The Journal of Supercomputing, 7, 9–50 (1993)

Instruction-Level Parallel Processing: History, Overview, and Perspective

B. RAMAKRISHNA RAU AND JOSEPH A. FISHER
Hewlett-Packard Laboratories, 1501 Page Mill Road, Bldg. 3U, Palo Alto, CA 94304

(October 20, 1992)

Abstract. Instruction-level parallelism (ILP) is a family of processor and compiler design techniques that speed up execution by causing individual machine operations to execute in parallel. Although ILP has appeared in the highest performance uniprocessors for the past 30 years, the 1980s saw it become a much more significant force in computer design. Several systems were built and sold commercially, which pushed ILP far beyond where it had been before, both in terms of the amount of ILP offered and in the central role ILP played in the design of the system. By the end of the decade, advanced microprocessor design at all major CPU manufacturers had incorporated ILP, and new techniques for ILP had become a popular topic at academic conferences. This article provides an overview and historical perspective of the field of ILP and its development over the past three decades.

Keywords. Instruction-level parallelism, VLIW processors, superscalar processors, pipelining, multiple operation issue, speculative execution, scheduling, register allocation.

1. Introduction

Instruction-level parallelism (ILP) is a family of processor and compiler design techniques that speed up execution by causing individual machine operations, such as memory loads and stores, integer additions, and floating point multiplications, to execute in parallel. The operations involved are normal RISC-style operations, and the system is handed a single program written with a sequential processor in mind. Thus an important feature of these techniques is that like circuit speed improvements, but unlike traditional multiprocessor parallelism and massive parallel processing, they are largely transparent to users. VLIWs and superscalars are examples of processors that derive their benefit from instruction-level parallelism, and software pipelining and trace scheduling are example software techniques that expose the parallelism that these processors can use.

Although small amounts of ILP have been present in the highest performance uniprocessors of the past 30 years, the 1980s saw it become a much more significant force in computer design. Several systems were built and sold commercially, which pushed ILP far beyond where it had been before, both in terms of the amount of ILP offered and in the central role ILP played in the design of the system. By the early 1990s, advanced microprocessor design at all major CPU manufacturers incorporated ILP, and new techniques for ILP became a popular topic at academic conferences. With all of this activity we felt that, in contrast to a report on suggested future techniques, there would be great value in gathering, in an archival reference, reports on experience with real ILP systems and reports on the measured potential of ILP. Thus this special issue of *The Journal of Supercomputing*.

1.1. ILP Execution

A typical ILP processor has the same type of execution hardware as a normal RISC machine. The differences between a machine with ILP and one without is that there may be more of that hardware, for example, several integer adders instead of just one, and that the control will allow, and possibly arrange, simultaneous access to whatever execution hardware is present.

Consider the execution hardware of a simplified ILP processor consisting of four functional units and a branch unit connected to a common register file (Table 1). Typically ILP execution hardware allows multiple-cycle operations to be pipelined, so we may assume that a total of four operations can be initiated each cycle. If in each cycle the longest latency operation is issued, this hardware could have ten operations "in flight" at once, which would give it a maximum possible speedup of a factor of ten over a sequential processor with similar execution hardware. As the papers in this issue show, this execution hardware resembles that of several VLIW processors that have been built and used commercially, though it is more limited in its amount of ILP. Several superscalar processors now being built also offer a similar amount of ILP.

There is a large amount of parallelism available even in this simple processor. The challenge is to make good use of it—we will see that with the technology available today, an ILP processor is unlikely to achieve nearly as much as a factor of ten on many classes of programs, though scientific programs and others can yield far more than that on a processor that has more functional units. 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? In reality, as we shall see in Section 4, the two questions have to be reversed; in the absence of techniques to find and exploit ILP, it remains hidden, and we are left with a pessimistic answer.

Figure 1a shows a very large expression taken from the inner loop of a compute-intensive program. It is presented cycle by cycle as it might execute on a processor with functional units similar to those shown in Table 1, but capable of having only one operation in flight

Table 1. Execution hardware for a simplified ILP processor.

Functional Unit	Operations Performed	Latency
Integer unit 1	Integer ALU operations	1
	Integer multiplication	2
	Loads	2
	Stores	1
Integer unit 2/branch unit	Integer ALU operations	1
	Integer multiplication	2
	Loads	2
	Stores	1
	Test-and-branch	1
Floating point unit 1 Floating point unit 2	Floating point operations	3

```
CYCLE  1 xseed1 = xseed * 1309
CYCLE  2 nop
CYCLE  3 nop
CYCLE  4 yseed1 = yseed * 1308
CYCLE  5 nop
CYCLE  6 nop
CYCLE  7 xseed2 = xseed1 + 13849
CYCLE  8 yseed2 = yseed1 + 13849
CYCLE  9 xseed  = xseed2 && 65535
CYCLE  0 yseed  = yseed2 && 65535
CYCLE 11 tseed1 = tseed * 1307
CYCLE 12 nop
CYCLE 13 nop
CYCLE 14 vseed1 = vseed * 1306
CYCLE 15 nop
CYCLE 16 nop
CYCLE 17 tseed2 = tseed1 + 13849
CYCLE 18 vseed2 = vseed1 + 13849
CYCLE 19 tseed  = tseed2 && 65535
CYCLE 20 vseed  = vseed2 && 65535
CYCLE 21 xsq = xseed * xseed
CYCLE 22 nop
CYCLE 23 nop
CYCLE 24 ysq = yseed * yseed
CYCLE 25 nop
CYCLE 26 nop
CYCLE 27 xysumsq = xsq + ysq
CYCLE 28 tsq = tseed * tseed
CYCLE 29 nop
CYCLE 30 nop
CYCLE 31 vsq = vseed * vseed
CYCLE 32 nop
CYCLE 33 nop
CYCLE 34 tvsumsq = tsq + vsq
CYCLE 35 plc = plc + 1
CYCLE 36 tp = tp + 2
CYCLE 37 if xysumsq > radius goto @xy-no-hit
```

