International Journal of Parallel Programming, Vol. 22, No. 3, 1994

Code Scheduling for Multiple Instruction Stream Architectures

Gary Tyson[1] and Matthew Farrens

Received June 24, 1993

Extensive research has been done on extracting parallelism from single instruction stream processors. This paper presents our investigation into ways to modify MIMD architectures to allow them to extract the instruction level parallelism achieved by current superscalar and VLIW machines. A new architecture is proposed which utilizes the advantages of a multiple instruction stream design while addressing some of the limitations that have prevented MIMD architectures from performing ILP operation. A new code scheduling mechanism is described to support this new architecture by partitioning instructions across multiple processing elements in order to exploit this level of parallelism.

KEY WORDS: Compiler scheduling; instruction level parallelism; stream processing; decoupled design; multi-issue architecture.

1. INTRODUCTION

The compiler and code scheduler for a *multi-issue* architecture requires a high degree of sophistication in order to realize the full potential for parallel execution. It must be able to assign independent instructions to operational units in a manner that minimizes the number of cycles in which no instructions can be issued. The task of the scheduler in a multi-issue system is further complicated by the fact that while the latency of operational units and memory remain fixed, the number of instructions that must be scheduled in a period is increased by the width of the issue stage.

[1] Computer Science Department, University of California, Davis, California 95616. (e-mail: tyson@cs.ucdavis.edu, farrens@cs.ucdavis.edu).

0885-7458/94/0600-0243$07.00/0

Several studies[1,2] indicate that compilers using simple scheduling techniques are capable of identifying 2-3 independent instructions per cycle. Other studies[3,4] suggest that even more parallelism can be found if the compiler's scheduler is capable of performing extensive code motion.

In this paper, we will present a brief overview of single and multiple instruction stream approaches to multiple issue processor design. We will then introduce the basics of a multiple instruction stream/multiple issue architecture we have developed, show how code is scheduled for several loops, and present an analysis of the performance of this architecture.

2. MULTIPLE INSTRUCTION ISSUE ARCHITECTURES

Several distinct approaches have been taken in the development of multiple issue architectures. The *Very Long Instruction Word (VLIW)* approach increases the resources available to (and the demands on) the compiler. It is responsible for *all* scheduling, including the assignment of null operations to functional units that cannot be assigned a useful task during a given cycle. Since the compiler has the most complete information about the entire program, it is well suited to deal with the inclusion of additional resources (e.g., ALUs, FPUs, and I/O units) and is able to increase instruction execution bandwidth in areas of the code that were previously performance limited by resource constraints. However, VLIW does not support out-of-order execution well, and any change of the hardware description requires all code to be recompiled in order for the program to work correctly.

Superscalar architectures employ a hardware scheduler that uses dynamic run-time information in order to efficiently allocate resources to the list of instructions ready for execution. However, the hardware implementation of the scheduler is restricted to selecting instructions from a fixed-size window of available instructions, and thus does not have the breadth of information available to it that the compiler does at compile time.

A third approach to issuing multiple instructions takes advantage of the characteristics found in the Von Neumann computational model. *Decoupled* architectures attempt to exploit the independent nature of *control flow*, *memory access*, and *data manipulation* operations that comprise conventional computations by splitting a task into distinct pieces and executing them on separate pieces of hardware. Since these hardware units communicate via FIFO queues, the instruction streams are allowed to slip with respect to one another, providing dynamic support for out-of-order execution. This approach attempts to take advantage of the best that VLIW and superscalar have to offer; the compiler partitions the tasks in a manner similar

to VLIW, and the queues provide the same dynamic scheduling benefits found in superscalar.

These decoupled systems differ from VLIW and superscalar designs in the manner which the independently issued instructions interact. VLIW and superscalar processors can be thought of as very tightly coupled shared memory systems; they share not only addressable memory but also register space. This shared register approach differs from the explicit message passing (via FIFO ordered queues) found in decoupled machines. Furthermore, in order to transmit data among operational units by writing and then reading the contents of a register, the clocks on VLIW and superscalar processors must be synchronized. This requirement is relaxed with an explicit message passing approach.[5]

The greater flexibility found in a decoupled design allows both single and multiple instruction stream descriptions of a task. The **ZS-1**[6] and **WM**[7] systems operate in a decoupled manner while receiving instructions from a single instruction stream. Their architectural component descriptionsare similar to those of *Split Register* superscalar designs.[8,9] The **PIPE** machine,[10] in contrast, consists of two PIPE processors[11] which run asynchronously, each with their own instruction stream, and cooperate on the execution of a single task.

3. EXPLOITING ILP ON A MIMD ARCHITECTURE

Parallelism in a single instruction stream architecture resides primarily at the instruction level and is a well-studied problem.[1,12] Extracting parallelism on a MIMD architecture, on the other hand, has traditionally been accomplished by partitioning the program into data independent portions and assigning them to separate processing elements, ignoring any other parallelism that might exist. Little research has been done on exploiting instruction level parallelism across processors on a multiple instruction stream machine.

There are a number of reasons why this approach merits further investigation, however. Superscalar machines do not scale well—expanding the number of processing elements available necessitates a corresponding increase in the size of the hardware window over which code scheduling occurs, significantly increasing the scheduling complexity. Compilers for VLIW machines can help circumvent this problem, but do not support out-of-order execution well.

Exploiting instruction level parallelism on MIMD architectures can overcome both these problems. The instruction issue stage of each processor can perform in a simple single-issue, in-order manner, avoiding much of the hardware complexity required to support out-of-order issue in

a single instruction stream approach. Out-of-order issue is also supported on an MIMD because the processors are run independently; therefore, any independent instructions executed on different processors can issue in any order without necessitating any hardware support. This is fundamentally different from multiple issue in a VLIW machine because a strict ordering of instructions is not imposed by the compiler unless a dependence exists. Furthermore, by incorporating multiple program counters, a MIMD machine provides the architecture with more dataflow information by enriching the specification of the object language; taken to its extreme this would allow a dataflow machine description of the program.

Separating a program into multiple single issue instruction streams additionally allows the decentralization of the hardware resources, since there is no central instruction window from which instructions are issued. Similarly, there is no central register file to be overloaded with contention among the processing elements, which allows for easier expandability in a MIMD approach.

While a MIMD approach to code scheduling clearly possesses certain advantages, historically these architectures have suffered from severe limitations. Data transfer latencies have been high, and the bandwidth required to support high-throughput, low contention data transfer has been unavailable because of pin limitations and/or board-level interconnects. Even if maximum data transfer rates can be made acceptable, the need to provide synchronization points can cause unacceptable performance loss. Using main memory to handle data transfers between processors can also lead to an unacceptable dependence on memory latency. These problems help explain why current MIMD designs do not exploit ILP.

Increasing the number of transistors that can be fabricated per square centimeter provides the means by which many of the interprocessor communication problems can be eliminated. Placing several of these processing elements on the same die circumvents the pin limitations on bandwidth and supports high on-chip data transfer rates. In addition, using FIFO queues in a manner similar to that used by decoupled machines provides a clean way to handle synchronization. If transistor densities continue to increase as they have over the last decade, by the middle of this decade such a design will be realizable. One study[13] indicates that as tens of millions of transistors become available, something more than simply increasing on-chip cache sizes must be done. These facts led to the design of the MISC architecture, a decoupled MIMD machine that is designed to support and exploit instruction level parallelism.

