting this global condition code bit and performing the collective branch are added to the processor instruction set.

Corresponding to every regular instruction that modifies the condition code, another special instruction, *test and broadcast* (*tstbd*), is provided. Execution of *tstbd* not only sets the appropriate bits of the condition code belonging to the executing processor, but also broadcasts it to all other processors in the system. The processor that tests the condition then executes the instruction *branch on local condition code* (*blcc*). The remaining processors that receive the branch bits then branch based upon the broadcast bit by executing the instruction *branch on special condition code* (*bscc*). Synchronization is needed to ensure that a new branch bit is not sent to a processor until the previous branch bit has been used. This synchronization is enforced using the fuzzy barrier[6]. The fuzzy barrier synchronization ensures that no processor can execute any instruction following a barrier region till all processors have com-

The shaded regions represent fuzzy barriers.

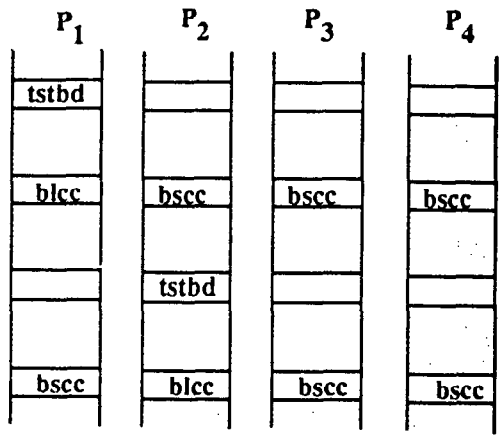

Figure 4: Achieving Collective Branching with Fuzzy Barriers

pleted the execution of instructions preceding their respective barrier regions. The presence of barriers is indicated by a special bit in each instruction. If this bit is one the instruction falls in the barrier region and if it is zero it is outside the barrier region.

In the example shown in Fig. 4, processor P_1 computes the branch condition, by executing the instruction *tstbd*, and all processors are using the same bit to branch together. Synchronization is needed to ensure that a new branch bit is not sent to a processor until the previous branch bit has been used. This is achieved through a fuzzy barrier represented by shaded regions in Fig. 4. The first barrier separates the *tstbd* instruction and the corresponding *bscc* instructions to ensure that processors P_2, P_3, and P_4 execute the branch instruction after the condition has been tested by P_1. The second barrier ensures that the branch bit is not overwritten by the next execution of *tstbd* by P_1 until processors P_2, P_3, and P_4 have executed the branch instruction. By using the fuzzy barrier the streams are still allowed to drift as the processors may execute their respective branch instructions at different times. In this case, the compiler should schedule operations in such a manner that the branch condition is evaluated as early as possible. This will reduce the likelihood of processors stalling at the barrier. The fuzzy barriers are also used to enforce across processor data dependencies arising from parallel execution of loops with loop carried dependencies. A value computed by a processor and written to memory prior to barrier synchronization is available to the remaining processors after barrier synchronization.

The above approach maintains the desirable characteristic of allowing the streams to drift relative to one another. The drift is allowed because the processors are not forced to execute their respective branch instructions simultaneously. On the other hand, in VLIW machines the processors operate in strict lockstep mode. Thus, if any one of the processors takes longer to complete its operation all processors are delayed. Unexpected delays are caused in VLIW machines if the memory bank containing the data value required for an operation cannot be predicted at compile-time. After the global condition code has been set, the processors execute their respective branch instructions in the order they arrive at the branch instruction.

3. Channel Operations

The instruction set for the processors has been extended not only to allow for collective branching but also to perform channel operations. The new instructions added to the RISC processors maintains the load/store nature of the architecture. Channels are read and written by instructions in the same manner as ordinary registers. Fig. 5 shows a typical RISC instruction which reads operands from $source_1$ and $source_2$ and after performing an operation specified by the *opcode* writes the result to the destination *dest*. The source and destination fields refer to registers in uniprocessor RISC architectures. In our architecture they refer to either a register or a register channel. As indicated in Fig. 5 the bit *REG/CHL* indicates whether the field refers to a register or a channel and the field *REG/CHL ID* provides the specific register or channel id. The bit *DES/NDES* specifies whether the read or write to a channel is a destructive operation or a non-destructive operation. These operations are later discussed in greater detail. Although 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. This restriction is enforced by the compiler which is responsible for the allocation of the channels. The read and write operations that can be performed on the channels are as follows:

CLEAR - The channel is cleared by setting the synchronization bit to zero which indicates that the channel is empty. This instruction is used to initialize channels to empty at the beginning of the program.