(a)

```
        INT ALU                 INT ALU             FLOAT ALU             FLOAT ALU

CYCLE  1 tp=tp+2               plc=plc+1           vseed1=vseed*1306 tseed1=tseed*1307
CYCLE  2                                           yseed1=yseed*1308 xseed1=xseed*1309
CYCLE  3 nop
CYCLE  4 vseed2=vseed1+13849 tseed2=tseed1+13849
CYCLE  5 yseed2=yseed1+13849 xseed2=xseed1+13849
CYCLE  6 yseed=yseed2&&65535 xseed=xseed2&&65535
CYCLE  7 vseed=vseed2&&65535 tseed=tseed2&&65535 ysq=yseed*yseed   xsq=xseed*xseed
CYCLE  8                                         vsq=vseed*vseed   tsq=tseed*tseed
CYCLE  9 nop
CYCLE  0 xysumsq=xsq+ysq
CYCLE 11 tvsumsq=tsq+vsq      if xysumsq>radius goto @xy-no-hit
```

(b)

Figure 1. (a) An example of the sequential record of execution for a loop. (b) The instruction-level parallel record of execution for the same loop.

at a time. Figure 1b shows the same program fragment as it might be executed on the hardware indicated in Table 1.

Note that several of the cycles in Figure 1a contain no-ops. This is because the sequential processor must await the completion of the three-cycle latency multiply issued in cycle 1 before issuing the next operation. (These no-ops would not appear in the text of a program, but are shown here as the actual record of what is executed each cycle.) Most instruction-level parallel processors can issue operations during these no-op cycles, when previous operations are still in flight, and many can issue more than one operation in a given cycle. In our ILP record of execution (Figure 1b), both effects are evident: In cycle 1, four operations are issued; in cycle 2, two more operations are issued even though neither multiply in cycle 1 has yet completed execution.

This special issue of *The Journal of Supercomputing* concerns itself with the technology of systems that try to attain the kind of record of execution in Figure 1b, given a program written with the record of execution in Figure 1a in mind.

1.2. Early History of Instruction-Level Parallelism

In small ways, instruction-level parallelism factored into the thinking of machine designers in the 1940s and 1950s. Parallelism that would today be called horizontal microcode appeared in Turing's 1946 design of the Pilot ACE [Carpenter and Doran 1986] and was carefully described by Wilkes [1951]. Indeed, in 1953 Wilkes and Stringer wrote, "In some cases it may be possible for two or more micro-operations to take place at the same time" [Wilkes and Stringer 1953].

The 1960s saw the appearance of transistorized computers. One effect of this revolution was that it became practical to build reliable machines with far more gates than was necessary to build a general-purpose CPU. This led to commercially successful machines that used this available hardware to provide instruction-level parallelism at the machine-language level. In 1963 Control Data Corporation started delivering its CDC 6600 [Thornton 1964, 1970], which had ten functional units—integer add, shift, increment (2), multiply (2), logical branch, floating point add and divide. Any one of these could start executing in a given cycle whether or not others were still processing data-independent earlier operations. In this machine the hardware decided, as the program executed, which operation to issue in a given cycle; its model of execution was well along the way toward what we would today call superscalar. Indeed, in many ways it strongly resembled its direct descendant, the scalar portion of the CRAY-1. The CDC 6600 was the scientific supercomputer of its day.

Also during the 1960s, IBM introduced, and in 1967–68 delivered, the 360/91 [IBM 1967]. This machine, based partly on IBM's instruction-level parallel experimental Stretch processor, offered less instruction-level parallelism than the CDC 6600, having only a single integer adder, a floating point adder, and a floating point multiply/divide. But it was far more ambitious than the CDC 6600 in its attempt to rearrange the instruction stream to keep these functional units busy—a key technology in today's superscalar designs. For various nontechnical reasons the 360/91 was not as commercially successful as it might have been, with only about 20 machines delivered [Bell and Newell 1971]. But its CPU architecture was the start of a long line of successful high-performance processors. As with the CDC 6600, this ILP pioneer started a chain of superscalar architectures that has lasted into the 1990s.

In the 1960s, research into "parallel processing" often was concerned with the ILP found in these processors. By the mid-1970s the term was used more often for multiple processor parallelism and for regular array and vector parallelism. In part, this was due to some very pessimistic results about the availability of ILP in ordinary programs, which we discuss below.

In the late 1970s the beginnings of a new style of ILP, called very long instruction word (VLIW), emerged on several different fronts. In many ways VLIWs were a natural outgrowth of horizontal microcode, the first ILP technology, and they were triggered, in the 1980s, by the same changes in semiconductor technology that had such a profound impact upon the entire computer industry.

For sequential processors, as the speed gap between writeable and read-only memory narrowed, the advantages of a small, dedicated, read-only control store began to disappear. One natural effect of this was to diminish the advantage of microcode; it no longer made as much sense to define a complex language as a compiler target and then interpret this in very fast read-only microcode. Instead, the vertical microcode interface was presented as a clean, simple compiler target. This concept was called RISC [Hennessy, Jouppi, Baskett et al. 1982; Patterson and Sequin 1981; Radin 1982]. In the 1980s the general movement of microprocessor products was towards the RISC concept, and instruction-level parallel techniques fell out of favor. In the minisupercomputer price-bracket though, one innovative superscalar product, the ZS-1, which could issue up to two instructions each cycle, was built and marketed by Astronautics [Smith et al. 1987].

The same changes in memory technology were having a somewhat different effect upon horizontally microcoded processors. During the 1970s a large market had grown in specialized signal processing computers. Not aimed at general-purpose use, these CPUs hardwired FFTs and other important algorithms directly into the horizontal control store, gaining tremendous advantages from the instruction-level parallelism available there. When fast, writeable memory became available, some of these manufacturers, most notably Floating Point Systems [Charlesworth 1981], replaced the read-only control store with writeable memory, giving users access to instruction-level parallelism in far greater amounts than the early superscalar processors had. These machines were extremely fast, the fastest processors by far in their price ranges, for important classes of scientific applications. However, despite attempts on the part of several manufacturers to market their products for more general, everyday use, they were almost always restricted to a narrow class of applications. This was caused by the lack of good system software, which in turn was caused by the idiosyncratic architecture of processors built for a single application, and by the lack at that time of good code generation algorithms for ILP machines with that much parallelism.