4. THE MULTIPLE INSTRUCTION STREAM COMPUTER (MISC)

The MISC architecture was designed to handle many of the dynamic characteristics of program execution by allowing the compiler to convey more information to the hardware during code translation. Variable operational unit latencies (primarily memory loads) create difficulties for code scheduling in VLIW and superscalar processors, due to the sequential instruction flow imposed by translating a dataflow intermediate representation to a single instruction stream architecture. Superscalar designs can remove some of the restrictions imposed by single stream scheduling by regenerating some the dataflow information at the issue stage of the pipeline, but not without considerable hardware issue logic. Furthermore, software pipelining[14] and loop unrolling schemes[15] have difficulty in efficiently scheduling instructions with variable latency dependencies.

MISC avoids these scheduling problems by allowing operations with indeterminate latencies to transfer data between PEs. The inherent asynchronous relationship among the PEs can compensate for the variability of the latency without affecting the execution rate of nondependent instructions.

The MISC processor has been described in detail in Ref. 16. A brief overview of MISC will be presented here, focusing on aspects of the architecture that will be featured in the code scheduling discussion later in the paper. MISC is a direct descendant of the PIPE project, but unlike the two processor PIPE design, the MISC system is capable of balancing the processor load of instructions performing control flow, memory access, and execute operations among multiple processors. As its name indicates, MISC is composed of multiple *Processing Elements* (PEs) which cooperate in the execution of a task.

The example MISC configuration used throughout this paper consists of four processing elements, a bank selected data cache (DCache), and a set of internal data paths used to transmit data among PEs and the DCache. The component design of this MISC configuration is illustrated in Fig. 1. Each PE executes in an asynchronous manner from other PEs and the DCache. The internal data paths are used to facilitate communication between elements (PEs and/or DCache). Each data path is controlled by a single element; for instance, the internal data path labeled PBUS1 is controlled (written) solely by PE1. Each PE has its own bus (PBUS{1–4}), and the data cache controls two busses (CBUS1 and CBUS2). Two separate bank selected I/O channels support the transmission of data between the MISC chip and the rest of the system (main memory). These channels are controlled by the DCache. Each PE is capable of transmitting

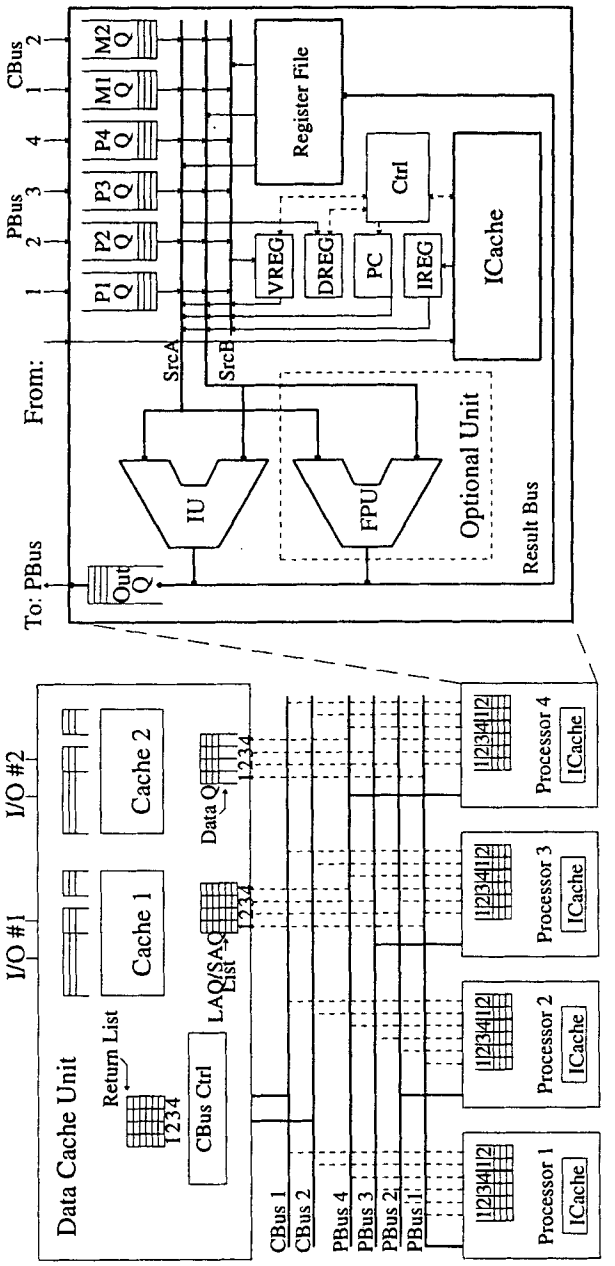

Fig. 1. MISC machine and the MISC processing element.

a message directly to any other processor (including itself), or of broadcasting a message to all processors.

The processing elements (Fig. 1) are collectively responsible for the execution of a single task, with each PE having its own independent instruction stream and its own instruction cache. Each PE is identical and maintains all state information required to function as an independent processor—in fact, the MISC hardware is capable of running four completely unrelated tasks in parallel. However, it is assumed that a single task will be partitioned (by the compiler) into four instruction streams that cooperate in the execution of that task. Each PE contains a 5-stage pipeline, 32 General Purpose Registers (GPRs) which are available for data storage that persists over multiple references, a FIFO processor queue (PQ) for each PE in the system (including itself) to store data transfers between PEs, a *Program Counter* (PC), a *Vector Register* (VREG), and 2 memory queues (MQs) which contain data requested from memory. (The size of each of the queues (PQs and MQs) is not given here; however, their size is an architecturally visible component. The compiler must know the size of each queue (they need not all be the same size) in order to schedule code correctly and avoid deadlocks due to resource depletion.)

All instructions in MISC are 32 bits in length and have three 6-bit source operand fields and a 6-bit destination operand field. In addition, many instructions allow for two of the source fields to be replaced by a 12-bit constant. Each source field can address any of the 32 GPRs, a PQ, a MQ, the PC, the VREG, or a small signed constant (-16 to 15). The destination specifier may address a GPR or *routing* information for a data transfer onto the PE's PBUS. At the instruction issue stage, if a queue is specified as a source input and that queue is currently empty, that instruction is delayed until all required input operands are available.

Since MISC uses a decoupled approach to memory operations, the load and store instructions are different from those in a conventional machine. Invoking a memory read operation provides the DCache with the memory address to be read, the set of destination PEs that are to receive the data, and which CBUS (and therefore MQ) will receive the data. In a Load Address Queue (LAQ) instruction, the *dest* field contains the set of PEs that are to receive the data: the *dest* field should not specify a register. The address requested is the sum of the *src1* and *src2* operands. There are both LAQ and LAQ2 instructions, in order to specify which MQ of the destination PEs should receive the data. The store request (SAQ) instruction operates in a similar manner, except that the *dest* operand specifies a single PE from which the DCache should receive data to be written to memory.

There are three types of MISC instructions: predicated operations,[17]

vector operations, and sentinel operations (which use a sentinel value to terminate iterations of the instruction). Predicated ALU/FPU operations perform all scalar operations as well as allow conditional operations to be specified concisely. A predicate operation uses the third source field to determine whether or not the operation will complete (and thereby change the state of the machine). This allows the issue logic to proceed without interrupt through short segments of conditionally executed code by conditionally completing instead of branching around code.

In the case of control flow operations, the *dest* field is used as a constant to determine the number of *delayed branch* slots[18] to be filled. The address of the branch is calculated as the sum of the *src1* and *src2* operands, and the *src3* operand specifies the register to be tested.

Vector instructions use the third source operand (*src3*) to specify a vector count. When a vector instruction arrives at the issue stage of the instruction pipeline, the vector register (*VREG*) is cleared and a vector count register (*VCOUNT*) is loaded from *src3*. The scalar version of the vector instruction is then executed and the *VREG* is incremented until the contents of *VREG* are equal to *VCOUNT*. Once the *VREG* equals *VCOUNT* normal instruction pipeline function continues.

