Instruction-Level Parallel Processing

Joseph A. Fisher and B. Ramakrishna Rau

The performance of microprocessors has increased steadily over the past 20 years at a rate of about 50% per year. This is the cumulative result of architectural improvements as well as increases in circuit speed. Moreover, this improvement has been obtained in a transparent fashion, that is, without requiring programmers to rethink their algorithms and programs, thereby enabling the tremendous proliferation of computers that we see today. To continue this performance growth, microprocessor designers have incorporated instruction-level parallelism (ILP) into new designs. ILP utilizes the parallel execution of the lowest level computer operations—adds, multiplies, loads, and so on—to increase performance transparently. The use of ILP promises to make possible, within the next few years, microprocessors whose performance is many times that of a CRAY-1S. This article provides an overview of ILP, with an emphasis on ILP architectures—superscalar, VLIW, and dataflow processors—and the compiler techniques necessary to make ILP work well.

Much of the dramatic increase in the performance of microprocessors over the past 20 years has come from continuing improvements in processor architecture. Indeed, while overall performance has grown steadily at the compounded rate of about 50% per year, raw circuit speed has accounted for less than half of that annual increase (*1*). The rest has been due to such factors as an increase in word size from 8 bits to 32 bits, a move from complex instruction sets (CISC) to reduced instruction sets (RISC), the inclusion of caches (*2*) as well as floating point hardware on-chip, and, most recently, the incorporation of features, such as pipelining (*3*) that previously were only used in mainframes and supercomputers. These features would not be possible without an increase in chip density: the number of transistors on a microprocessor chip has increased at about 40% per year, and this trend shows no sign of abating.

An important feature of these architectural improvements is that, like circuit speed improvements, they are largely transparent to users, who do not have to change their way of working to derive performance benefits. Continuing this remarkable trend has presented a challenge to processor designers, and their response has spawned important new technologies, chief among them instruction-level parallelism (ILP). ILP has as its objective the execution in parallel of the lowest level machine operations, such as memory loads and stores, integer additions and floating point multiplications. These are the normal RISC-style operations that are executed when a program runs, but by using ILP multiple operations are executed in parallel, even though the system is handed a single program written with a sequential processor in mind.

The authors are with the Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94303.

Traditional parallel processing derives its speed improvements in a fundamentally different way from ILP. When parallel processing is used, large sections of code are run in parallel on independent processors. There is hardly a development more exciting than parallel processing: it has the potential to change fundamentally what can be done with a computer. Some problems that would be unthinkable on an ordinary computer can be solved using massively parallel processing. But, for algorithmic reasons, many applications probably never will be subject to speed improvement through parallel processing. And when it is possible, major investments are often required—usually algorithms and code must be thoroughly rethought and reworked to be effective. There are billions of dollars worth of "dusty deck" uniprocessor-oriented software, ranging from engineering and scientific simulations to commercial processing, running on the uniprocessors that constitute virtually all of the computers in use today. These programs will not soon be running on parallel processors; those that ultimately do are likely to be running on systems in which each parallel node is itself an ILP processor. Thus traditional parallel processing and instruction-level parallel processing complement each other—only infrequently are they viewed as alternatives in seeking the same end.

The challenge of ILP. The use of ILP promises to continue the remarkable 20-year record of microprocessor performance improvement that has sustained the computer industry and made possible the birth of new scientific disciplines such as computational physics, computational chemistry and computational cosmology. While the potential of ILP is exciting, many important questions face the designer of an ILP system. In this article, with the help of a hypothetical ILP processor running a sample program, we will examine these issues and will describe increasingly sophisticated strategies for exposing and utilizing instruction-level parallelism.

Parallel Instruction Execution

ILP hardware. Consider a model of hardware consisting of four functional units and a branch unit connected to a common register file (Fig. 1A). We will use this model in all three of the hypothetical processors that we will discuss below. This processor is capable of starting two integer operations, two memory operations, and a branch or compare operation every cycle (though on a real machine one might also expect functional units for floating point and address computation). In addition to the input operands that one would normally expect for an operation, each operation has an additional single-bit input called the predicate. If the predicate input is true, the operation executes normally. If false, the operation does nothing and the result register is not updated.

All of these operations get their input operands, including the predicate, from registers. With this hardware, the fact that five predicated operations can be started every cycle implies that up to fifteen operands need to be read from the register file and up to five results need to be stored into the register file every cycle. A register file of this kind, which is capable of supporting multiple reads and writes each cycle, is referred to as a multiported register file. (A real machine with this many functional units would be likely instead to have several multiported register banks.)

Certain operations, such as integer addition and subtraction, complete in a single cycle (Fig. 1B). Other operations, such as integer multiplication and memory loads, take multiple cycles to complete. Nevertheless, much as in a manufacturing assembly line, this processor can start a new operation on each functional unit before the previous ones have completed, employing a technique called pipelining. With pipelining it is possible to have multiple operations, in various stages of completion, simultaneously executing on a single functional unit. In fact, if a program were causing the longest latency operation to be issued on every unit in each cycle, this machine would be capable of starting five operations per cycle and of having 11 operations "in flight" at all times.

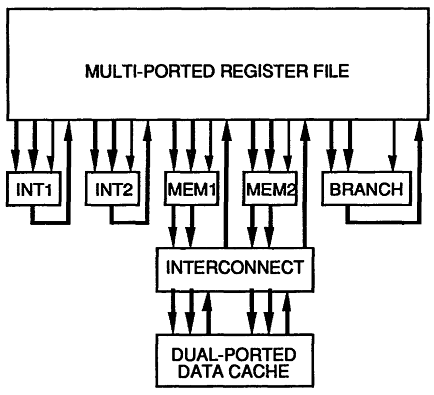
There is thus a fair amount of parallelism available even in this simple processor. The challenge is to make good use of it. The first question that comes to mind is whether enough ILP exists in programs to make this possible. Then, if this is so, what must the compiler and hardware do to successfully exploit it?

A sample program fragment. We shall use a simple program fragment to illustrate how ILP can be found, enhanced and exploited (Fig. 2A). A basic block is the largest contiguous set of statements with no branches into or out of the middle of that set. The program fragment includes two loops. The inner loop is the one around basic block D. The outer loop is around the entire code fragment. On each iteration of the outer loop, the program may either execute the inner loop or it may execute basic block B. This behavior is shown pictorially in a control flow graph (Fig. 2B).

Many techniques for optimizing and parallelizing programs are dependent upon knowing which portions of the code are executed more frequently than others. A useful statistic is the frequency of transitioning between each pair of basic blocks, from which one can also calculate the execution frequency for each basic block. In practice, these statistics are gathered by a profiling tool while the program runs and are saved for later use.