As with RISC, the crucial step was to present a simple, clean interface to the compiler. However, in this case the clean interface was horizontal, not vertical, so as to afford greater ILP [Fisher 1983; Rau, Glaeser, and Greenawalt 1982]. This style of architecture was dubbed VLIW [Fisher 1983]. Code generation techniques, some of which had been developed for generating horizontal microcode, were extended to these general-purpose VLIW machines so that the compiler could specify the parallelism directly [Fisher 1981; Rau and Glaeser 1981].

In the 1980s VLIW CPUs were offered commercially in the form of capable, general-purpose machines. Three computer start-ups—Culler, Multiflow, and Cydrome—built VLIWs with varying degrees of parallelism [Colwell et al. 1988; Rau et al. 1989]. As a group these companies were able to demonstrate that it was possible to build practical machines that achieved large amounts of ILP on scientific and engineering codes. Although, for various reasons, none was a lasting business success, several major computer manufacturers acquired access to the technologies developed at these start-ups and there are several active VLIW design efforts underway. Furthermore, many of the compiler techniques developed with VLIWs in mind, and reported upon in this issue, have been used to compile for superscalar machines as well.

1.3.1. ILP in the 1990s. Just as had happened 30 years ago when the transistor became available, CPU designers in the 1990s now have offered to them more silicon space on a single chip than a RISC processor requires. Virtually all designers have begun to add some degree of superscalar capability, and some are investigating VLIWs as well. It is a safe bet that by 1995 virtually all new CPUs will embody some degree of ILP.

Partly as a result of this commercial resurgence of interest in ILP, research into that area has become a dominant feature of architecture and systems conferences of the 1990s. Unfortunately, those researchers who found themselves designing state-of-the-art products at computer start-ups did not have the time to document the progress that was made and the large amount that was learned. Virtually everything that was done by these groups was relevant to what designers wrestle with today.

2. ILP Architectures

The end result of instruction-level parallel execution is that multiple operations are simultaneously in execution, either as a result of having been issued simultaneously or because the time to execute an operation is greater than the interval between the issuance of successive operations. 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 between the instructions or operations in the program. 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 are representative of ILP processor implementations for sequential architectures [Anderson et al. 1967; Apollo Computer 1988; Bahr et al. 1991; Blanck and Krueger 1992; DeLano et al. 1992; Diefendorff and Allen 1992; IBM 1990; Intel 1989b; Keller et al. 1975; Popescu et al. 1991; Smith et al. 1987; Thompson 1964].
- *Dependence architectures*: architectures for which the program explicitly indicates the dependences that exist between operations. Dataflow processors [Arvind and Gostelow 1982; Arvind and Kathail 1981; Gurd et al. 1985] 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 (VLIW) processors [Charlesworth 1981; Colwell et al. 1988; Rau et al. 1989] are examples of the class of independence architectures.

In the context of this taxonomy, vector processors [Hintz and Tate 1972; Russell 1978; Watson 1972] are best thought of as processors for a sequential, CISC (complex instruction set computer) architecture. The complex instructions are the vector instructions that do possess a stylized form of instruction-level parallelism internal to each vector instruction. Attempting to execute multiple instructions in parallel, whether scalar or vector, incurs all of the same problems that are faced by a superscalar processor. Because of their stylized approach to parallelism, vector processors are less general in their ability to exploit all forms of instruction-level parallelism. Nevertheless, vector processors have enjoyed great commercial success over the past decade. Not being true ILP processors, vector processors are outside the scope of this special issue. (Vector processors have received a great deal of attention elsewhere over the past decade and have been treated extensively in many books and articles, for instance, the survey by Dongarra [1986] and the book by Schneck [1987].) Also, certain hybrid architectures [Danelutto and Vanneschi 1990; Franklin and Sohi 1992; Wolfe and Shen 1991], which also combine some degree of multithreading with ILP, fall outside of this taxonomy for uniprocessors.

If ILP is to be achieved, between the compiler and the run-time hardware, the following functions must be performed:

1. The dependences between operations must be determined.
2. The operations that are independent of any operation that has not as yet completed must be determined.
3. These independent operations must be scheduled to execute at some particular time, on some specific functional unit, and must be assigned a register into which the result may be deposited.

Figure 2 shows the breakdown of these three tasks, between the compiler and run-time hardware, for the three classes of architecture.

2.1. Sequential Architectures and Superscalar Processors

The program for a sequential architecture contains no explicit information regarding the dependences that exist between instructions. Consequently, the compiler need neither identify parallelism nor make scheduling decisions since there is no explicit way to communicate this information to the hardware. (It is true, nevertheless, that there is value in the compiler performing these functions and ordering the instructions so as to facilitate the hardware's task of extracting parallelism.) In any event, if instruction-level parallelism is to be employed, the dependences that exist between instructions must be determined by the hardware. It is only necessary to determine dependences with sequentially preceding operations that are in flight, that is, those that have been issued but have not yet completed.

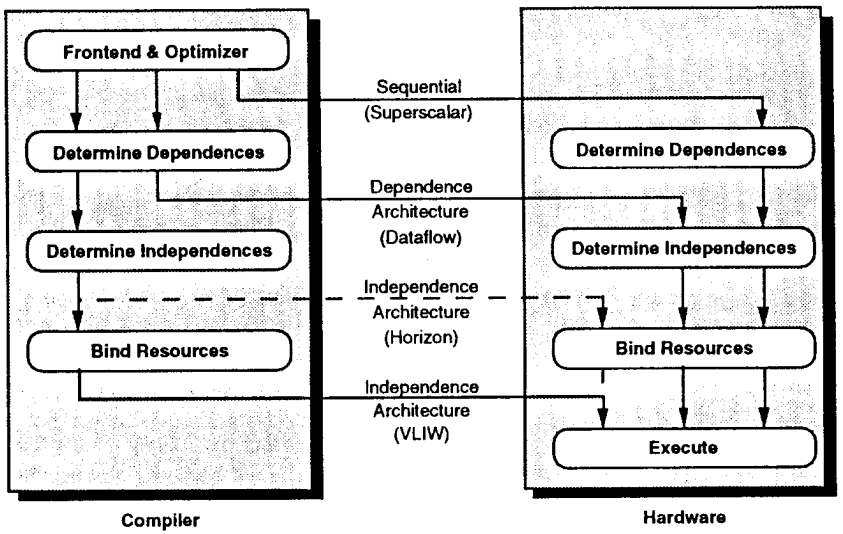

Figure 2. Division of responsibilities between the compiler and the hardware for the three classes of architecture.