In a sentinel instruction, the *src3* field specifies a register whose contents are compared to the sentinel value (assumed to be zero in the initial design). If the contents of the register do not match the sentinel, the scalar version of the instruction is allowed to issue. This pattern of compare and issue is repeated until the comparison produces a match.

5. RELATED WORK

There have been several decoupled compilers that have been developed. These include the original PIPE compiler,[19] the WM *streams* compiler,[20] and the compiler for the Briarcliff Multiprocessor.[21]

The PIPE compiler separates code into *access* and *execute* instruction streams. This is accomplished by assigning each branch and memory access operation to the *access* processor, then examining a Program Dependence Graph (PDG)[22] to determine which additional branch control calculation operations and address calculation operations should also be assigned the *access* processor. All remaining instructions, as well as duplicate branch operations, are then assigned to the *execute* processor. Once this separation is accomplished, register allocation and other optimization transformations can be applied to each instruction stream.

The WM compiler is more conventional in its use of a single instruction stream. Dataflow analysis and many of the optimization transformations performed are unchanged from *standard* RISC architectures.

Additional restrictions must be placed on the register allocation method to allow for the nature of the memory queues found in this decoupled architecture. The decoupled nature of this processor is found in the dynamic separation of the instruction stream performed during execution.

The compiler used in the Briarcliff Multiprocessor performs in a much different manner from the previous two compilers. This compiler is far more aggressive in separating code into multiple instruction streams. This machine shares many characteristics of a restricted dataflow architecture.[23] Instructions are partitioned equally over the available processing elements with those data dependencies that exist between PEs being allocated a register channel.[21] Optimization is then performed to reduce the number of channels required without degrading code performance. Memory operations can also be performed on register channels. This allows for decoupled memory access in which one PE performs the address calculation and memory request for data that is destined for a different processor. The Briarcliff design bears more resemblance to a VLIW architecture than a decoupled architecture in its treatment of control flow operations. PEs synchronize on branch operations by generating a global condition code used to determine whether to branch or not. While each PE may reach the actual branch instruction on different cycles, no PE can continue to process the next branch operation until each PE has completed its branch decision on the original branch. This *fuzzy barrier*[24] mechanism allows more flexibility than a VLIW implementation but fails to provide the flexibility found in true MIMD approaches like PIPE and MISC.

There are several other designs that attempt to exploit ILP on a MIMD. The XIMD processor[25] adds control logic to each functional unit of a VLIW in order to transform it into a MIMD. However, since it is a VLIW at the core, it requires a high-performance, completely orthogonal, global register file in order to support inter-process communication. The coMP approach[26] is a communication oriented multi-processor that can be operated in VLIW mode. However, it does not allow dynamic slip between processing elements, and the communication ports between processors are only depth 1. The work by Keckler and Dally[27] is similar to XIMD in that a 4-ALU machine is augmented to allow it to run in a MIMD fashion. PEs can write to each other's register files, but no mention of queues is given, and multi-ported register files are still required.

These machines all differ from MISC primarily in that MISC was designed to be latency-tolerant, and MISC does not require global clocking. We feel that distributing a synchronized clock is going to continue to be more and more difficult, and by featuring queues at all the I/O interfaces the MISC design by nature allows different processing elements to run correctly at different rates.

6. THE MISC COMPILER

The *very portable C compiler* (vpcc)[20] under development at the University of Virginia serves as the base compiler for MISC. Existing optimization techniques are used whenever possible; for those optimizations that are unique to MISC, or where existing techniques require modification (e.g., register allocation incorporating queues), care has been taken to maintain the same level of complexity found in current optimizers. The front end of vpcc translates C code into *Register Transfer List* (RTL) form for a single processor MISC machine. This code is highly unoptimized, but is correct and will run on any MISC configuration.

The code generator translates this RTL description of a program into parallel machine code for the MISC machine. An overview of the optimization algorithm is described in Fig. 2.

Once the RTL description of a function has been loaded, many standard transformations (e.g., common sub-expression elimination, code motion, dead code removal) can be performed. It is best to do these transformations at this point, before the complexity of inter-PE dependencies must be considered. Similarly, *if-conversion*[28] can also be performed at this point in order to simplify the control flow. Global dataflow analysis can then be performed, and a PDG built.

The code separation phase partitions the operations required by the program onto multiple (virtual) processing elements in a manner that maximizes the number of processing elements utilized. Processor load balancing then repartitions the schedule to evenly distribute the operations onto the number of physical processing elements available on the target machine. Once the instructions have all been allocated to the PEs, more code

```
foreach function
    load initial RTL description provided by front end
    perform standard single stream transformations and register allocation
    perform if conversion
    build program dependence graph
    separate code
    balance code
    foreach PE
        perform if conversion
        perform loop optimizations
        schedule code
    output MISC machine code for function
```

Fig. 2. An overview of the optimization algorithm.

```
int inner_product() {
        int k, q=0;
        for (k=0; k<1024 ; k++)
                q = q + z[k] * x[k];
}
```

Fig. 3. Livermore loop 3 (inner product).

optimizations can be applied to each of the PE instruction streams. Many of the standard transformations described previously can be reapplied to each individual stream (with new restrictions to maintain interPE dependences added). Finally, each instruction stream is scheduled and the MISC machine code is generated.

The code scheduling method used on MISC uses the asynchronous behavior of the processing elements to provide many of the characteristics found in software pipelining. Individual PEs can be executing instructions originating from different iterations, while the PE queues perform a simplified form of register renaming. Variable latency poses no problem because any instructions dependent on the data can be issued to a trailing PE.

A detailed description of this process follows in the remainder of this section. To illustrate each phase of the code generation process, a simple example (Lawrence Livermore Loop 3) will be used (see Fig. 3).

6.1. Register Allocation

Standard register allocation methods can be used, with one exception: MISC has a large number of *register classes*, unlike most architectures (which have only 2 register classes, integer and floating point). Since each PE has a general purpose register class, two separate memory input queues, and a complete interconnection of interPE transfer queues, a four processor MISC machine would have 28 different register classes ($1\,\text{GP} \times 4 + 2\,\text{MQ} \times 4 + 4\,\text{PEQ} \times 4$). In addition to the large number of register classes, all but the general purpose registers are FIFO queues. This necessitates some restrictions on standard register allocation methods in order to provide correct FIFO ordering of queue use. If the allocation of a new register instance would violate the FIFO ordering of the queue, for example, the allocation is disallowed and the architectural register dependency remains.

In order to provide a compact 2 byte representation of any machine register, the intermediate RTL format employed by the vpcc compiler reserves 4 bits to identify one of 16 different classes and 12 bits to identify the register within that class. Since this format is incapable of accurately

representing a full MISC machine, the MISC register class model must be modified. The modified model supports only unidirectional communication between PEs through the PE transfer queues—PE1 can send data though queues to PE2-PE4, PE2 can send data to PE3 and PE4, and PE3 can send data to PE4. The queues to send data back cannot be represented in the existing RTL format. Fortunately, the code transformation strategy employed in MISC rarely requires data to be transmitted "back" to PE1, and in those cases a transfer through memory can be performed.

Function calls pose an interesting problem within a tightly coupled MIMD architecture. When a function call is made, any variable may be passed as a parameter to the function. One standard compiler technique to improve the performance of function calls is to place the first few parameters in registers before executing the call. However, in MISC variables are distributed among the PEs. How should parameters be passed in MISC during a function call? We chose to place all parameters on the stack—while this does not provide the best performance, it simplifies a number of problems with incomplete dataflow analysis between functions.