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.

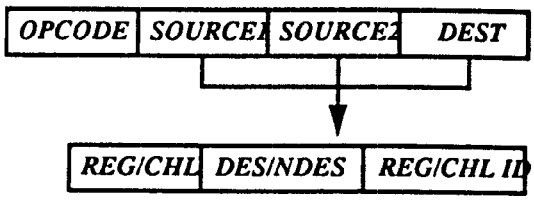

Figure 5: Register Channel Read/Write Instructions

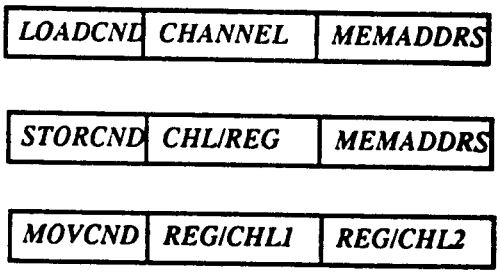

Figure 6: Conditional LOAD, STORE, and MOVE Instructions

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.

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.

DESTRUCTIVE WRITE - The value is written and the synchronization bit is set to one indicating that the channel is full. Thus, if the channel was full prior to a destructive write the old value is destroyed.

Some conditional instructions are provided to allow generation of efficient code in the presence of potential aliasing. In uniprocessor architectures correct execution of code is achieved by not assigning a register to potential aliases. Thus, every time an alias is used/computed its value is read/written from/to memory using a load/store instruction. Load/store instructions create memory traffic and the code takes longer to execute as compared to the code generated when registers are assigned to the variables. In a multiprocessor system the problem of memory traffic is even more severe and therefore it would be desirable to reduce memory traffic by avoiding loads/stores even in the presence of aliasing. Although a data element can have an arbitrary number of aliases, in practice most data elements have only two aliases. If there is a potential aliasing problem due to two array accesses then conditional load, store, and move instructions can be used to generate efficient code. These new instructions are described next and later in this section their use is illustrated through an example.

Conditional Load (LDCND) - If the specified bit in the condition code is one, i.e., the condition is true, then the load is not issued. On the other hand if the condition is false the value from the specified memory address (*MEMADDRS*) is loaded into the specified channel (*CHANNEL*).

Conditional Store (STCND) - If the specified condition is true the store is not carried out. If the condition is false the value from the specified channel or register (*CHL/REG*) is stored into the specified memory address (*MEMADDRS*).

Conditional Move (MVCND) - If the specified condition code bit is one, i.e., the condition is true, then the contents of (*REG/CHL* 1) are moved to *REG/CHL* 2, otherwise the move is ignored.

3.1. Read and Write Operations

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 used more than once non-destructive reads are used in all but the last use and a destructive read is used during the last use of the value. In the example shown in Fig. 7(i) processor P_1 computes the value of A and send it to P_2 through a channel. This value is read using a non-destructive read by P_2 because it is required by P_1 later on. In the example shown in Fig. 7(ii) processor P_1 computes the value of A which is used by processor P_2 twice. Processor P_2 performs a non-destructive read during the first use and a destructive read during the last use.

If a value written to a channel is no longer useful it can be overwritten using a destructive write. In the example shown in Fig. 8(i) the value of A computed by P_1 is used by P_2 is the true branch is taken. Thus it is