When the operation is independent of all other operations it may begin execution. At this point the hardware must make the scheduling decision of when and where this operation is to execute.

A superscalar processor[1] strives to issue an instruction every cycle so as to execute many instructions in parallel, even though the hardware is handed a sequential program. The problem is that a sequential program is constructed with the assumption only that it will execute correctly when each instruction 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 in flight, that is, one 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.

If either of these conditions is true, the instruction in question must be delayed until the instructions on which it is dependent have completed execution. For each waiting operation, these dependences must be monitored to determine the point at which neither condition is true. When this happens, the instruction is independent of all other uncompleted instructions and can be allowed to begin executing at any time thereafter. In the meantime the processor may begin execution of subsequent instructions that prove to be independent

of all sequentially preceding instructions in flight. Once an instruction is independent of all other ones in flight, the hardware must also decide exactly when and on which available functional unit to execute the instruction. The Control Data CDC 6600 used a mechanism, called the *scoreboard*, to perform these functions [Thornton 1964]. The IBM System/360 Model 91, built in the early 1960s, used an even more sophisticated method known as Tomasulo's algorithm to carry out these functions [Tomasulo 1967].

The further goal of a superscalar processor is to issue *multiple* instructions every cycle. The most problematic aspect of doing so is determining the dependences between the operations that one wishes to issue simultaneously. Since the semantics of the program, and in particular the essential dependences, are specified by the sequential ordering of the operations, the operations must be processed in this order to determine the essential dependences. This constitutes an unacceptable performance bottleneck in a machine that is attempting parallel execution. On the other hand, eliminating this bottleneck can be very expensive, as is always the case when attempting to execute an inherently sequential task in parallel. An excellent reference on superscalar processor design and its complexity is the book by Johnson [1991].

A number of superscalar processors have been built during the past decade including the Astronautics' ZS-1 decoupled access minisupercomputer [Smith 1989; Smith et al. 1987], Apollo's DN10000 personal supercomputer [Apollo 1988; Bahr et al. 1991], and, most recently, a number of microprocessors [Blanck and Krueger 1992; DeLano et al. 1992; Diefendorff and Allen 1992; IBM 1990; Intel 1989b; Popescu et al. 1991].

Note that an ILP processor need not issue multiple operations per cycle in order to achieve a certain level of performance. For instance, instead of a processor capable of issuing five instructions per cycle, 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, which has been termed *superpipelining* [Jouppi 1989], goes full circle back to the single-issue, superscalar processing of the 1960s. Superpipelining may result in some parts of the processor (such as the instruction unit and communications buses) being less expensive and better utilized and other parts (such as the execution hardware) being more costly and less well used.

2.2. Dependence Architectures and Dataflow Processors

In the case of dependence architectures the compiler or the programmer identifies the parallelism in the program and communicates it to the hardware by specifying, in the executable program, the dependences between operations. The hardware must still determine, at run time, when each operation is independent of all other operations and then perform the scheduling. However, the inherently sequential task, of scanning the sequential program in its original order to determine the dependences, has been eliminated.

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 [Arvind and Gostelow 1982; Arvind and Kathail 1981]. To do so, it counts on the program to provide information about the dependences between instructions. Typically, 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.

Computation within a basic block typically does not provide adequate levels of parallelism. Superscalar and VLIW processors use control parallelism and speculative execution to keep the hardware fully utilized. (This is discussed in greater detail in Sections 3 and 4.) Dataflow processors have traditionally counted on using 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 since every instruction executed is a useful one and the processor does not have to deal with error conditions raised by speculative operations.

As far as the authors are aware, there have been no commercial products built based on the dataflow architecture, except in a limited sense [Schmidt and Caesar 1991]. There have, however, been a number of research prototypes built, for instance, the ones built at the University of Manchester [Gurd et al. 1985] and at MIT [Papadopoulos and Culler 1990].

2.3. Independence Architectures and VLIW 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 since programs for it lack any explicit information, it must also first determine the dependences between instructions. In contrast, for an independence architecture 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 operations of which it is independent is far greater than the number of operations on which it is dependent, 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 at all and the code is virtually identical to the desired record of execution. The VLIW processors that have been built to date are of this type and represent the predominant examples of machines with independence architectures. 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. A particular processor implementation of a VLIW architecture 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.

With a 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 that 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 nonspeculative 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, but it does so at compile time.

The earliest VLIW processors built were the so-called attached array processors [Charlesworth 1981; Floating Point Systems 1979; IBM 1976; Intel 1989a; Ruggiero and Coryell 1969] of which the best known were the Floating Point Systems products, the AP-120B, the FPS-164, and the FPS-264. The next generation of products were the minisupercomputers: Multiflow's Trace series of machines [Colwell et al. 1988; Colwell et al. 1990] and Cydrome's Cydra 5 [Beck et al. 1993; Rau 1988; Rau et al. 1989] and the Culler machine for which, as far as we are aware, there is no published description in the literature. Over the last few years the VLIW architecture has begun to show up in microprocessors [Kohn and Margulis 1989; Labrousse and Slavenburg 1988, 1990a, 1990b; Peterson et al. 1981].

Other types of processors with independence architectures have been built or proposed. A superpipelined machine may issue 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 dependences 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* [Thistle and Smith 1988], encodes an integer H into each operation. The architecture guarantees that all of the next H operations in the instruction stream are data-independent of the current operation. All the hardware has to do to release an operation, then, is to assure itself that no more than H subsequent operations are allowed to issue before this operation has completed. The hardware does all of its own scheduling, unlike the VLIWs and deeply pipelined machines that rely on the compiler, but the hardware is relieved of the task of determining data dependence. The key distinguishing features of these three ILP architectures are summarized in Table 2.

Table 2. A comparison of the instruction-level parallel architecture discussed in this paper.