6.2. Memory Operation

Memory operations must be carefully scheduled in MISC. Since each PE executes its instruction stream independent of all others (in the absence of a data dependence), memory operations initiated from two separate PE can execute in any order. This poses two distinct problems to the generated schedule:

6.2.1. Two PEs Cannot Initiate Read Memory Operations Destined for the Same PE

It is common to require both operands of an operation (e.g., addition) to come from memory. Data sent directly from one PE to another uses dedicated interPE transfer queues; however, data arriving from memory uses the destination PEs memory queue. Since only a single read is allowed per cycle for each queue, both memory operands cannot be processed in a cycle (a similar problem was discovered in the PIPE compiler). In MISC this problem is solved by providing a second memory queue for each PE. When checking FIFO ordering of memory operations, this second queue is used to alleviate these problems.

6.2.2. Memory Aliasing Cannot Be Resolved if an Ordering of Memory Operations Cannot Be Established

There is no execution order imposed on instructions from different PEs unless a data dependence exists between the instructions. (This differs

from a processor that allows out-of-order execution of memory operations —on such a machine an ordering exists, but is ignored when the memory operations do not conflict.) The code separation phase of the MISC compiler imposes an ordering on memory operations that may conflict by assigning those operations to a single PE. *Conflict Buffers*[29] can then be used to reorder these memory operations during execution to maximize memory throughput. All memory operations that the compiler can guarantee do not conflict can be assigned to alternate PEs, which allows the conflict buffers to ignore these memory requests when issuing the (potentially) aliased memory operations.

6.3. Code Separation

The task of the code separator is to partition the task across processing elements, with the goal of minimizing the effects of high memory latency and high functional unit latency for operations like multiply and divide by decoupling the *definition* of the data item from its *use*. Much like initial register allocation strategies, code separation assumes an infinite number of processing elements. The mapping of operations to the physical processing elements available on the target architecture is left to the processor load balancing phase.

The first step in code separation is to perform global data flow analysis and construct a *definition-use chain* for the RTLs in each function. This chain can then be used to partition RTLs into dependent groups. The primary grouping, referred to as the *control group*, contains all branch operations and the RTLs required to calculate branch conditions and targets (i.e., branch instructions and the instructions on which they depend). The branch instructions are duplicated for each PE in order to maintain a consistent control flow through the code. Later transformations may relax this condition if no data is being manipulated in a block by some PE(s).

Branch condition calculations are assigned to the *lead* PE. The results of these calculations (a simple boolean condition) are then transmitted to the other PEs executing the branch. Branch target calculations are distributed among all PEs performing the branch; however, most of these branch targets can be specified in the branch instruction and therefore do not contribute to code expansion.

The concept of a *leading* (or *lead*) processing element is central to the understanding of code separation. In a MIMD architecture, each of the instruction streams executes independently (ignoring for a moment any data dependences). Therefore, if operations are scheduled carefully, some of the streams can be allowed to proceed farther ahead in the computation than others. Staggering the relative entry cycles for the execution of a

section of code provides a perfect method for hiding the delay imposed by high latency operations. For example, if the instruction that issues a high latency operation is scheduled on a processor that enters that section of code a sufficient number of cycles before the processor that uses the item, the effects of the latency will be hidden. In such a case it is possible for the *leader* PE will be executing instructions in a new section of code while *trailing* PEs are still completing previous sections.

The scheduling of the control group determines a minimum cost traversal through the instruction sequence. Each PE must either follow this control path, or at some later point in the code wait for the *lead* PE to provide a branch condition. Other dependencies in the unscheduled code may (and likely will) further increase the time required to complete execution on some PEs, but the *control group* time is the only limitation on all PEs in the machine. It should be pointed out, however, that the *lead* PE must transmit branch condition information to the other PEs through the interPE FIFO registers, and thus may stall if one or more of these queues are full (and therefore incapable of accepting more data). It is therefore important that all PEs have approximately the same traversal time through their instruction schedule.

Once the *control group* is scheduled, the code separation phase determines how to partition the remaining RTLs. There are several different strategies that can be employed at this point, with each strategy having strengths and weaknesses in generating efficient code and simplifying the task of the later phases. Two strategies will be discussed here: *Latency Removal* and *Group Separation.*

In the *Latency Removal* strategy, a program dependence graph is constructed augmented with latency information. A *ready list* of RTLs that have no outstanding dependences is then examined, and any RTLs that can be scheduled without generating pipeline stalls are assigned to the current PE. Dependency loops (due to loop recurrence variables) are exposed when no items are available in the ready list. These dependency loop operations must be scheduled on a single PE and determine a minimum cost (Initiation Interval) for this PE. Once all RTLs that do not generate stalls are assigned, scheduling continues on the following PE. This technique generates a schedule that minimizes pipeline stalls and tends to separate code into access/execute streams due to the high latency of memory operations.

An alternative approach, *Group Separation*, tries to maximize the number of PEs that are scheduled. Again, a dependency graph is constructed. RTLs are then partitioned into *dependence groups* which contain only mutually dependent RTLs. Each group is assigned to different PEs, with any remaining interPE dependencies forming a Directed Acyclic

Graph (DAG) from *leading* PEs to *trailing* PEs. This representation of the schedule forms a hybrid dataflow representation with control flow shared and interPE data having unidirection flow.

The partitioning algorithm processes the program dependency graph from the *root* (first operation) to the *leaves* (dependent operations). Information concerning operational latency and register use is examined and operations are assigned to processing elements in a bottom-up or dependent-first order. This leads to a reverse schedule in which the final operations for a block are scheduled first and those instructions that use data items that are not available (due to operational latencies) are scheduled on a different processing element.

Branching is handled slightly differently. Branch instructions are duplicated across all processing elements to ensure that each processing element conforms to the same control flow, and all instructions that calculate branch conditions are assigned to the processing element that has no instructions containing data dependencies; this allows a single processor to *lead* the execution.

The algorithm for Code Separation is described in Fig. 4.

The following example shows how the code separation process works using the *Latency Removal* strategy. (For this example, both strategies lead to the same resulting code.)

The RTL representation of the intermediate code prior to code scheduling appears in Fig. 5. Each line of the RTL description either defines a label or describes an operation to be performed in the resulting

```
get RTL description
calculate control group
foreach RTL calculate
        dependence group
        recurrence data dependencies
        resource constraints ( memory FIFO queue ordering )
        memory address disambiguation
if (approach is Latency Removal) then
        foreach group in the DAG with no remaining unscheduled dependencies
        assign to "leading" PE that does not create pipeline stall in the
        PE schedule.
elseif (approach is Group Separation) then
        assign each dependence group to a separate virtual PE

convert inter group dependencies into queue transfers
return modified RTL description
```

Fig. 4. Algorithm code for code separation.

```
[1]   t1 = 0              ; q=0
[2]   t2 = 0              ; k=0
[3]   t3 = 1024           ; set register for test
[4]   t4 = LOC[_z]        ; t4 = base of array z
[5]   t5 = LOC[_x]        ; t5 = base of array x

[6]   L1:
[7]   t6 = (t2>=t3)       ; calculate branch cond
[8]   PC = t6, L2         ; branch if true
[9]   t7 = M[ t4+t2 ]     ; load t7= z[k]
[10]  t8 = M[ t5+t2 ]     ; load t8= x[k]
[11]  t9 = t7 * t8        ; (z[k] * x[k])
[12]  t1 = t1 + t9        ; q = q + (z[k] * x[k])
[13]  t2 = t2 + 1         ; k++
[14]  PC = L1
[15]  L2:
```

Fig. 5. RTL representation of LLL3 prior to code separation.

code. Each RTL line shown throughout this section will contain a comment (delimited by ";") to explain its operation. Virtual register labels (specified as *t1*, *t2*, *t3*, etc.) define intermediate points in the calculation and may or may not map to physical registers or queues. To avoid confusion the actual register mapping has been omitted.