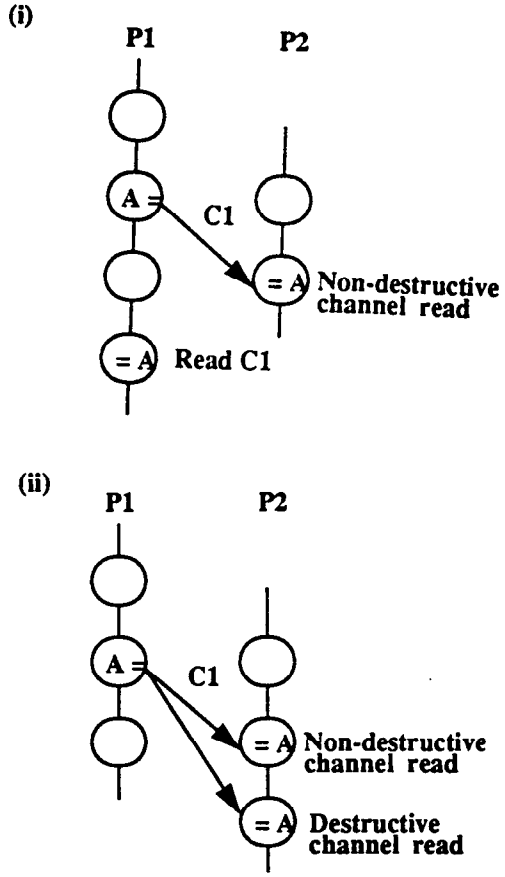

Figure 7: Channel Read Operations

written to a channel. However, if the false branch is taken the value of A is not live. In this branch the same channel can be used in the false branch. However, a destructive write to the channel must be used so that the old value of A can be discarded in the process. In the example shown in Fig. 8(ii) the first value of A computed by P_1 may or may not be used by P_2. When P_1 computes A the second time the channel C_1 may or may not be empty. Thus, P_2 performs a destructive write on channel C_1 to pass the second value of A to P_2. These examples demonstrate that if appropriate choice of channel operations is made the channels can be used as efficiently as local registers.

The decisions to use destructive or non-destructive read and write informations can only be made if global data flow information is available. The def-use and use-def chains, which provide us with a list of uses for a definition and list of definitions that reach a particular use, are computed. Based upon this information it can be determined whether a use of a definition is the last use or additional uses may follow.

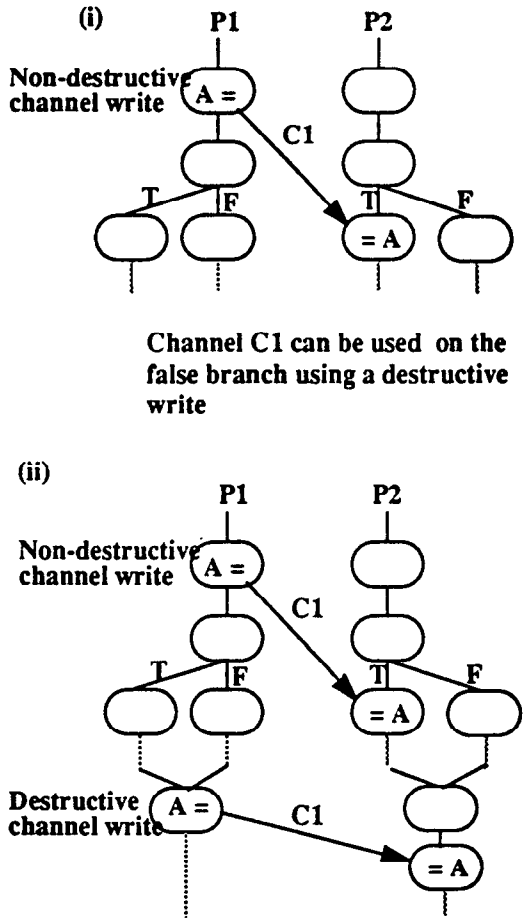

Figure 8: Channel Write Operations

3.2. Conditional Operations

Consider the fragment of sequential code shown below in Fig. 9. If the compiler cannot determine that i and j always have different values then $a[i]$ and $a[j]$ can be potential aliases for the same array element. Thus, the assignment to $a[i]$ must be carried out prior to an access to $a[j]$ to ensure correct results. Traditionally in such cases the store for statement S_1 is generated and a load for statement S_2 is generated (see Fig. 9(i)). However, it would be preferable if in situations where i and j have the same value the value of $a[i]$ is kept in a register and used during execution of statement S_2. In context of the fine-grained architecture if statement S_1 and statement S_2 are being executed on different processors then we would like to pass the value of $a[i]$ from one processor to another through a channel if i and j have the same value. The conditional load and move instructions allow us to achieve this effect.