	Sequential Architecture	Dependence Architecture	Independence Architecture
Additional information required in the program	None	Complete specification of dependences between operations	Minimally, a partial list of independences. Typically, a complete specification of when and where each operation is to be executed
Typical kind of ILP processor	Superscalar	Dataflow	VLIW
Analysis of dependences between operations	Performed by hardware	Performed by the compiler	Performed by the compiler
Analysis of independent operations	Performed by hardware	Performed by hardware	Performed by the compiler
Final operation scheduling	Performed by hardware	Performed by hardware	Typically, performed by the compiler
Role of compiler	Rearranges the code to make the analysis and scheduling hardware more successful	Replaces some analysis hardware	Replaces virtually all the analysis and scheduling hardware

3. Hardware and Software Techniques for ILP Execution

Regardless of which ILP architecture is considered, certain functions must be performed if a sequential program is to be executed in an ILP fashion. The program must be analyzed to determine the dependences; the point in time at which an operation is independent, of all operations that are as yet not complete, must be determined; scheduling and register allocation must be performed; often, operations must be executed speculatively, which in turn requires that branch prediction be performed. All these functions must be performed. The choice is, first, whether they are to be performed by the compiler or by run-time hardware and, second, which specific technique is to be used. These alternatives are reviewed in the rest of this section.

3.1. Hardware Features to Support ILP Execution

Instruction-level parallelism involves the existence of multiple operations in flight at any one time, that is, operations that have begun, but not completed, executing. This implies the presence of execution hardware that can simultaneously process multiple operations. This has, historically, been achieved by two mechanisms: first, providing multiple, parallel functional units and, second, pipelining the functional units. Although both are fairly similar from a compiler's viewpoint—the compiler must find enough independent operations to keep the functional units busy—they have their relative strengths and weaknesses from a hardware viewpoint.

In principle, pipelining is the more cost-effective way of building ILP execution hardware. For the relatively low cost of adding pipeline latches within each functional unit, the amount of ILP can be doubled, tripled, or more. The limiting factors in increasing the performance by this means are the data and clock skews and the latch setup and hold times. These issues were studied during the 1960s and 1970s, and the upper limits on the extent of pipelining were determined [Chen 1971; Cotten 1965, 1969; Fawcett 1975; Hallin and Flynn 1972]. However, the upper limit on pipelining is not necessarily the best from the viewpoint of achieved performance. Pipelining adds delays to the execution time of individual operations (even though multiples of them can be in flight on the same functional unit). Beyond a certain point, especially on computations that have small amounts of parallelism, the increase in the latency counterbalances the benefits of the increase in ILP, yielding lower performance [Kunkel and Smith 1986]. Parallelism achieved by adding more functional units does not suffer from this drawback, but has its own set of disadvantages. First, the amount of functional unit hardware goes up in linear proportion to the parallelism. Worse, the cost of the interconnection network and the register files goes up proportional to the square of the number of functional units since, ideally, each functional unit's output bus must communicate with every functional unit's input buses through the register file. Also, as the number of loads on each bus increases, so must the cycle time or the extent of pipelining, both of which degrade performance on computation with little parallelism.

The related techniques of pipelining and overlapped execution were employed as early as in the late 1950s in computers such as IBM's STRETCH computer [Bloch 1959; Buchholz 1962] and UNIVAC's LARC [Eckert et al. 1959]. Traditionally, overlapped execution refers to the parallelism that results from multiple active instructions, each in a different one of the phases of instruction fetch, decode, operand fetch, and execute, whereas pipelining is used in the context of functional units such as multipliers and floating point adders [Chen 1975; Kogge 1981]. (A potential source of confusion is that, in the context of RISC processors, overlapped execution and pipelining, especially when the integer ALU is pipelined, have been referred to as *pipelining* and *superpipelining*, respectively [Jouppi 1989].)

The organization of the register files becomes a major issue when there are multiple functional units operating concurrently. For ease of scheduling, it is desirable that every operation (except loads and stores) be register-register and that the register file be the hub for communication between all the functional units. However, with each functional unit performing two reads and one write per cycle from or to the register file, the implementation of the register file becomes problematic. The chip real estate of a multiported register file is proportional to the product of the number of read ports and the number of write ports. The loading of multiple read ports on each register cell slows down the access time. For these reasons, highly parallel ILP hardware is structured as multiple clusters of functional units, with all the functional units within a single cluster sharing the same multiported register files [Colwell et al. 1988; Colwell et al. 1990; Fisher 1983; Fisher et al. 1984]. Communication between clusters is slower and occurs with lower bandwidth. This places a burden upon the compiler to partition the computation intelligently across the clusters; an inept partitioning can result in worse performance than if just a single cluster were used, leaving the rest of them idle.

The presence of multiple, pipelined function units places increased demands upon the instruction issue unit. In a fully sequential processor, each instruction is issued after the previous one has completed. Of course, this totally defeats the benefits of parallel execution hardware. However, if the instruction unit attempts to issue an instruction every cycle, care must be taken not to do so if an instruction, upon which this one is dependent, is still not complete. The scoreboard in the CDC 6600 [Thornton 1964] was capable of issuing an instruction every cycle until an output dependence was discovered. In the process, instructions following one that was waiting on a flow dependence could begin execution. This was the first implementation of an out-of-order execution scheme. Stalling instruction issue is unnecessary on encountering an output dependence if register renaming is performed. The Tomasulo algorithm [Tomasulo 1967], which was implemented in the IBM System/360 Model 91 [Anderson et al. 1967], is the classical scheme for register renaming and has served as the model for subsequent variations [Hwu and Patt 1986, 1987; Oehler and Blasgen 1991; Popescu et al. 1991; Weiss and Smith 1984]. A different, programmatically controlled register renaming scheme is obtained by providing rotating register files, that is, base-displacement indexing into the register file using an instruction-provided displacement off a dedicated base register [Advanced Micro Devices 1989; Charlesworth 1981; Rau 1988; Rau et al. 1989]. Although applicable only for renaming registers across multiple iterations of a loop, rotating registers have the advantage of being considerably less expensive in their implementation than are other renaming schemes.

The first consideration given to the possibility of issuing multiple instructions per cycle from a sequential program was by Tjaden and Flynn [1970]. This line of investigation into the logic needed to perform multiple-issue was continued by various researchers [Acosta et al. 1986; Dwyer and Torng 1992; Hwu and Patt 1986, 1987; Tjaden and Flynn 1973;

Uht 1986; Wedig 1982]. This idea, of multiple instruction issue of sequential programs, was probably first referred to as superscalar execution by Agerwala and Cocke [1987]. A careful assessment of the complexity of the control logic involved in superscalar processors is provided by Johnson [1991]. An interesting variation on multiple-issue, which made use of architecturally visible queues to simplify the out-of-order execution logic, was the decoupled access/execute architecture proposed by Smith [1982] and subsequently developed as a commercial product [Smith 1989; Smith et al. 1987].