As referred to previously, the code scheduler can use the knowledge that basic blocks are entered by successive processing elements on different cycles to hide the latency of long latency instructions. For example, if an instruction with a completion latency of 5 cycles can be issued by a leading PE with the result destined for a PE trailing by 5 or more cycles, the effects of the operational latency are completely subsumed.

A simple approach the scheduler can take is to assume a fixed latency period between processing element block entries and enforce a strict ordering on PE leadership. This approach can be used to get reasonable performance from the scheduler; however, greater benefit can be found from calculating the expected *stagger* in relative basic block entry/exit cycles. This can be accomplished by augmenting the standard dataflow analysis gathered in the code separation phase with expected completion times for each of the processing elements in the virtual machine.

It is possible that a constant value for the entry time stagger will be grossly erroneous; this is the case when a loop containing a recurrence is scheduled such that only one processing element is delayed by a recurrence relation while the remaining processing elements proceed to the next iteration (and finally to the loop completion). In this situation, the processing

element delayed by the recurrence (and any processing elements dependent on that one) will enter the basic block following the loop termination many cycles later than the lead processing elements. If the code scheduler can recognize when this occurs, a normally less efficient schedule can be created for the basic block succeeding the loop that avoids any use of the processing elements that are lagging far behind. This has the effect of concurrently executing the successor blocks and the loop containing the recurrence. The MISC code scheduler incorporates this augmentation to the dataflow analysis, and the benefits of this approach are discussed in the analysis in Section 6.

In order to determine which operations should migrate to a new processing element, the scheduler locates all data dependencies that have latencies that cannot be hidden with a simple reordering of instructions. From this list of candidate operations, ones that are involved in a dependency circuit are scheduled to a single processing element. This tends to negate the transmission time penalties between processing elements as well as greatly simplifying the deadlock detection scheme in the code generator. Unfortunately, migrating the operation to another processing element may create new hardware dependencies, which may conflict with others. For example, if migrating a multiply operation requires that the destination for the product be a FIFO ordered queue (since queues are used to transfer data between processing elements), a conflict may arise if that queue is already allocated to a previously existing instruction. In these cases, the migration cannot proceed unless that conflict can be resolved in a later optimization step.

This separation tends to identify operations by function: *flow control*, *memory access* or *data manipulation*, with the leader PE performing control flow operations, and trailing PEs supporting memory access and data manipulation. This creates a dependency between the PE that generates the data and a trailing PE which consumes the data, which tends to separate code into memory access and data manipulation functions due to the long latency of memory loads. A graph of these inter-PE dependencies is constructed to aid in the scheduling process, and the scheduler attempts to avoid circuits in the dependency graph to simplify deadlock elimination and to ensure that the ordering of PE execution is maintained (i.e., PE leadership does not transfer).

Code separation provides a model which is capable of balancing PE loads in applications that contain little balance between memory access and data manipulation. It also provides the flexibility to handle code generation for a variable number of processing elements. It should be noted that unlike more coarsely grained parallel computations, the separation of instructions on MISC tends to be between dependent operations.

```
[1]  [PE3]    t1 = 0              ; q=0
[2]  [PE1]    t2 = 0              ; k=0
[3]  [PE1]    t3 = 1024           ; set register for test
[4]  [PE1]    t4 = LOC[_z]        ; t4 = base of array z
[5]  [PE1]    t5 = LOC[_x]        ; t5 = base of array x

[6]  [PE1-3]  L1:
[7]  [PE1]    t6 = (t2>=t3)       ; PE1 calcs branch cond
[8]  [PE1-3]  PC = t6, L2         ; branch if true
[9]  [PE1]    t7 = M[ t4+t2 ]     ; load t7= z[k]
[10] [PE1]    t8 = M[ t5+t2 ]     ; load t8= x[k]
[11] [PE2]    t9 = t7 * t8        ; (z[k] * x[k])
[12] [PE3]    t1 = t1 + t9        ; q = q + (z[k] * x[k])
[13] [PE1]    t2 = t2 + 1         ; k++
[14] [PE1-3]  PC = L1
[15] [PE1-3]  L2:
```

Fig. 6. RTL for LLL3 after code separation.

Applying the code partitioning strategy to the example code in Fig. 5 results in the specification in Fig. 6. A second column has been inserted after each line number to indicate which processing element(s) process this RTL line. For instance, the first line ([1]) initializes the variable **q** to zero. Since the only use of **q** is in code allocated to PE3, the initialization of **q** is allocated to PE3. Notice also that branch operations (lines [8] and [14]) are allocated across all active processing elements. The separation of operations occurs in lines [9] through [12]. Since line [12] has a high latency dependency (due to the multiplication) to line [11] they are partitioned to different PEs. Similarly, the memory latency between lines [9] and [11], and lines [10] and [11] require a third PE to be included in the schedule. Lines [9] and [10] can be issued to the same PE because no dependency exists between them; loop control variable calculation also are issued to the *lead* PE. .

6.4. Processor Load Balancing

The schedule generated by the code separator is incomplete in two areas. First, the partitioning is performed on virtual processors, which may not be equal to the actual number of physical processors. Second, no consideration has been given to equally distributing operations among the processors. The goal of the load balancing phase is to remedy these deficiencies.

There are two aspects to load balancing: instruction balancing and register pressure balancing. In a machine with 128 general purpose registers (as a 4-processor MISC configuration has), register pressure is not a significant problem. In the initial register allocation, prior to code separation, 128 registers are assumed available to the single PE stream. This does not correspond to the actual architecture, but provides a good heuristic on the final variable assignment. During the code separation phase register limitations are ignored, leaving the final task of register assignment to the load balancing phase. The goal is to minimize the need to *spill* register contents to memory, and when spills are required, maintain the performance of the *lead* PEs. This can be accomplished by scheduling the instructions that require spills on trailing PEs.

The desire to balance the instructions over the PEs is far more important. During code separation, groups of instructions were constructed. From this *dependence groups* can be obtained (directly, if the *Group Separation* strategy was used). These groups are then scheduled onto a set of PEs minimizing the variance in instruction count. This problem is similar to *bin-packing*, and a variant of the bin-packing algorithm could be used. We chose to use a greedy method. The method we chose builds a Directed Acyclic Graph (DAG) of dependence groups with each arc of the graph corresponding to an interPE dependence and weighted with the latency of that dependence operation. A ready set of dependence groups is then determined and considered for scheduling on the leading PE. If the size of this group (in instruction count) does not exceed the available instructions on the lead PE, and a pipeline stall does not occur, then the group can be assigned to the lead PE. This process continues until a PE is found for the group. The last PE accepts all remaining (unscheduled) groups. The greedy method has the advantage of allocating memory access

---

get RTL description
construct DAG of dependence groups
while DAG is not empty
foreach group in the DAG with no remaining unscheduled dependencies
    assign to "leading" PE that:
        1) does not place the number of instruction of the block(s) associated with this group to total instruction for block(s)/number of PEs
        2) does not generate an interPE dependency from a trailing PE.
        3) does not create pipeline stall in the PE schedule.
        4) does not force a register spill to occur