A serial execution of a program is one in which each statement is completed before the next one is begun. The time a program takes to execute serially can be calculated by adding the latencies of all operations in a basic block, multiplying that by the number of times the block is executed, and summing over all blocks. Serial execution of our example results in 18,320 operations being executed in 29,680 cycles for an average execution rate of 0.62 operations per cycle (Fig. 3, A and B).

Basic block ILP. A program can run faster if we use the instruction-level parallelism that exists between instructions within a basic



A

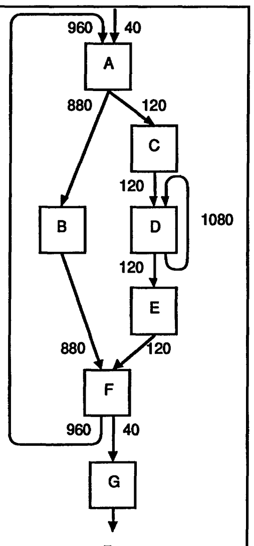
Operation	Functional Unit Used	Execution Latency
Integer Add	Integer Unit	1
Integer Multiply	Integer Unit	3
Load	Memory Port	2
Store	Memory Port	1
Compare	Branch Unit	1
Compare and Branch	Branch Unit	1
Branch Unconditional	Branch Unit	1

B

Fig. 1. Execution hardware of a sample processor for executing programs with instruction-level parallelism. (**A**) Datapaths of the processor. Only those functional units necessary for the program example are shown. These include two identical integer units and two identical memory ports. INT1 and INT2 perform integer addition, subtraction and multiplication. MEM1 and MEM2 represent the hardware that prepares and presents load and store operations to the dual-ported data cache. BRANCH performs conditional and unconditional branches. It is able to read a pair of registers, compare them and branch on the Boolean result. It is also able to store the boolean result of a comparison into the register file for use as a predicate. Every operation has a third input, a boolean value, shown as the thin arrow into each functional unit. If this boolean value is true, the operation executes normally. If not, it is not executed at all. (**B**) Latencies of the various operations. Only those operations necessary for the program example are listed. Integer addition, memory stores and branches complete in a single cycle. Integer multiplication and memory loads take three and two cycles, respectively, to complete, but they are pipelined; a new operation can be started every cycle on a functional unit even if the previous ones have not yet completed. INT1 and INT2, both of which perform operations having unequal latencies, have the restriction that two operations on the same functional unit may not finish at the same time since there is only a single result bus per functional unit to communicate the result back to the register file. MEM1 and MEM2 are not subject to this restriction since store operations do not return a value to the register file.

Operation Label	Basic Block Label	Statement
A1	A:	T1 = LOAD X(I-1)
A2		T2 = LOAD X(I)
A3		T3 = 2 * T1
A4		T4 = T3 - T2
A5		IF T4 ≤ 0 GOTO B
C1	C:	U = 0
C2		J = 1
D1	D:	T6 = LOAD X(I-J)
D2		T4 = T4 + 5
D3		T7 = T4 * T6
D4		U = U + T7
D5		J = J + 1
D6		IF J ≤ 10 GOTO D
E1	E:	GOTO F
B1	B:	T5 = T2 - T1
B2		U = 2 * T5
F1	F:	T8 = 3 * V
F2		V = T8 + U
F3		I = I + 1
F4		IF I ≤ 25 GOTO A
	G:	

A



B

Fig. 2. (**A**) A fragment of a sample program. Each statement corresponds to a single operation on the example processor (Fig. 1). (Address computation is ignored for simplicity.) The first column is the label for the corresponding statement in the third column. The second column lists the labels for the basic blocks. All statements starting with the one on the same line as a basic block label up to the statement just prior to the next label are in the same basic block. (**B**) The control flow graph for the sample code. Each block corresponds to a single basic block. The arcs indicate the order in which flow of control can move from one basic block to another. The existence of two arcs out of a block is the result of having a conditional branch in that basic block. The number on an arc indicates the frequency of that arc, that is, the number of times that that arc was found to have been traversed during an execution of the program. The frequency of a certain basic block is the sum of the frequencies of all incoming (or outgoing) arcs.

block. Instead of starting an instruction only after the previous instruction has finished, one could start it as soon as the instructions that generate its input operands have finished and sufficient resources, such as functional units and registers, exist to execute it. This execution strategy yields a record of execution for most basic blocks that is slightly more parallel than before. For instance, block D executes in six cycles instead of in seven cycles (Fig. 4). This strategy results in a slight reduction in the execution times for basic blocks A, C, D and F, dropping the total execution to 21,960 cycles for an average of 0.83 operations per cycle (Fig. 3C) and with as many as three operations being issued at one time.

Parallelism within each basic block is limited by the data dependences between instructions, which results in serial execution. This is compounded by the execution latency of operations, which lengthens the time to execute the serial chain of instructions. For instance, in basic block D, statement D3 uses T6 which is the result of D1. So, D3 is dependent on D1. Likewise, D4 is dependent on D3 through its use of T7. On the other hand, D5 is not dependent on D1, D2, D3 or D4; independences like this can lead to a small amount of parallelism within a single basic block. The dependences between operations are often represented as a graph (Fig. 5) with the nodes representing the statements and the arcs representing the dependences between statements. The longest path through a dependence graph is called the critical path. Consider the dependence graph for basic block D. Taking into account the latencies of the operations, the critical path for D, through D1, D3 and D4 (which are thus called critical path operations), is six cycles long. Six cycles is thus the shortest time in which basic block D can be executed.

TIME	INT1	INT2	MEM1	MEM2	BRANCH
1	D2	D5	D1	-	-
2	-	-	-	-	-
3	D3	-	-	-	-
4	-	-	-	-	-
5	-	-	-	-	-
6	D4	-	-	-	D6

Fig. 4. Record of execution for basic block D when exploiting the ILP within that block.

Global ILP. Exploiting the parallelism within individual basic blocks led only to a 30% speedup (Fig. 3C); we could expect more if the computations in separate basic blocks could also be executed in parallel with one another. The problem is that basic blocks are separated by branches, most often conditional branches. Frequently, the branch condition can only be calculated at the end of the basic block and only then is it possible to determine the identity of the basic block that is to be executed next. This would appear to frustrate any attempts to execute basic blocks in parallel.