A completely different approach to achieving multiple instruction issue, which grew out of horizontal microprogramming, was represented by attached-processor products such as the Floating Point Systems AP-120B [Floating Point Systems 1979], the Polycyclic project at ESL [Rau and Glaeser 1981; Rau, Glaeser, and Greenwalt 1982; Rau, Glaeser, and Picard 1982], the Stanford University MIPS project [Hennessy, Jouppi, Przybylski et al. 1982] and the ELI project at Yale [Fisher 1983; Fisher et al. 1984]. The concept is to have the compiler decide which operations should be issued in parallel and to group them in a single, long instruction. This style of architecture, which was dubbed a *very long instruction word* (VLIW) architecture [Fisher 1983], has the advantage that the instruction issue logic is trivial in comparison to that for a superscalar machine, but suffers the disadvantage that the set of operations that are to be issued simultaneously is fixed once and for all at compile time. One of the implications of issuing multiple operations per instruction is that one needs the ability to issue (and process) multiple branches per second. Various types of multiway branches, each corresponding to a different detailed model of execution or compilation, have been suggested [Colwell et al. 1988; Ebcioglu 1988; Fisher 1980; Nicolau 1985a].

The first obstacle that one encounters when attempting ILP computation is the generally small size of basic blocks. In light of the pipeline latencies and the interoperation dependences, little instruction-level parallelism is to be found. It is important that operations from multiple basic blocks be executed concurrently if a parallel machine is to be fully utilized. Since the branch condition, which determines which block is to be executed next, is often resolved only at the end of a basic block, it is necessary to resort to speculative execution, that is, continuing execution along one or more paths before it is known which way the branch will go. Dynamic schemes for speculative execution [Hwu and Patt 1986, 1987; Smith and Pleszkun 1988; Sohi and Vajapayem 1987] must provide ways to

- terminate unnecessary speculative computation once the branch has been resolved,
- undo the effects of the speculatively executed operations that should not been executed,
- ensure that no exceptions are reported until it is known that the excepting operation should, in fact, have been executed, and
- preserve enough execution state at each speculative branch point to enable execution to resume down the correct path if the speculative execution happened to proceed down the wrong one.

All this can be expensive in hardware. The alternative is to perform speculative code motion at compile time, that is, move operations from subsequent blocks up past branch operations into preceding blocks. These operations will end up being executed before the branch that they were supposed to follow; hence, they are executed speculatively. Such code motion is fundamental to global scheduling schemes such as trace scheduling [Ellis 1985; Fisher 1979, 1981]. The hardware support needed is much less demanding: first, a mechanism to ensure that exceptions caused by speculatively scheduled operations are reported if and only if the flow of control is such that they would have been executed in the nonspeculative version of the code [Mahlke, Chen et al. 1992] and, second, additional architecturally visible registers to hold the speculative execution state. A limited form of speculative code motion is provided by the "boosting" scheme [Smith et al. 1992; Smith et al. 1990].

Since all speculative computation is wasted if the wrong path is followed, it is important that accurate branch prediction be used to guide speculative execution. Various dynamic schemes of varying levels of sophistication and practicality have been suggested that gather execution statistics of one form or another while the program is running [Lee and Smith 1984; McFarling and Hennessy 1986; Smith 1981; Yeh and Patt 1992]. The alternative is to use profiling runs to gather the appropriate statistics and to embed the prediction, at compile time, into the program. Trace scheduling and superblock scheduling [Hwu et al. 1989; Hwu et al. 1993] use this approach to reorder the control flow graph to reflect the expected branch behavior. Hwu and others claim better performance than with dynamic branch prediction [Hwu et al. 1989]. Fisher and Freudenberger [1992] have examined the extent to which branch statistics gathered using one set of data are applicable to subsequent runs with different data. Although static prediction can be useful for guiding both static and dynamic speculation, it is not apparent how dymamic prediction can assist static speculative code motion.

Predicted execution is an architectural feature that permits the execution of individual operations to be determined by an additional, Boolean input. It has been used to selectively squash operations that have been moved up from successor blocks into the delay slots of a branch operation [Ebcioglu 1988; Hsu and Davidson 1986]. In its more general form [Beck et al. 1993; Rau 1988; Rau et al. 1989] it is used to eliminate branches in their entirety over an acyclic region of a control flow graph [Dehnert and Towle 1993; Dehnert et al. 1989; Mahlke, Lin et al. 1992] that has been IF-converted [Allen et al. 1983].

3.2. ILP Compilation

3.2.1. Scheduling. Scheduling algorithms can be classified based on two broad criteria. The first one is the nature of the control flow graph that can be scheduled by the algorithm. The control flow graph can be described by the following two properties:

- whether it consists of a single basic block or multiple basic blocks, and
- whether it is an acyclic or cyclic control flow graph.

Algorithms that can only schedule single acyclic basic blocks are known as *local scheduling* algorithms. Algorithms that jointly schedule multiple basic blocks (even if these are multiple iterations of a single static basic block) are termed *global scheduling* algorithms. Acyclic global scheduling algorithms deal either with control flow graphs that contain no cycles or, more typically, cyclic graphs for which a self-imposed scheduling barrier exists

at each back edge in the control flow graph. As a consequence of these scheduling barriers, back edges present no opportunity to the scheduler and are therefore irrelevant to it. Acyclic schedulers can yield better performance on cyclic graphs by unrolling the loop, a transformation which though easier to visualize for cyclic graphs with a single back edge, can be generalized to arbitrary cyclic graphs. The benefit of this transformation is that the acyclic scheduler now has multiple iterations' worth of computation to work with and overlap. The penalty of the scheduling barrier is amortized over more computation. Cyclic global scheduling algorithms attempt to directly optimize the schedule across back edges as well. Each class of scheduling algorithms is more general than the previous one and, as we shall see, attempts to build on the intuition and heuristics of the simpler, less general algorithm. As might be expected, the more general algorithms experience greater difficulty in achieving near-optimality or of even articulating intuitively appealing heuristics.