Fig. 7. The load balancing algorithm.

```
[1]  [PE4]    t1 = 0              ; q=0
[2]  [PE1]    t2 = 0              ; k=0
[3]  [PE1]    t3 = 1024           ; set register for test
[4]  [PE1]    t4 = LOC[_z]        ; t4 = base of array z
[5]  [PE2]    t5 = LOC[_x]        ; t5 = base of array x

[6]  [PE1-4]  L1:
[7]  [PE1]    t6 = (t2>=t3)       ; PE1 calcs branch cond
[8]  [PE1-4]  PC = t6, L2         ; branch if true
[9]  [PE1]    t7 = M[ t4+t2 ]     ; load t7= z[k]
[10] [PE2]    t8 = M[ t5+t2 ]     ; load t8= x[k]
[11] [PE3]    t9 = t7 * t8        ; (z[k] * x[k])
[12] [PE4]    t1 = t1 + t9        ; q = q + (z[k] * x[k])
[13] [PE1-2]  t2 = t2 + 1         ; k++
[14] [PE1-4]  PC = L1
[15] [PE1-4]  L2:
```

Fig. 8. RTL for LLL3 after load balancing.

consistently to the lead PEs, simplifying memory aliasing analysis. Fig. 7 describes the load-balancing algorithm.

In the example program (Fig. 6) the code separation phase allocates only three processing elements while the physical target machine contains four processing elements. Therefore, the two independent memory operations (the vector loads for x and z) are split onto two processing elements. This leads to a schedule that utilizes the full capabilities of the target architecture. The modified schedule is show in Fig. 8.

6.5. Loop Translation

Of particular importance is the optimization of inner loops. Two optimization techniques are applied to loops in this compiler—*branch reduction* and *induction variable calculation.* These two optimizations attempt to eliminate instructions in inner loops, which can lead to significant performance improvements when these loop iterations account for a large portion of the execution time.

6.5.1. Branch Reduction

One ramification of having multiple processors cooperating on the execution of a task is that many more branch instructions are required in order to keep the instruction flows synchronized. This duplication of branch instructions in each stream can lead to a significant increase in the total number of instructions required to perform a task. The code

Table I. Code Expansion Due to Branch Replication[a]

bench	inst count	branch	expand	pred	loop
awk	12211251	2164041	18703374	17693006	17547358
compress	16672328	2651705	24627443	23234851	23216415
grep	13590761	3056768	22761065	21372373	20630417
nroff	12806704	2773837	21128215	19706583	19704251
troff	10302886	2037759	16416163	15224263	15203607
linpacks	73957172	4134742	86361398	85229646	74884782

[a] bench = name of the benchmark program examined
inst count = the total number of instructions executed
branch = the total number of branch instruction executed
expand = the total number of instructions executed/predicted after code has been separated (and balanced)
pred = the total # of instr executed after balancing if predicate transformations were applied
loop = the total # of instr executed after balancing and loop transformations

expansion due to branch duplication for a set of 6 benchmark programs is shown in Table I.

The MISC architecture provides two mechanisms to reduce the need for branch duplication: VLOOP/SLOOP instructions and predicated execution. The *vector loop* (VLOOP) instruction uses the vector register in conjunction with a *Delay Register* (DREG) to realize a very simple branch hiding (or zero cycle branch) instruction. It is used in cases where the number of times a basic block will execute is known at the initial entry into the loop (either as a constant or register variable).

The *sentinel loop* (SLOOP) instruction provides branch hiding capability to more generalized (*while*) loops. SLOOP operates in a manner similar to the VLOOP except that a sentinel comparison is made instead of a VREG calculation. When a SLOOP instruction reaches the issue stage of the pipeline, the *src3* operand is tested to see if the first iteration of the loop should be executed. The *src3* operand specifier is then saved to allow for additional sentinel tests to be performed during the last instruction issued each time through the loop body. Loop iteration continues until the sentinel marker is reached.

Conventional wisdom holds that instruction level implementations of higher level semantics seldom lead to performance improvements because of the complexity of implementation and the scarcity of application. While this may be true for a standard sequential processor, the vector and sentinel instructions defined in MISC can be implemented with minimal hardware modification to the issue logic and the existence of multiple instruction streams leads to a greater potential for application of these

instructions. Often the application of a complex construct (e.g., a *while* loop) is virtually impossible in a single instruction stream design because of the complexity of evaluating the test condition. The need both to evaluate a complex test condition and perform the control flow operation cannot be reduced to a single instruction. However, in a multiple instruction stream machine such as MISC, a single PE can evaluate the test condition and broadcast the boolean result to all PEs, increasing the number of simple boolean tests evaluated during the execution of these loop semantics. The reduction figures for the benchmark programs are shown in Table I.

If-conversion[28] also reduces the effects of branching by eliminating branching operations in favor of predicated execution. One problem with if-conversion is that while it may be useful to eliminate a branch to allow more efficient use of some operational units (e.g., load/store operations), the necessity of translating all instructions affected by the branch into a predicated form can lead to an overall decrease in execution performance. Converting these branches in MISC code can be performed for each PE in an independent manner. This allows PEs that can benefit from the removal of a branch operation to proceed with the hyperblock transformation while other PEs may more efficiently schedule instructions by retaining that branch. This also allows the application of VLOOP and SLOOP on a greater number of inner loops (containing small segments of conditionally executed code), since loop operations cannot contain branches, but can contain predicated instructions. The reduction figures for the benchmarks are also shown in Table I.

During branch reduction, a form of if-conversion is used to convert jumps around short segments of conditionally executed code into predicated instructions. After this has been accomplished, inner loops are again examined to determine whether the VLOOP or SLOOP operations can be inserted in place of the explicit branching operations. For loops containing no remaining branch operations (other than a single branch exit and the backward branch at the end of the loop), the VLOOP or SLOOP operator is inserted.

In the example of Fig. 8 it is a simple matter to determine that a VLOOP operation provides the required loop control operation. The effects of applying the loop translation along with induction variable calculation can be seen in Fig. 9.

6.5.2. Induction Variable Calculation

An induction variable is a variable whose value is consistently modified (incremented or decremented) by a constant value on each iteration of a loop. These variables are often used to determine the number of

```
[1]  [PE4]    t1 = 0                ; q=0
[2]
[3]  [PE1-4]  t3 = 1024             ; set register for test
[4]  [PE1]    t4 = LOC[_z]          ; t4 = base of array z
[5]  [PE2]    t5 = LOC[_x]          ; t5 = base of array x

[6]
[7]
[8]  [PE1-4]  VLOOP 1,0,t3          ; cee = 1, dee = 0
[9]  [PE1]    t7 = M[ t4+VREG ]     ; load t7=z[k]
[10] [PE2]    t8 = M[ t5+VREG ]     ; load t8=x[k]
[11] [PE3]    t9 = t7 * t8          ; (z[k] * x[k])
[12] [PE4]    t1 = t1 + t9          ; q = q + (z[k]*x[k])
[13]
[14] [PE1-4]  VLOOP_END
[15] [PE1-4]  L2:
```

Fig. 9. RTL for LLL3 after loop translation.

iterations. Furthermore, induction variables are often used to index array data items or manipulate memory pointers. Induction variables can be defined in terms of an *induction expression.* While a number of expressions are possible, a useful induction expression is:

$$IV_{(1)} = dee, \qquad IV_{(i+1)} = IV_{(i)} + cee \qquad \text{for } all \quad i > 1$$

where **i** is the iteration count (value 1 on the first iteration)

The detection of induction variables is a well understood problem. The algorithm used in this compiler is derived from Ref. 30 (Algorithm 10.9).