In the code shown in Fig. 9(ii) P_1 computes $a[i]$ into a channel so that it is available to processor P_2. A conditional load is used by processor P_2 to avoid issuing a load instruction if i and j have the same value. To

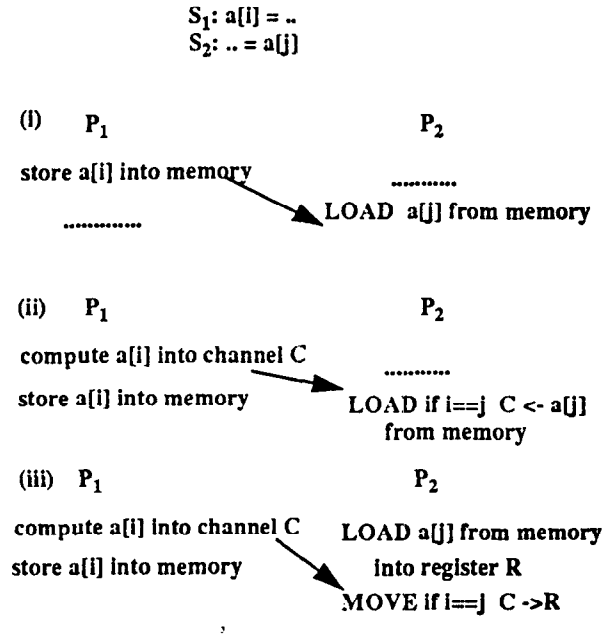

Figure 9: Conditional Load and Move Operations

further hide the latency of memory access due to the load performed by processor P_2 the approach shown in Fig. 9(iii) can be taken. The value of $a[j]$ is loaded by P_2 into a local register R. A conditional move is used to overwrite this value by the value of $a[i]$ computed by P_1 if i and j have the same value. The comparison of i and j can be scheduled early in the schedule by the compiler. Thus, if aliasing is absent processor P_2 can execute the conditional *MOVE* instruction even before the value of $a[i]$ has been computed by P_1. Using the conditional load and move instructions we have successfully replaced an **unconditional synchronization** in the schedule of Fig. 9(i) by a **conditional synchronization** in the solutions described in Fig. 9(ii) and 9(iii). This not only improves the quality of the schedules because it avoids unnecessary delays due to synchronization of the processors but also reduces memory traffic. However, there is some additional cost associated with conditional synchronizations. In Fig. 9(ii) anf 9(iii) a test to check whether i was equal to j had to be performed. However, the test need not be a part of the conditional load/move instruction. The test (i==j) could have been performed prior to the conditional load/move instruction and during the conditional load/move instruction the appropriate bit in the condition code could have been examined.

The example in Fig. 10 illustrates the use of a conditional store instruction. Again we assume that $a[i]$ and $a[j]$ could be potential aliases for the same array element. In the traditional approach the stores for S_1 and S_2 will be performed in the correct order to ensure

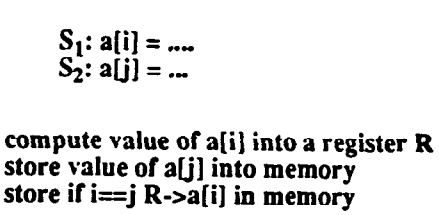

Figure 10: Conditional Store Operation

correct results. Using a conditional store instruction the first store to memory can be eliminated as follows. The value of $a[i]$ is computed into a local register R. The value of $a[j]$ is stored into memory. The value $a[i]$ in R is now stored into memory using a conditional store. Thus, if i and j have the same value the store will not be issued and memory traffic will be reduced. Thus, using conditional instructions the memory traffic can be reduced in presence of aliasing. Furthermore, the values are communicated between processors using channels whenever possible.