The second classifying criterion is the type of machine for which scheduling is being performed, which in turn is described by the following assumed properties of the machine:

- finite versus unbounded resources
- unit latency versus multiple cycle latency execution, and
- simple resource usage patterns for every operation (i.e., each operation uses just one resource for a single cycle, typically during the first cycle of the operation's execution) versus more complex resource usage patterns for some or all of the operations.

Needless to say, real machines have finite resources, generally have at least a few operations that have latencies greater than one cycle, and often have at least a few operations with complex usage patterns. We believe that the value of a scheduling algorithm is proportional to the degree of realism of the assumed machine model.

Finally, the scheduling algorithm can also be categorized by the nature of the process involved in generating the schedule. At one extreme are one-pass algorithms that schedule each operation once and for all. At the other extreme are algorithms that perform an exhaustive, branch-and-bound style of search for the schedule. In between is a spectrum of possibilities such as iterative but nonexhaustive search algorithms or incremental algorithms that make a succession of elementary perturbations to an existing legal schedule to nudge it toward the final solution. This aspect of the scheduling algorithm is immensely important in practice. The further one diverges from a one-pass algorithm, the slower the scheduler gets until, eventually, it is unacceptable in a real-world setting.

3.2.1.1. Local Scheduling. Scheduling, as a part of the code generation process, was first studied extensively in the context of microprogramming. Local scheduling is concerned with generating as short a schedule as possible for the operations within a single basic block; in effect a scheduling barrier is assumed to exist between adjacent basic blocks in the control flow graph. Although it was typically referred to as *local code compaction*,[2] the similarity to the job of scheduling tasks on processors was soon understood [Adam et al. 1974; Baker 1974; Coffman 1976; Coffman and Graham 1972; Fernandez and Bussel 1973; Gonzalez 1977; Hu 1961; Kasahara and Narita 1984; Kohler 1975; Ramamoorthy et al. 1972], and a number of notions and algorithms from scheduling theory were borrowed by the microprogramming community. Attempts at automating this task have been made since at least the late 1960s [Agerwala 1976; Davidson et al. 1981; DeWitt 1975; Fisher 1979; 1981; Kleir and Ramamoorthy 1971; Landskov et al. 1989; Ramamoorthy and Gonzalez 1969; Tokoro et al. 1977; Tsuchiya and Gonzalez 1974, 1976; Wood 1978]. Since scheduling is known to be NP-complete [Coffman 1976], the initial focus was on defining adequate heuristics [Dasgupta and Tartar 1976; Fisher 1979; Gonzalez 1977; Mallett 1978; Ramamoorthy and Gonzalez 1969; Ramarmoorthy and Tsuchiya 1974]. The consensus was that list scheduling using the highest-level-first priority scheme [Adam et al. 1974; Fisher 1979] is relatively inexpensive computationally (a one-pass algorithm) and near-optimal most of the time. Furthermore, this algorithm has no difficulty in dealing with nonunit execution latencies.

The other dimension in which local scheduling matured was in the degree of realism of the machine model. From an initial model in which each operation used a single resource for a single cycle (the simple resource usage model) and had unit latency, algorithms for local scheduling were gradually generalized to cope with complex resource usage and arbitrary latencies [Dasgupta and Tartar 1976; DeWitt 1975; Kleir 1974; Mallett 1978; Ramamoorthy and Tsuchiya 1974; Tsuchiya and Gonzalez 1974; Yau et al. 1974] culminating in the fully general resource usage "microtemplate" model proposed in [Tokoro et al. 1981], and which was known in the hardware pipeline design field as a reservation table [Davidson 1971]. In one form or another, this is now the commonly used machine model in serious instruction schedulers. This machine model is quite compatible with the highest-level-first list scheduling algorithm and does not compromise the near-optimality of this algorithm [Fisher 1981].

3.2.1.2. Global Acyclic Scheduling. A number of studies have established that basic blocks are quite short—typically about 5–20 instructions on the average—so whereas local scheduling can generate a near-optimal schedule, data dependences and execution latencies conspire to make the optimal schedule itself rather disappointing in terms of its speedup over the original sequential code. Further improvements require overlapping the execution of successive basic blocks, which is achieved by global scheduling.

Early strategies for global scheduling attempted to automate and emulate the ad hoc techniques that hand coders practiced of first performing local scheduling of each basic block and then attempting to move operations from one block to an empty slot in a neighboring block [Tokoro et al. 1981; Tokoro et al. 1978]. The shortcoming of such an approach is that, during local compaction, too many arbitrary decisions have already been made that failed to take into account the needs of and opportunities in the neighboring blocks. Many of these decisions might need to be undone before the global schedule can be improved.

In one very important way the mindset inherited from microprogramming was an obstacle to progress in global scheduling. Traditionally, code compaction was focused on the objective of reducing the size of the microprogram so as to allow it to fit in the microprogram memory. In the case of individual basic blocks the objectives of local compaction and local scheduling are aligned. This alignment of objectives is absent in the global case. Whereas global code compaction wishes to minimize the sum of the code sizes for the individual basic blocks, global scheduling must attempt to minimize the total execution time of all the basic blocks. In other words, global scheduling must minimize the sum of the code sizes of the individual basic blocks *weighted by the number of times each basic block is executed*. Thus, effective global scheduling might actually increase the size of the program

by greatly lengthening an infrequently visited basic block in order to slightly reduce the length of a high-frequency basic block. This difference between global compaction and global scheduling, which was captured neither by the early ad hoc techniques nor by the syntactically-driven hierarchical reduction approach proposed by Wood [1979], was noted by Fisher [1979, 1981].