Once the control state of the machine has been modified to support loop operations, it is a simple modification to handle the calculation of induction variables used in the loop. The *src1* and *src2* fields of the loop instructions are free to contain the **cee** and **dee** values; **VREG** will maintain the induction value and *src3* will control loop termination as described above.

In the example in Fig. 8 both array index calculations can be performed by the hardware. This leads to the RTL description after loop translation show in Fig. 9.

6.6. Instruction Scheduling

A least-cost schedule is developed that attempts to schedule all instructions in the shortest time. List scheduling on MISC operates on each of the

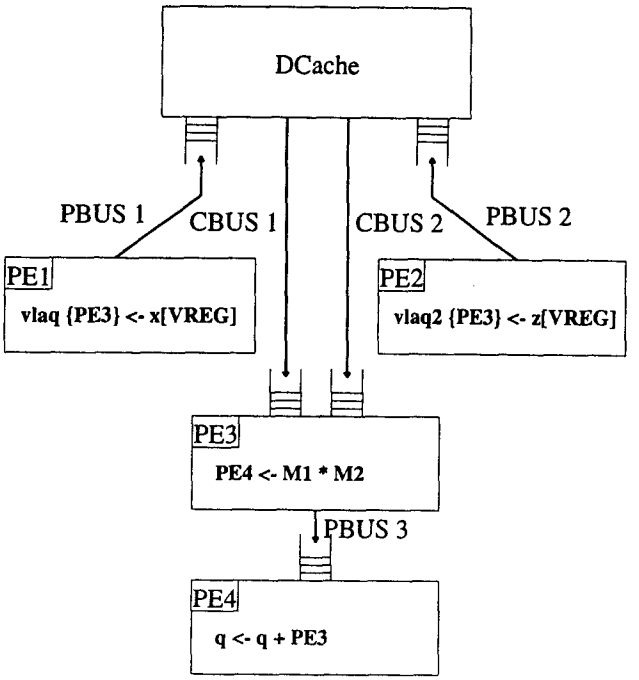

Fig. 10. Execution flow for LLL3.

processing elements individually, scheduling to avoid wasted cycles (due to latency). Simple list scheduling is complicated by the necessity to interpret queue register specifications in the RTL and avoid reordering queueing operations. Furthermore, the implementation of loop operations is left to this phase of the code generation. In the example the VLOOP operations for each of the processors can be replaced with the vector version since each loop consist of a single instruction and the default induction calculation is used (or no induction variable is referenced in PE3 and PE4). A flow graph representation of the MISC object code is shown in Fig. 10.

7. ANALYSIS

The Lawrence Livermore Loops were selected as the benchmarks, because they are amenable to hand-coding and are representative of a large class of scientific programs. The first 12 loops were compiled for both the MIPS and MISC architectures. The MIPS code was compiled using the cc compiler with optimization $-O2$, and the MISC code was generated using

the IAGO[31] compilation environment using the techniques discussed in Section 5 of this paper.

In order to compare the performance of MISC to the MIPS processor we examined the total number of cycles required to complete (**Cmplet**) a given loop and the number of cycles required before the execution of instructions after the loop can start (**Next**). The **Next** number is interesting because, since MISC PEs operate independently, one PE can complete its work on a given loop and begin execution of the code following the loop prior to the official loop termination.

As can be seen in Table II, for a majority of the loops we see a three- to four-fold decrease in the cycles required by the MISC machine over the MIPS processor in completion of a loop. This demonstrates that MISC is effectively extracting the parallelism available in the benchmark. In several of these benchmarks there is less of a performance increase (most notably in LLL6 and LLL11); this is due to a recurrence constraint found in the data manipulated by the loop. In these cases, adding more processors will not increase performance regardless of the approach used, since the parallelism is simply not available in the loop.

To provide a comparison with a similarly configured single instruction stream/multiple issue architecture, the loops were also hand compiled for a four-issue VLIW architecture based upon the version found in Ref. 32. This VLIW machine allows four instructions to be issued per clock cycle and places no limitations on the type of instructions that can be

Table II. Comparison of MIPS and MISC Cycle Counts for Livermore Loops

		MISC		Improvement	
Bnch	MIPS	Cmplet	Next	Cmplet	Next
LLL1	5611	1232	1205	4.55	4.65
LLL2	1112	256	201	4.34	5.53
LLL3	6664	2063	1025	3.23	6.50
LLL4	3011	753	385	3.99	7.82
LLL5	6979	1994	977	3.50	7.14
LLL6	7726	4982	0	1.55	∞
LLL7	4338	859	727	5.05	5.97
LLL8	3218	1476	586	2.18	5.49
LLL9	4081	813	609	5.02	6.70
LLL10	3107	1007	506	3.08	6.14
LLL11	3049	2003	0	1.52	∞
LLL12	3759	1013	1002	3.71	3.75

Table III. Comparison of Cycle Counts for MISC, VLIW, and Ideal Machines

Bnch	PE1	PE2	PE3	PE4	MISC	VLIW	Ideal
LLL1	1205	1215	1221	1232	1232	1236	1000
LLL2	201	201	211	256	256	228	200
LLL3	1025	1025	1035	2063	2063	2070	2048
LLL4	385	395	404	753	753	771	576
LLL5	997	999	1993	4982	4982	4984	4980
LLL6	0	997	1995	4982	4982	4984	4980
LLL7	727	846	736	859	859	863	780
LLL8	586	720	1240	1476	1476	—	950
LLL9	609	707	712	813	813	708	700
LLL10	506	506	1006	1007	1007	1014	750
LLL11	0	999	1000	2003	2003	2004	1998
LLL12	1002	1012	1013	1013	1013	1013	1000

issued. Furthermore, it assumes sufficient resources (e.g., register transfer bandwidth) to sustain a four-instructions-per-cycle execution rate. The "ideal" entry in Table III is derived by determining the total number of instructions required to complete the program loop, exclusive of branches (which can be removed by software or hardware techniques in any ideal machine). Barring any recurrence relations, the total instruction count is then divided by the issue bandwidth (four in this analysis).

The results in Table III show that the MISC approach and the VLIW model are capable of extracting about 80% to 99% of the instruction level parallelism available in these loops. However, as mentioned previously, the MISC PEs that finish prior to the overall completion of the loop are available to begin execution of the code following the loop exit. This demonstrates an important point in the performance capabilities of a multiple instruction stream processor; the MISC processor requires only those processing elements necessary to perform the task to be allocated to the loop while the unallocated processing elements can proceed into the following code blocks. In contrast, all functional units in the VLIW processor are locked into the loop (even if they have nothing to do).

A superscalar design might be capable of allocating processing resources across loops, but only with a sufficiently large instruction window and the ability to predict correctly many branches ahead in the instruction stream. The MISC approach of separating instruction streams alleviates these requirements.

To demonstrate the effects of this splitting of resources let us examine two of the loops in more detail. If we look at the execution of LLL6 we

Table IV. Comparison of Cycle Times for MISC and VLIW Executing Two Loops Sequentially

Loop	PE1	PE2	PE3	PE4	MISC	VLIW	Improvement	
6	0	997	1995	4982	4982	4984	∞	1.00
11	999	1000	2003	0	2003	2004	∞	1.00
6–11	999	1997	3998	4982	4982	6988	6.99	1.40

notice that only the final processing element is required to perform the majority of the loop calculation. This is due to a tight recurrence relation found in the loop equation. In the VLIW machine all functional units are forced to sit idle in the loop body until the machine (as a whole) completes calculation of the loop. In the MISC approach, the three processing elements not involved in the recurrence calculation are free to continue execution.