4. Compiler Support for Utilizing Channels

Code generation for the fine-grained MIMD architecture described in this paper is based upon trace scheduling and it involves three major steps[4]. In the first step the compiler constructs traces, which are sequences of basic blocks that lie along an execution path in the program and they do not cross loop boundaries. A data dependency graph for the trace is constructed and an *instruction schedule* is generated exploiting the parallelism available in the data dependency graph. The compiler repeatedly traces out paths and generates instruction schedules till the entire program has been processed. By examining traces, instead of individual basic blocks, the compiler can exploit parallelism across basic blocks. Every time two processors exchange a data value synchronization may not be required. Through analysis the compiler identifies and eliminates *redundant synchronizations*. Finally the compiler carries out *channel assignment* to all non-redundant synchronizations. Next the three steps mentioned above are briefly described. Further details can be found in[7].

4.1. Instruction Scheduling

An instruction scheduling algorithm that generates schedules for exploiting parallelism has been developed. The schedules that require less communication among the processors are chosen. This is because a fixed number of channels are available in the system and we would like to enforce maximum number of cross-processor dependencies using these channels. Some of

the key features of the scheduling algorithm are discussed next.

Consider an operation in a DAG that receives its two operands from two other operations in the same DAG. If all three operations are assigned different processors two channels will be required to enforce the data dependencies due to the two operands. Without sacrificing any parallelism 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 the number of channels required to one channel.

For a computation containing more parallelism than the processors in the system are able to exploit schedules requiring a larger number of channels may exist. However, equally fast processor assignments requiring fewer channels may exist. 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. 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. When an entire subgraph is scheduled the operations are scheduled by traversing the graph in a top down and breadth first fashion. By scheduling the operations in the above fashion the number of channels needed is reduced.

We would like the scheduler to generate schedules that distribute the work equally among the processors and hence provide earlier finish times. This feature is easily encorporated into the scheduling algorithm 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. Thus, this scheduling algorithm tries to minimize the number of channels needed without sacrificing the degree of parallelism exploited.

After the generation of an instruction schedule some bookkeeping tasks must be performed. 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[3].

4.2. Eliminating Redundant Synchronizations

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. 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. The redundant synchronizations are arcs that can be eliminated by examining those arcs that result from conputing the transitive closure of other arcs. 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)->(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 -> is transitive. Thus, the order in which the redundant synchronizations are eliminated has no bearing on the final outcome.

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.

4.3. Channel Assignment

The algorithm described here tries to minimize the number of channels used by reusing the channels. Initial discussion will assume that unlimited number of channels are available. However, later a modification to the algorithm so that it functions for a fixed number of channels is described. 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 channels can be safely reused only if certain conditions are true. In particular 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 at compile-time and the writes are not performed by the same processor. 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. A channel is used for writing by a processor only if 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 technique described above 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. As long as there is a single channel dedicated from each processor to every other processor, any schedule can be correctly executed. This observation 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. Assignment of the same channel to enforce all remaining dependencies from one processor to another in the final step of the algorithm will result in schedules that may execute slower as a processor may have to wait between performing successive writes to the channel.

5. Summary and Conclusion

In this paper a MIMD architecture based upon register channels was described. This architecture exploits fine-grained parallelism in sequential programs in a manner similar to VLIW machines. The use of channels should provide improvement in performance over VLIW machines as the multiple processors are no longer constrained to execute in lockstep. The use of register channels in straightline code allows processors to drift and thus provides tolerance to delays caused by events unpredictable at compile-time. The collective branching mechanism together with the fuzzy barrier allows processors to drift across branch instructions. Compile-time techniques for allocation of register channels were presented.

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.

References

1. A. Aiken and A. Nicolau, "Optimal Loop Parallelization," *Proceedings SIGPLAN'88 Conf. on Programming Language Design and Implementation, ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 308-317, June, 1988.
2. 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.
3. J.R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1986.
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, "A Reconfigurable LIW Architecture," *Proc. of the International Conf. on Parallel Processing*, pp. 893-900, August, 1987.
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," *Proceedings of the Second ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, pp. 118-127, March, 1990.

8. R. Gupta, M. Epstein, and M. Whelan, "The Design of a RISC based Multiprocessor Chip," *Proceedings of Supercomputing'90, New York*, November, 1990.

9. 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.

10. B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Real-Time Signal Processing*, vol. 298, pp. 241-248, August, 1981.

11. J.A. Solworth, "The Microflow Architecture," *Proc. of the International Conference on Parallel Processing*, vol. I, pp. 113-117, August, 1988.