One approach is to recognize that although we may not yet know whether the next basic block is to be executed, we could be certain that a subsequent one must be. In our example, while executing basic block A, even though branch A5 may not have been executed, it is certain that basic block F must be executed since all paths from A lead to it. In addition, we could test the branch condition in F4 immediately to determine whether another iteration is to be executed. If this turns out to be the case, we now know that basic blocks A and F from the next iteration, too, must definitely be executed. This already allows us to execute the operations from four basic blocks in parallel (assuming that they are not dependent on one another) without yet knowing which way the branch A5 in the first iteration is going. This type of parallelism is termed control parallelism.

At times, we might find that there is inadequate control parallelism to fully utilize the functional units. The second approach for finding more parallelism, termed speculative execution, relies on the fact that there is little to be lost in using otherwise idle resources to execute operations whose results we might, possibly, end up using. Although we may not yet be positive that the program will actually flow to certain subsequent blocks, we can still choose to speculatively execute operations from those blocks in parallel. Operations executed speculatively must be capable of being ignored if the flow does not later mandate their execution (and in real machines one must solve the difficult problem of treating error conditions raised by speculative operations). Thus, for example, stores to memory are rarely candidates for speculative execution.

Without knowing which way any branch actually does go, the transition frequencies for our example (Fig. 2B) show that the path from basic block A to B to F and back to A will be repeatedly executed multiple times. Thus a reasonable strategy is to use every cycle that would otherwise be wasted to execute instructions speculatively down this expected path. If we execute operations on this path as soon as their input operands are available, the resulting record of execution displays considerable parallelism. When this is

Basic Block	Number of Operations	Frequency	Operations Executed
A	5	1000	5000
C	2	120	240
D	6	1200	7200
E	1	120	120
B	2	880	1760
F	4	1000	4000
Total			18320

A

Basic Block	Execution Time	Frequency	Weighted Time
A	9	1000	9000
C	2	120	240
D	9	1200	10800
E	1	120	120
B	4	880	3520
F	6	1000	6000
Total			29680
Average Operations Per Cycle			0.62

B

Basic Block	Execution Time	Frequency	Weighted Time
A	7	1000	7000
C	1	120	120
D	6	1200	7200
E	1	120	120
B	4	880	3520
F	4	1000	4000
Total			21960
Average Operations Per Cycle			0.83

C

Basic Block	Execution Time	Frequency	Weighted Time
A	7	1000	7000
C	5	120	600
D	2	960	1920
E	4	120	480
B	4	880	3520
F	4	1000	4000
Total			17520
Average Operations Per Cycle			1.05

D

Basic Block	Execution Time	Frequency	Weighted Time
1A	8	500	3500
1C	5	60	300
1D	2	480	960
1E	4	60	240
1F	1	500	500
2A	1	500	500
2C	5	60	300
2D	2	480	960
2E	4	60	240
2F	1	500	1500
Total			8500
Average Operations Per Cycle			2.16

E

Basic Block	Execution Time	Frequency	Weighted Time
H	8	1	8
C	5	120	600
D	2	960	1920
E	4	120	480
ABF	4	998	3992
G	8	1	8
Total			7008
Average Operations Per Cycle			2.61

F

Fig. 3. (**A**) The number of operations executed in the serial execution of the program. (**B–F**) Execution time (per basic block and total) for various schemes that are discussed over the course of this article. The time to execute the sample code can be calculated by first computing, for each basic block, the execution time for a single visit to that basic block, next, weighting the execution time for each basic block by its execution frequency and, finally, adding up the weighted execution times. In the case of serial execution, the execution time for each basic block is computed by adding up the products of the number of times each operation is executed per visit to that block and the latency of that operation: (B) serial execution, (C) execution exploiting only the ILP within individual basic blocks, (D) execution exploiting only the ILP within individual basic blocks except for loop D which is software pipelined, (E) outer loop unrolled once, basic blocks 1A, 1B, 1F, 2A, 2B, 2F trace scheduled, and both copies of loop D software pipelined, (F) outer loop trace through A, B, F software pipelined and loop D software pipelined.

```
            Basic
Operation   Block
  Label     Label    Statement

   D1         D:     T6 = LOAD X(I-J)
   D2                T4 = T4 +5
   D3                T7 = T4 * T6
   D4                U = U + T7
   D5                J = J + 1
   D6                IF J ≤ 10 GOTO D

                 A
```

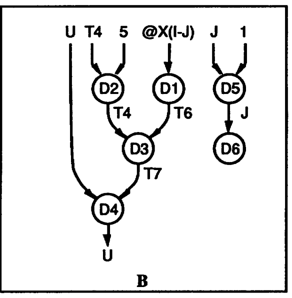


Fig. 5. (**A**) The code for basic block D. Parallelism within a basic block is limited by the data dependences between instructions and by the execution latency of operations. For instance, operation D3 uses T6 which is the result of D1. So, D3 is dependent on D1 and cannot begin until D1 has finished. Likewise, D4 is dependent on D3 through its use of T7. On the other hand, D5 is not dependent on D1, D2, D3 or D4. Such independence leads to a small amount of parallelism within individual basic blocks. (**B**) The dependence graph for basic block D. The ILP within the basic block is represented by a graph with the nodes representing the operations and the arcs representing the dependences between operations. The input to D1, labeled "@X(I-J)," is the address of the array element X(I-J).

compared to the records of execution for the individual basic blocks A, B, and F with no speculative execution, the improvement (assuming that branches always go in the expected direction) is almost a factor of 4. In this case three or four operations are consistently started per cycle. This is by no means an atypical situation.

For the past 20 years, researchers have been gathering experimental data on the amount of available instruction-level parallelism for various workloads and under a variety of assumptions (*4–7*). Although the results have varied with the experimental conditions, a consistent pattern has emerged: potential ILP, when limited to basic blocks, offers speedups of between a factor of 2 and 3.5. When techniques that move operations from block to block are used, the potential jumps to a factor of between 5 and 20, with even more speedup available in various applications such as array-oriented programs. By using estimated probabilities of branch direction, a system can selectively execute those operations which are most likely to be profitable. Experiments (*8, 9*) and experience have indicated that branch directions are sufficiently predictable to make speculative execution practical.

ILP Architectures

So far we have only demonstrated the end result of instruction-level parallelism, the record of execution, without having said much about how this is effected. How exactly are the necessary decisions made as to when an operation should be executed and whether an operation should be speculatively executed? The alternatives can be broken down depending on the extent to which these decisions are made by the compiler rather than by the hardware and on the manner in which information regarding parallelism is communicated by the compiler to the hardware via the program.