If we now assume that LLL11 follows the execution of LLL6, we can determine the different *stagger* rates on exit from LLL6 and reschedule LLL11 to take advantage of the free processing elements. Table IV shows the result of this rescheduling (done at compile time) and compares it to the VLIW architecture. As seen in Table IV, the ability to overlap execution of the loops allows the MISC processor to perform both loops in the time required by the VLIW architecture to perform the first alone.

We expect this final result to be demonstrative of the advantage that the multiple instruction stream attains across basic blocks. Little dataflow analysis is required to achieve this capability.

8. CONCLUSIONS/FUTURE WORK

In this paper we have examined the feasibility of using a MIMD approach to extracting instruction level parallelism. Current MIMD architectures suffer from various deficiencies which prevent their direct application to instruction level parallel tasks. A new architecture has been proposed which alleviates these deficiencies and provides both reduced hardware complexity and simplified software scheduling compared to conventional (single instruction stream) approaches. When supported with a new code scheduling method, this architecture provides performance equivalent to the most powerful VLIW and Superscalar architectures proposed, while maintaining simple hardware and software schemes.

The ability to specify more information in the object language of the MISC machine (by explicitly defining separate instruction streams) simplifies the hardware mechanisms required to support out-of-order execution.

This provides a powerful combination of software based control flow optimizations with the dynamic features found in out-of-order execution models in a more cooperative way than that found in existing VLIW and Superscalar designs, by increasing the information content between the two.

We believe these results will prove to be even more significant in non-vectorizable code, because of the latency hiding effects inherent in the decoupled model. We are currently refining the compilation environment, in order to examine a much wider range of benchmark programs.

ACKNOWLEDGMENTS

This work was suported by the National Science Foundation under Grant MIP-9257259, and by a generous donation from SUN Microsystems.

REFERENCES

1. N. P. Jouppi and D. W. Wall, Available instruction-level parallelism for superscalar and superpipelined machines, *Proc. of the Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, Mass, pp. 272–282 (April 1989).
2. M. E. Benitez and J. W. Davidson, Code generation for streaming: an access/execute mechanism, *Proc. of the Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, pp. 132–141 (April 1991).
3. T. Austin and G. Sohi, Dynamic dependency analysis of ordinary programs, *Proc. of the 19th Ann. Symp. on Computer Architecture* **20**(2):342–351 (May 1992).
4. M. Butler, T. Yeh, and Y. Patt, Single instruction stream parallelism is greater than two, *Proc. of the Eighteenth Ann. Int. Symp. on Computer Architecture*, Toronto, Canada, pp. 276–286 (May 1991).
5. J. E. Smith, Decoupled access/execute computer architectures, *Trans. on Computer Systems* **2**(4):289–308 (November 1984).
6. J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon, The ZS-1 central processor, *Proc. of the Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, pp. 199–204 (October 1987).
7. W. Wulf, Evaluation of the WM architecture, *Proc. of the 19th Ann. Symp. on Computer Architecture* **20**(2):382–390 (May 1992).
8. R. L. Sites, Alpha AXP architecture, *Comm. of the ACM* **36**(2):33–44 (February 1993).
9. C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. P. Shen, Instruction level profiling and evaluation of the IBM RS/6000, *Proc. of the 19th Ann. Symp. on Computer Architecture* **20**(2):180–189 (May 1992).
10. J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, PIPE: a VLSI decoupled architecture, *Proc. of the Twelveth Ann. Int. Symp. on Computer Architecture*, pp. 20–27 (June 1985).
11. G. L. Craig, J. R. Goodman, R. H. Katz, A. R. Pleszjun, K. Ramachandran, J. Sayah, and J. E. Smith, PIPE: a high performance VLSI processor implementation, *Journal of VLSI and Computer Systems*, Vol. 2 (1987).

12. D. Wall, Limits of instruction-level parallelism, *Proc. of the Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 176–189 (April 1991).
13. M. Farrens, G. Tyson, and A. Pleszkun, A study of single-chip processor/cache organizations for large numbers of transistors, *Proc. of the 21th Ann. Int. Symp. on Computer Architecture*, Chicago, Illinois (April 1994).
14. M. S. Lam, Software pipelining: an effective scheduling technique for VLIW machines, *Proc. of the ACM SIGPLAN Notices 1988 Conf. on Programming Languages and Implementations*, pp. 318–328 (June 1988).
15. S. Weiss and J. E. Smith, A study of scalar compaction techniques for pipelined supercomputers, *Proc. of the Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, pp. 105–109 (October 1987).
16. G. Tyson, M. Farrens, and A. Pleszkun, MISC: a multiple instruction stream computer, *Proc. of the 25th Ann. Int. Symp. on Computer Architecture*, Portland, Oregon, pp. 193–196 (December 1992).
17. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, Conversion of control dependencies to data dependencies, *Proc. of the 10th ACM Symp. on Principles of Programming Languages*, pp. 177–189 (January 1983).
18. M. Farrens and A. Pleszkun, Overview of the PIPE processor implementation, *Proc. of the 24th Ann. Hawaii Int. Conf. on System Sciences*, Kapaa, Kauai, pp. 433–443 (January 1991).
19. H. C. Young, *Evaluation of a Decoupled Computer Architecture and the Design of a Vector Extension*, Ph.D. Thesis, University of Wisconsin-Madison (July 1985).
20. M. E. Benitez and J. W. Davidson, Code generation for streaming: an access/execute mechanism, *Proc. of the Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 132–141 (April 1991).
21. R. Gupta, A fine-grained MIMD architecture based upon register channels, *Proc. of the 23rd Ann. Symp. and Workshop on Microprogramming and Microarchitectures*, Orlando, Florida, pp. 54–64 (November 1990).
22. F. Ferrante, K. Ottenstein, and J. Warren, The program dependence graph and its use in optimization, *ACM Trans. on Programming Languages and Systems*, pp. 319–349 (July 1987).
23. R. A. Iannucci, Toward a dataflow/von Neumann hybrid architecture, *Proc. of the 15th Ann. Symp. on Computer Architecture*, pp. 131–140 (1988).
24. R. Gupta, The fuzzy barrier: a mechanism for high-speed synchronization of processors, *Proc. of the Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 54–64 (1989).
25. A. Wolfe and J. P. Shen, A variable instruction stream extension to the VLIW architecture, *Proc. of the Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 2–14 (April 1991).
26. M. Danelutto and M. Vanneschi, VLIW-in-the-large: a model for fine grain parallelism exploitation on distributed memory multiprocessors, *Proc. of the 23rd Ann. Symp. and Workshop on Microprogramming and Microarchitectures*, Orlando, Florida, pp. 7–16 (November 1990).
27. S. W. Keckler and W. J. Dally, Processor coupling: integrating compile time and runtime scheduling for parallelism, *Proc. of the 19th Ann. Int. Symp. on Computer Architecture*, Queensland, Australia, pp. 202–213 (May 1992).
28. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, Effective compiler support for predicated execution using the hyperblock, *Proc. of the 25th Ann. Int. Symp. on Microarchitecture*, Portland, Oregon, pp. 45–54 (December 1992).

29. J. E. Smith, Decoupled access/execute computer architectures, *Proc. of the Ninth Ann. Int. Symp. on Computer Architecture*, Austin, Texas, pp. 112–119 (April 1982).
30. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley Publishing, pp. 644.
31. G. S. Tyson, R. Shaw, and M. Farrens, An interactive compiler development system, *Tcl/Tk Workshop* (June 1993).
32. B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, Code generation schema for modulo scheduled loops, *Proc. of the 25th Ann. Int. Symp. on Microarchitecture*, Portland, Oregon, pp. 158–169 (December 1992).