A computer architecture is a contract between the class of programs that are written for the architecture and the set of processor implementations of that architecture. Usually this contract is concerned with the instruction format and the interpretation of the bits that constitute an instruction, but in the case of ILP architectures it extends to information embedded in the program pertaining to the available parallelism. With this in mind, ILP architectures can be classified as follows:

- *Sequential architectures*: Architectures for which the program is not expected to convey any explicit information regarding parallelism. Superscalar processors (*10–14*) are representative of ILP processor implementations for sequential architectures.
- *Dependence architectures*: Architectures for which the program explicitly indicates the dependences that exist between operations. Dataflow processors (*15–17*) are representative of this class.
- *Independence architectures*: Architectures for which the program provides information as to which operations are independent of one another. Very Long Instruction Word (*18–21*) processors are an example of the class of independence architectures.

These architectural alternatives are elaborated upon below and the three types of processors are discussed in the next section (*22*).

Sequential architectures. Since the compiler neither identifies parallelism nor makes scheduling decisions, the program contains no explicit information regarding the dependences between the instructions. If instruction-level parallelism is to be employed, the dependences that exist must be determined by the hardware, which must then make the scheduling decisions as well.

Dependence architectures. The compiler, or the writer of a program, identifies the parallelism in the program and communicates it to the hardware by specifying the dependences between operations. This is typically done by specifying, for each operation, the list of other operations that are dependent upon it. The hardware must make scheduling decisions at run-time which are consistent with the dependence constraints.

Independence architectures. The compiler identifies the parallelism in the program and communicates it to the hardware by specifying which operations are independent of one another. This information is of direct value to the hardware, since it knows with no further checking which operations it can execute in the same cycle. Unfortunately, for any given operation, the number of independent operations is far greater than the number of dependent ones and, so, it is impractical to specify all independences. Instead, for each operation, independences with only a subset of all independent operations (those operations that the compiler thinks are the best candidates to execute concurrently) are specified.

By listing operations that could be executed simultaneously, code for an independence architecture may be very close to the record of execution produced by an implementation of that architecture. If the architecture additionally requires that programs specify where (on which functional unit) and when (in which cycle) the operations are executed, then the hardware makes no run-time decisions and the code is virtually identical to the desired record of execution. The Very Long Instruction Word (VLIW) processors that have been built to date are of this type and represent the predominant examples of machines with independence architectures. A particular processor implementation of an independence architecture of this type could choose to disregard the scheduling decisions embedded in the program, making them at run-time instead. In doing so, the processor would still benefit from the independence information but would have to perform all of the scheduling tasks of a superscalar processor. Furthermore, when attempting to execute concurrently two operations that the program did not specify as being independent of each other, it must determine independence, just as a superscalar processor must.

ILP Processor Implementations

Superscalar processors. The goal of a superscalar processor is to execute many operations in parallel even though the hardware is handed a sequential program. But a sequential program is constructed with the assumption only that it will execute correctly when each operation waits for the previous one to finish, and that is the only order that the architecture guarantees to be correct. The first task, then, for a superscalar processor is to understand, for each instruction, which other instructions it actually is dependent upon. With every instruction that a superscalar processor issues, it must check whether the instruction's operands (registers or memory locations that the instruction uses or modifies) interfere with the operands of any other instruction that is either:

- Already in execution, or
- Has been issued but is waiting for the completion of interfering instructions that would have been executed earlier in a sequential execution of the program, or
- Is being issued concurrently but would have been executed earlier in a sequential execution of the program.

If none of these conditions is true, the instruction in question is independent of all these other instructions and it can be allowed to begin executing immediately. If not, it must be delayed until the instructions on which it is dependent have completed execution. In the meantime, the processor may begin execution of later instructions which prove to be independent of the ones that are being delayed. In addition, the superscalar processor must decide exactly when and on which available functional unit to execute the instruction. The IBM System/360 Model 91, built in the early 1960s, utilized a method called Tomasulo's Algorithm (*23*) to carry out these functions.

Note that a superscalar processor need not issue multiple operations per cycle in order to achieve a certain level of performance. For instance, in the case of our sample processor (Fig. 1), the same performance could be achieved by pipelining the functional units and instruction issue hardware five times as deeply, speeding up the clock rate by a factor of five but issuing only one instruction per cycle. This strategy has been termed superpipelining (*24*). Superpipelining may result in some parts of the processor (such as the instruction unit and communications busses) being less expensive and better utilized and other parts (such as the execution hardware) being more costly and less well used.

Dataflow processors. The objective of a dataflow processor is to execute an instruction at the earliest possible time subject only to the availability of the input operands and a functional unit upon which to execute the instruction. Unlike a superscalar processor, it counts on the program to provide information about the dependences between instructions. This is accomplished by including in each instruction a list of successor instructions. (An instruction is a successor of another instruction if it uses as one of its input operands the result of that other instruction.) Each time an instruction completes, it creates a copy of its result for each of its successor instructions. As soon as all of the input operands of an instruction are available, the hardware fetches the instruction, which specifies the operation to be performed and the list of successor instructions. The instruction is then executed as soon as a functional unit of the requisite type is available. This property, whereby the availability of the data triggers the fetching and execution of an instruction, is what gives rise to the name of this type of processor. Because of this property, it is redundant for the instruction to specify its input operands. Rather, the input operands specify the instruction! If there is always at least one instruction ready to execute on every functional unit, the dataflow processor achieves peak performance.

As we have seen, computation within a basic block typically does not provide adequate levels of parallelism. Thus superscalar and VLIW processors use control parallelism and speculative execution to keep the hardware fully utilized. Dataflow processors have traditionally counted on control parallelism alone to fully utilize the functional units. A dataflow processor is more successful than the others at looking far down the execution path to find abundant control parallelism. When successful, this is a better strategy than speculative execution because every instruction executed is a useful one and the processor does not have to deal with error conditions raised by speculative operations.

Very Long Instruction Word processors. In order to execute operations in parallel, the system must determine that the operations are independent of one another. Superscalar processors and dataflow processors represent two ways of deriving this information at run-time. In the case of the dataflow processor, the explicitly provided dependence information is used to determine when an instruction may be executed so that it is independent of all other concurrently executing instructions. The superscalar processor must do the same but, because programs for it lack any explicit information, it must also first determine the dependences between instructions. In contrast, the program for a VLIW processor specifies exactly which functional unit each operation should be executed on and exactly when each operation should be issued so as to be independent of all operations that are being issued at the same time as well as of those that are in execution.

With the VLIW processor, it is important to distinguish between an instruction and an operation. An operation is a unit of computation, such as an addition, memory load, or branch, which would be referred to as an instruction in the context of a sequential architecture. A VLIW instruction is the set of operations that are intended to be issued simultaneously. It is the task of the compiler to decide which operations should go into each instruction. This process is termed scheduling. Conceptually, the compiler schedules a program by emulating at compile-time what a dataflow processor, with the same execution hardware, would do at run-time. All operations that are supposed to begin at the same time are packaged into a single VLIW instruction. The order of the operations within the instruction specifies the functional unit on which each operation is to execute. A VLIW program is a transliteration of a desired record of execution which is feasible in the context of the given execution hardware.

The compiler for a VLIW machine specifies that an operation be executed speculatively merely by performing speculative code motion, that is, scheduling an operation before the branch that determines that it should, in fact, be executed. At run-time, the VLIW processor blindly executes this operation exactly as specified by the program just as it would for a non-speculative operation. Speculative execution is virtually transparent to the VLIW processor and requires little additional hardware. When the compiler decides to schedule an operation for speculative execution, it can arrange to leave behind enough of the state of the computation to assure correct results when the flow of the program requires that the operation be ignored. The hardware required for the support of speculative code motion consists of having some extra registers, of fetching some extra instructions, and of suppressing the generation of spurious error conditions. The VLIW compiler must perform many of the same functions that a superscalar processor performs at run-time to support speculative execution.

Other types of independence architecture processors have been built or proposed. The superpipelined machine, described above, issues only one operation per cycle. But if there is no superscalar hardware devoted to preserving the correct execution order of operations, the compiler will have to schedule them with full knowledge of dependencies and latencies. From the compiler's point of view, these machines are virtually the same as VLIWs, though the hardware design of such a processor offers some tradeoffs with respect to VLIWs. Another proposed independence architecture, dubbed Horizon (*25*), encodes an integer H into each operation. The architecture guarantees that all of the past H operations in the instruction stream are independent of the current operation. All the hardware has to do to release an operation, then, is assure itself that no operation older than the Hth previous operation is in flight or pending. The hardware does all of its own scheduling, unlike VLIWs and deeply pipelined machines which rely on the compiler, but the hardware is relieved of the task of determining data dependence.

ILP Compiler Techniques

Regardless of whether an ILP processor makes the final scheduling decisions at run-time or compile-time, a compiler must generate code in which operations are rearranged for better performance. In the case of VLIW processors, the operations are arranged precisely in their execution order. In the case of superscalar processors, the hardware can only consider some number of nearby operations (called the instruction window) at one time. Compilers make use of their understanding of the processor's hardware scheduling algorithms to produce a rearranged program loosely tailored for that hardware, giving the processor the greatest possible opportunity to find parallelism.

Various compiler techniques have been developed to schedule operations exactly or approximately. The simplest of these address a single basic block, and are often referred to as local scheduling. For example, a compiler might schedule each operation by means of a procedure similar to the following:

1) Use heuristics to pick one of the unscheduled operations whose predecessors have already been scheduled. Simple heuristics usually suffice.

2) Calculate the completion time for each of its predecessor operations by adding their execution latency to the time at which they are scheduled to begin execution. Take the largest of these completion times. This is the earliest time at which the operation can be scheduled to begin execution.

3) Determine the earliest time thereafter when an appropriate functional unit is available to perform the operation, that is, no previously scheduled operation has been assigned to that functional unit at the same time. Schedule the operation at this time on this functional unit.

4) If there are any remaining unscheduled nodes, repeat the procedure from step 1.

This procedure could be employed, for example, to the computation in basic block D, yielding a schedule for a VLIW (Fig. 4). Note the similarity between the above procedure and the sequence of steps by which the dataflow processor would execute the same computation.

Global scheduling techniques. We have seen that most of the available parallelism is found beyond the boundaries of basic blocks. Until a decade ago, compiler techniques which scheduled ILP code by moving it from block to block (called global scheduling) were virtually nonexistent. But now two related sets of technologies have matured to the level that they are used in commercial environments. The first of these, which we may group under the heading software pipelining, schedules a loop so that successive iterations of the loop will execute concurrently while producing code that is as compact as possible. The second set of techniques, which we may call trace directed scheduling, deal with scheduling computations with a more general flow of control, either where the loop structure is not relevant, or within a (possibly software pipelined) loop body with a complex flow of control.

Loop parallelism and software pipelining. We begin by considering the loop around the basic block D, which is the most frequently executed basic block in the program. A record of execution of loop D would be ideal if it executes all operations as soon as their inputs and an appropriate functional unit are available. This is what a dataflow processor would do. If we knew the exact trip count, that is, the number of iterations of the loop actually executed, we could achieve the same result as the dataflow processor by unrolling the loop completely, treating the resulting code as a single basic block and generating the best possible schedule. Unfortunately, we do not generally know the trip count at compile-time and, even if we did, the unrolled code would usually be too large for this to be a practical strategy. Our goal, therefore, is to approach the performance of this impractical approach in a practical way. To achieve this, imagine the following conceptual strategy. First, we unroll the loop completely. Then we schedule the code, but with two constraints: (i) all iterations have identical schedules except that (ii) each iteration is scheduled some fixed number of cycles later than the previous iteration.

This fixed delay between the start of successive iterations is termed the initiation interval. This unrolled code, after scheduling, is repetitive except for a small portion at the beginning and, likewise, a small portion at the end. This repetitive portion can be re-rolled to yield a new loop which is known as the kernel. The prologue is the code corresponding to the record of execution that precedes the repetitive part and the epilogue is the code corresponding to the record of execution following the repetitive part. By executing the prologue, followed by the kernel an appropriate number of times, and finally the epilogue, one would come close to re-creating the ideal record of execution of the unrolled code. Thus, a relatively small amount of code is able to approximate the ideal (but impractical) strategy of unrolling the loop completely. This technique for executing loops is known as software pipelining (*26–29*).

Software pipelined code (Fig. 6) can be generated using an algorithm known as modulo scheduling (*26*). There are two steps to this: first, determining the initiation interval and, second, creating the schedule. The objective is to come up with a schedule having the smallest possible initiation interval, since this corresponds to the maximum performance. The limiting factor in deciding the initiation interval can either be a critical chain of dependences running through the loop iterations or a critical resource that is utilized fully. Scheduling itself proceeds much like local scheduling except that in step 3, resource conflicts must be avoided not only with operations from the same iteration but with operations from previous and subsequent iterations as well. This is done by ensuring that, within a single iteration, no machine resource is used at two points in time that are separated by a time interval that is a multiple of the iteration interval.

Compared to locally scheduled code, whose execution time is 21,960 cycles (Fig. 3C), the execution time with a software pipelined loop D drops to 17,520 cycles (Fig. 3D) for a further 26% increase in performance over local scheduling, yielding 1.05 operations per cycle.

Trace scheduling. Intuitively, one might approach the problem of scheduling operations globally as follows: first, schedule each basic block; second, move operations about from block to block to improve the schedule. Unfortunately, this will cause too many decisions, such as which registers and functional units to use for each operation, to be made with a local perspective, causing many false conflicts from a global viewpoint and greatly limiting the quality of the schedule. Instead, trace scheduling (*30, 31*) works as follows:

1) The scheduler selects a trace, that is, a linear sequence of basic blocks. Frequency profiles (Fig. 2B) are used to prune this selection to the most frequently executed code not yet scheduled. With the use of loop unrolling and other techniques, large bodies of code can be considered simultaneously (Fig. 7). (In practice, critical sections of code might be unrolled a lot, though the compiler must be mindful of how much code space is being used in doing so.)

2) The entire trace is considered at once. The compiler treats the trace as if it were a single basic block and schedules it in a manner

BLOCK	INT1	INT2	MEM1	MEM2	BRANCH
C:	-	C2	-	-	-
	1D5	-	1D1	-	-
	1D2	-	-	-	1D6
	2D5	1D3	2D1	-	-
	2D2	C1	-	-	2D6
D:	3D5	2D3	3D1	-	-
	3D2	1D4	-	-	3D6
E:	-	3D3	-	-	-
	-	2D4	-	-	-
	-	-	-	-	-
	-	3D4	-	-	E1

Fig. 6. VLIW software pipelined code for loop D. For this loop, the integer units are the critical resources. Since there are three integer operations per iteration and only two integer units, the maximum sustained rate at which new iterations can be started is one iteration every two cycles. Taken together, the prologue, kernel and epilogue correspond to three copies of the original block D. The operations from these three copies are labeled with the prefix 1, 2 or 3. The new basic block D (after modulo scheduling with an initiation interval of 2) consists of only the kernel of the software pipelined loop. The prologue has been merged in with basic block C, and the epilogue with basic block E. Since 1D1 in the prologue is dependent on C2, C2 must be scheduled one cycle earlier than the start of the prologue. On the other hand, C1 can be scheduled as late as the last instruction in the prologue since it is the predecessor of 1D4. E1, the jump back to basic block F, must be scheduled in the last instruction of E, that is, at the end of the epilogue. Good superscalar code would result from a linear ordering of the VLIW code obtained by a left-to-right, top-to-bottom scan of the VLIW code.

similar to the local scheduling described above. In general, the compiler will generate synthetic data-precedence edges to prevent the speculative execution of operations that would be illegal because they would have permanent effects. During this process, branches are given no special consideration. With the help of profile information, the compiler can choose to schedule operations early, to delay operations, or to do whatever seems desirable given the resources at hand. Register allocation, functional unit selection, and so on, are done only as an operation is being scheduled, preventing arbitrary choices from unnecessarily constraining the schedule.

3) After scheduling the trace, the compiler must under some circumstances duplicate operations that have been scheduled. This occurs for two reasons: first, some operations will have been scheduled after a conditional jump that they used to precede. In that case, the operation in question must be duplicated so that it appears in the off-trace target of the branch. Second, when there are rejoins, the rejoin must jump to a place after which only operations that were below the original rejoin may be found. The highest such place in the schedule formed by the compiler may be below the scheduled location of some of the operations that originally were after the rejoin. These operations must then be copied to the end of the off-trace block that is jumping to the new rejoin location. In practice, this extra code has been relatively small, but avoiding the generation of too much code when the flow of control is very complex can be a concern when scheduling for VLIWs and superscalar processors.

The result of this process is trace scheduled code (Fig. 8). With trace scheduling (and the previously applied software pipelining), our example executes in 8500 cycles, yielding 2.16 operations per cycle (Fig. 3E). This translates to a factor of 2.6 improvement over local ILP, and a 3.5 factor improvement over serial execution. In the procedure outlined above, we restricted ourselves to selecting a set of basic blocks that constitute a linear path through the code. In many cases, it might be desirable to select a set of blocks that represent a more general flow of control. Extensions of the above scheduling procedure can handle this more general case (*32, 33*).

Predicated execution. As we saw earlier, in certain cases there exists an alternative to this speculative approach to exploiting inter-block parallelism. This arises when there are distinct computations or portions of the program that can be executed in parallel without having to speculate. A dataflow processor (with multiple loci of control) is well suited to exploiting such parallelism; each locus of control executes a separate portion of the computation. Since each locus of control is independently and concurrently executing branches, the number of ways in which the overall computation can evolve is combinatorial in nature. This poses a problem for a VLIW or superscalar processor that is trying to emulate the record of execution of the dataflow machine, handicapped as it is by having only a single locus of control; a distinct code sequence must exist for each possible record of execution that can result on the dataflow machine, often leading to an intolerable amount of code.

Predicated execution is a mechanism that allows a uniprocessor to more efficiently emulate the dataflow processor. To begin with, all branches are eliminated in each of the code regions of interest and each operation is provided, as its predicate input, with a boolean value that is true if and only if flow of control would have passed through this operation in the original code. This process is termed IF-conversion. Given that all branches have been eliminated in the regions of interest, these regions can be scheduled to execute in parallel with no combinatorial problems of code size. During execution, the selective suppression of operations by the predicates yields the various combinations of execution records that would have been generated by a multiple locus machine. However, there are two drawbacks to this approach: first, the selective suppression of the predicated operations results in wasted execution cycles. Depending on the nature of the computation, the number of such wasted cycles may either be greater than or less than the number wasted during speculative execution. Second, the different paths through the original code may have different lengths. The IF-converted code must take as long as the longest path even when a shorter path is to be executed and, if this computation is on the

Operation Label	Basic Block Label	Statement
1A1	1A:	T1 = LOAD X(I-1)
1A2		T2 = LOAD X(I)
1A3		T3 = 2 * T1
1A4		T4 = T3 - T2
1A5		IF T4 > 0 GOTO 1C
1B1	1B:	T5 = T2 - T1
1B2		U1 = 2 * T5
1F1	1F:	T8 = 3 * V
1F2		V = T8 + U1
1F3		I = I + 1
1F4		IF I > 1000 GOTO EXIT
2A1	2A:	T11 = LOAD X(I-1)
2A2		T12 = LOAD X(I)
2A3		T13 = 2 * T11
2A4		T14 = T13 - T12
2A5		IF T14 > 0 GOTO 2C
2B1	2B:	T15 = T12 - T11
2B2		U2 = 2 * T15
2F1	2F:	T18 = 3 * V
2F2		V = T18 + U2
2F3		I = I + 1
2F4		IF I ≤ 1000 GOTO 1A
	G:	

Fig. 7. Code for the unrolled trace. The body of the outer loop now contains two copies each of every basic block in the original loop body. The labels of the two copies of a basic block are distinguished by a prefix of either 1 or 2. Likewise, the labels of the two copies of each operation are distinguished by a similar prefix. The temporary variables that hold the results of the second set of operations have been renamed in a systematic fashion. For instance, the temporary variable T1 has been renamed T11 in basic block 2A. Since the expectation is that the trace will tend not to be exited, the sense of branches 1A5, 1F4 and 2A5 has been reversed. The trace includes only the basic blocks 1A, 1B, 1F, 2A, 2B and 2F. Not shown are two copies, 1D and 2D, of the software pipelined inner loop as well as basic blocks 1C, 1E, 2C and 2E.

	BLOCK	INT1	INT2	MEM1	MEM2	BRANCH
1	1A:	1F1	1F3	1A2	1A1	–
2		–	–	2A2	2A1	–
3		1A3	1B1	–	–	–
4		2A3	1B2	–	–	–
5		–	2B1	–	–	–
6		–	2B2	–	–	–
7		1F2	1A4	–	–	–
8		2A4	2F1	–	–	1A5
9	1F:	2F3	–	–	–	1F4
10	2A:	–	–	–	–	2A5
11	2F:	2F2	–	–	–	2F4
12	G:	–	–	–	–	–

Fig. 8. Scheduled code (which is also the record of execution) for the unrolled trace in Fig. 7. The operations in the trace are scheduled as if the trace is a single basic block, even though it actually consists of six basic blocks. This results in operations freely moving above or below branches determined only by what yields a good schedule. The relative ordering of branches is not altered. After scheduling, the demarcation between blocks can be re-established to determine which operations have moved from one basic block to another. Each branch defines the end of its basic block. Thus 1A ends at time 8 since 1A5 is scheduled that time. Likewise, 1F, 2A and 2F end at times 9, 10, and 11, respectively. The boundary between basic blocks 1B and 1F and between 2B and 2F are defined by the points at which the branches from 1E and 2E, respectively, enter the trace. The rule used here is that the rejoin after trace scheduling should be at the earliest point that does not include operations that were originally above the rejoin. The earliest instruction, that does not include any operations that originally were from 1A or 1B, is instruction 9. Since instruction 8 is part of 1A and instruction 9 is in 1F, this means that 1B is now an empty block. Likewise, 2B is an empty block. Once the basic blocks have been demarcated, it is clear what code motion has been effected. For instance, 2F1 has been speculatively moved up by five blocks, past three conditional branches and two rejoins, into 1A. In general, if an operation is moved up past a rejoin, it must be copied into the off-trace code just prior to the rejoin. The rejoin at the top of 1F comes after 1F2 which must, therefore, be copied to the end of 1E. Likewise, code that has moved down past a conditional branch must be copied into the off-trace code immediately following the exit. Once again, good superscalar code would result from a linear ordering of the VLIW code obtained by a left-to-right, top-to-bottom scan of the VLIW code.

critical path, performance is degraded.

Predicated execution is often beneficial when executing loops which have branches in the body of the loop. The trace A-B-F through the outer loop of the sample code is such an example because it contains the branch A5. The successive iterations of the trace are the computations that should be executed in parallel. After IF-conversion, the operations in the trace can be software pipelined in much the same way that loop D was. This results in an overall execution time of 7008 cycles, yielding an average of 2.61 operations per cycle (Fig. 3F). This constitutes more than a fourfold speedup over the serial execution of the program. The software pipelining of a trace through the outer loop also illustrates the point that trace scheduling and software pipelining can be applied either singly, as alternatives to one another, or in conjunction with one another.

Future Work

In the past 3 years, ILP has gone from being a 30-year-old field of research with a handful of papers produced each year, to being one of the two or three dominant areas in the field of computer architecture. However, ILP is an extremely controversial field of computer science. Most researchers agree on the desirability and practicality of ILP and sophisticated compiling, but the consensus stops there. All three types of implementations discussed above have their advocates, as do various blends of those technologies. A better understanding of the relative merits of the alternatives is central to the future design and use of ILP systems. Much of the research in progress involves the quantitative evaluation of the alternatives presented above, and much work remains in the invention and prototyping of new techniques to exploit ILP.

Available parallelism. One area of investigation involves measuring how much parallelism is available in programs. Many of the disagreements explored above may boil down to this issue. Unfortunately, there is little agreement about what workloads are important. Often, depending upon their backgrounds, researchers have very different views on this matter. This contributes to the differences of opinion since the amount of ILP available varies dramatically with the choice of programs being measured. If relatively little ILP is available, perhaps a factor of 2 or 3, then the arguments in favor of superscalar architectures may become overwhelming. If instead the typical available improvement factor is in the 5 to 20 range, then the objections to VLIWs may be small compared to the difficulty of building a superscalar to exploit that much ILP; many researchers believe that such a superscaler would be completely impractical. Finally, if massive quantities of ILP are typically available, dataflow may turn out to be the only architecture that can exploit it. In the authors' opinion, most of the debates about ILP will remain hollow until the type and amount of parallelism available in programs is classified and quantified.

Object-code compatibility. Processors that can each run the same object code, unchanged, are said to be object-code compatible. This is desirable, since it permits the replacement of an old processor with a new one without recompilation. Superscalar processors are more flexible than VLIW processors when new hardware technologies mandate changed latencies, since they schedule operations after, rather than before, the object code is produced. Important unanswered questions relate to this issue: are superscalar processors as flexible as they seem, or is recompilation required for them as well if they are to perform well when latencies change? Instead of distributing machine-level object-code, can the paradigm of software distribution change to one with a greater emphasis on a language level representation that is between the machine language and source language levels?

Architectural variations. Another important area of research is the design of hybrid architectures and processors which distill the good properties of each of the above classifications while overcoming their shortcomings. The Horizon processor is a good example of such a compromise. Other important questions remain in the design of processors, such as whether to issue multiple operations per cycle or whether to design machines which are superpipelined.

Speculative execution. ILP is an area that has had a relatively real-world orientation. There have been several commercial implementations of ILP processors which have been quite novel and ambitious, but which have, nevertheless, only scratched the surface of what is possible. Much engineering and research remains outstanding in the areas of register allocation, handling general flow of control, the use of predicates, and so forth. Studies have suggested that the bulk of ILP is accessible only when one is willing to do large amounts of speculative execution. Yet only a few systems have begun to wrestle with the practical implications of speculatively triggering error conditions, which might or might not be spurious. Also, none of the systems built to date have seriously tried to extend ILP to nonscientific codes, where the memory disambiguation problem becomes much more difficult.

The merits of dataflow processors. Dataflow architectures have held great promise for over two decades. The dataflow model of computation is also an important unifying concept in ILP: it serves as an idealized framework and reference standard for ILP. But the commercial viability of dataflow architectures has remained a controversial question. The dataflow debate revolves around its potential for massive levels of ILP with excellent object code compatibility, its shortcomings in supporting existing programs written in conventional languages or programs with little parallelism, and the practicality of its synchronization hardware.

Software pipelining versus loop unrolling. The compiler techniques that we have described apply equally to VLIW and superscalar implementations. However, there are many open questions in the areas of scheduling, register allocation, and code generation. For example, we have presented two different approaches to scheduling loop codes: software pipelining and the trace scheduling of unrolled loops. There are examples of code where one or the other is clearly superior, but little is known about where the boundary lies. At other times, software pipelining and trace scheduling are complementary and work well in concert. Both sets of techniques have been implemented in commercial processors (*19, 20*), but neither implementation was a suitable testbed on which these issues could be explored.

There is little argument about the desirability of allowing software to arrange code for more ILP. But whether a large increment over what can be done today is desirable, possible, or practical is a controversial question. Whether a significant amount of this work can realistically be done in the hardware is an area with more opinions than facts.

REFERENCES AND NOTES

1. P. P. Gelsinger, P. A. Gargini, G. H. Parker, A. Y. C. Yu, *IEEE Spectrum* **26**, 43 (1989).
2. Data and instruction caches are fast, small memories which automatically retain frequently referenced data and instructions, allowing much faster access most of the time.
3. Pipelined functional units have the ability to break an operation into smaller steps, allowing new operations to start before earlier ones have finished using the same arithmetic unit.
4. E. M. Riseman and C. C. Foster, *IEEE Transactions on Computers* **21**, 1405 (1972).
5. C. C. Foster and E. M. Riseman, *ibid.*, p. 1411.
6. A. Nicolau and J. A. Fisher, in *Proceedings of the Fourteenth Annual Microprogramming Workshop* (IEEE, New York, NY, 1981), pp. 171–182.
7. D. W. Wall, in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, New York, NY, 1991), pp. 176–188.
8. S. McFarling and J. Hennessy, in *Proceedings of the Thirteenth International Symposium on Computer Architecture* (IEEE, New York, NY, 1986), pp. 396–403.
9. J. A. Fisher, *The Static Jump Predictability of Programs+Data for Instruction-Level Parallelism*, Hewlett-Packard Laboratories Technical Report (in press).
10. J. E. Thornton, in *Procedings of the AFIPS Fall Joint Computer Conference* (AFIPS, New York, NY, 1964), vol. 26, p. 33.
11. D. W. Anderson, F. J. Sparacio, R. M. Tomasulo, *IBM J. Res. Develop.* **11**, 8 (1967).
12. *80960CA User's Manual*, No. 270710-001 (Intel Corporation, Santa Clara, CA, 1989).
13. *IBM J. Res. Develop.* **34** (1990) (special issue on IBM RISC System/6000 processor).
14. M. Johnson, *Superscalar Microprocessor Design* (Prentice-Hall, Englewood Cliffs, NJ 1991).
15. Arvind and K. Gostelow, *IEEE Computer* **15** (1982).
16. J. Gurd, C. C. Kirkham, I. Watson, *Commun. ACM* **28**, 34 (1985).
17. G. M. Papadopoulos and D. E. Culler, in *Proceedings of the Seventeenth International Symposium on Computer Architecture* (ACM, New York, NY 1990), pp. 82–91.
18. A. E. Charlesworth, *IEEE Computer* **14**, 18 (1981).
19. R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, P. K. Rodman, in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, New York, NY, 1987), pp. 180–192.
20. B. R. Rau, D. W. L. Yen, W. Yen, R. A. Towle, *IEEE Computer* **22** (1989).
21. *i860 64-Bit Microprocessor Programmer's Reference Manual*, No. 240329-001 (Intel Corporation, Santa Clara, CA, 1989).
22. Note that our list does not include vector architectures (and associated processors), although one could be defined corresponding to our assumed execution hardware (Fig. 1). Vector processors are not true ILP processors. They are best thought of as complex instruction set computers (CISC), that is, processors for a sequential architecture with (complex) vector instructions which possess a certain stylized parallelism internal to each vector instruction. However, despite the commercial success that they have had, vector processors are less general in their ability to exploit all forms of instruction-level parallelism due to their stylized approach to parallelism. Vector processors are treated in many textbooks, for instance, the one by K. Hwang and F. A. Briggs [Computer Architecture and Parallel Processing (McGraw-Hill, New York, 1984)].
23. R. M. Tomasulo, *IBM J. Res. Develop.* **11**, 25 (1967).
24. N. P. Jouppi, *IEEE Transactions on Computers* **38**, 1645 (1989).
25. M. R. Thistle and B. J. Smith, in *Proceedings of Supercomputing '88* (IEEE, New York, NY, 1988), pp. 35–41.
26. B. R. Rau and C. D. Glaeser, in *Proceedings of the Fourteenth Annual Workshop on Microprogramming* (IEEE, New York, NY, 1981), pp. 183–198.
27. P. Y. T. Hsu, *Highly Concurrent Scalar Processing Technical Report No. CSG-49* (University of Illinois, Urbana, 1986).
28. M. Lam, in *Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (ACM, New York, NY, 1988), pp. 318–328.
29. J. C. Dehnert, P. Y.-T. Hsu, J. P. Bratt, in *Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, New York, NY, 1989), pp. 26–38.
30. J. A. Fisher, *IEEE Transactions on Computers* **30**, 478 (1981).
31. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures* (MIT Press, Cambridge, MA, 1985).
32. J. L. Lynn, in *Proceedings of the Sixteenth Annual Workshop on Microprogramming* (IEEE, New York, NY, 1983), pp. 11–22.
33. J. A. Fisher, *Trace Scheduling-2, an Extension of Trace Scheduling*, Hewlett-Packard Laboratories Technical Report (in press).