Furthermore, the focus of Fisher's work was on reducing the length of those *sequences* of basic blocks that are frequently executed by the program. These concepts were captured by Fisher in the global scheduling algorithm known as *trace scheduling* [Fisher 1979, 1981]. Central to this procedure is the concept of a trace, which is an acyclic sequence of basic blocks embedded in the control flow graph, that is, a path through the program that could conceivably be taken for some set of input data. Traces are selected and scheduled in order of their frequency of execution. The next trace to be scheduled is defined by selecting the highest frequency basic block that has not yet been scheduled as the seed of the trace. The trace is extended forward along the highest frequency edge out of the last block of the trace as long as that edge is also the most frequent edge into the successor block and as long as the successor block is not already part of the trace. Likewise, the trace is extended backwards, as well, from the seed block. The selected trace is then scheduled as if it were a single block; that is, there is no special consideration given to branches, except that they are constrained to remain in their original order. Implicit in the resulting schedule is interblock code motion along the trace in either the upward or downward direction. Matching off-trace code motions must be performed as prescribed by the rules of interblock code motion specified by Fisher. This activity is termed *bookkeeping*. Therafter, the next trace is selected and scheduled. This procedure is repeated until the entire program has been scheduled. The key property of trace scheduling is that, unlike previous approaches to global scheduling, the decisions as to whether to move an operation from one block to another, where to schedule it, and which register to allocate to hold its result (see Section 3.2.2 below) are all made jointly rather than in distinct compiler phases.

Fisher and his coworkers at Yale went on to implement trace scheduling in the Bulldog compiler as part of the ELI project [Fisher 1983; Fisher et al. 1986]. This trace scheduling implementation and other aspects of the Bulldog compiler have been extensively documented by Ellis [1986]. The motion of code downwards across branches and upwards across merges results in code replication. Although this is generally acceptable as the price to be paid for better global schedules, Fisher recognized the possibility that the greediness of highest-level-first list scheduling could sometimes cause more code motion and, hence, replication then is needed to achieve a particular schedule length [Fisher 1981]. Su and his colleagues have recommended certain heuristics for the list scheduling of traces to address this problem [Grishman and Su 1983; Su and Ding 1985; Su et al. 1984]. Experiments over a limited set of test cases indicate that these heuristics appear to have the desired effect.

The research performed in the ELI project formed the basis of the production-quality compiler that was built at Multiflow. One of the enhancements to trace scheduling implemented in the Multiflow compiler was the elimination of redundant copies of operations caused by bookkeeping. When an off-trace path, emanating from a branch on the trace, rejoins the trace lower down, an operation that is moved above the rejoin and all the way to a point above the branch can make the off-trace copy redundant under the appropriate circumstances. The original version of trace scheduling, oblivious to such situations, retains two copies of the operation. Gross and Ward [1990] describe an algorithm to avoid such redundancies. Freudenberger and Ruttenberg [1992] discuss the integrated scheduling and register allocation in the Multiflow compiler. Lowney and others provide a comprehensive description of the Multiflow compiler [1993].

Hwu and his colleagues on the IMPACT project have developed a variant of trace scheduling that they term *superblock scheduling* [Chang, Mahlke et al. 1991; Hwu and Chang 1988]. In an attempt to facilitate the task of incorporating profile-driven global scheduling into more conventional compilers, they separate the trace selection and code replication from the actual scheduling and bookkeeping. To do this, they limit themselves to only moving operations up above branches, never down, and never up past merges. To make this possible, they outlaw control flow into the interior of a trace by means of tail duplication, that is, creating a copy of the trace below the entry point and redirecting the incoming control flow path to that copy. Once this is done for each incoming path, the resulting trace consists of a sequence of basic blocks with branches out of the trace but no incoming branches except to the top of the trace. This constitutes a superblock, also known as an *extended basic block* in the compiler literature. Chang and Hwu [1988] have studied different trace selection strategies and have measured their relative effectiveness. A comprehensive discussion of the results and insights from the IMPACT project are provided in this special issue [Hwu et al. 1993].

Although the global scheduling of linear sequences of basic blocks represents a major step forward, it has been criticized for its total focus on the current trace and neglect of the rest of the program. For instance, if there are two equally frequent paths through the program that have basic blocks in common, it is unclear as part of which trace these blocks should be scheduled. One solution is to replicate the code as is done for superblock scheduling. The other is to generalize trace scheduling to deal with more general control flow graphs. Linn [1988] and Hsu and Davidson [1986] proposed profile-driven algorithms for scheduling trees of basic blocks in which all but the root basic block have a single incoming path. Nicolau [1985a, 1985b] attempted to extend global scheduling to arbitrary, acyclic control flow graphs using percolation scheduling. However, since percolation scheduling assumes unbounded resources, it cannot realistically be viewed as a scheduling algorithm. Percolation scheduling was then extended to nonunit execution latencies (but still with unbounded resources) [Nicolau and Potasman 1990].

The development of practical algorithms for the global scheduling of arbitrary, acyclic control flow graphs is an area of active research. Preliminary algorithms, assuming finite resources have been defined by Ebcioglu [Ebcioglu and Nicolau 1989; Moon and Ebcioglu 1992] and by Fisher [1992]. These are both generalizations of trace scheduling. However, there are numerous difficulties in the engineering of a robust and efficient scheduler of this sort. The challenges in this area of research revolve around finding pragmatic engineering solutions to these problems.

A rather different approach to global acyclic scheduling has been pursued in the IMPACT project [Mahlke, Lin et al. 1992]. An arbitrary, acyclic control flow graph, having a single entry can be handled by this technique. The control flow graph is IF-converted [Allen et al. 1983; Park and Schlansker 1991] so as to eliminate all branches internal to the flow graph. The resulting code, which is similar to a superblock in that it can only be entered at the top but has multiple exits, is termed a *hyperblock*. This is scheduled in much the

same manner as a superblock except that two operations with disjoint predicates (i.e., operations that cannot both be encountered on any single path through the original flow graph) may be scheduled to use the same resources at the same time. After scheduling, reverse IF-conversion is performed to regenerate the control flow graph. Portions of the schedule in which m predicates are active yield 2^m versions of the code.

3.2.1.3. Cyclic Scheduling. As with acyclic flow graphs, instruction-level parallelism in loops is obtained by overlapping the execution of multiple basic blocks. With loops, however, the multiple basic blocks are the multiple iterations of the same piece of code. The most natural extension of the previous global scheduling ideas to loops is to unroll the body of the loop some number of times and to then perform trace scheduling, or some other form of global scheduling, over the unrolled loop body. This approach was suggested by Fisher [Fisher et al. 1981]. A drawback of this approach is that no overlap is sustained across the back edge of the unrolled loop. Fisher and others went on to propose a solution to this problem, which is to continue unrolling and scheduling successive iterations until a repeating pattern is detected in the schedule. The repeating pattern can be rerolled to yield a loop whose body is the repeating schedule. As we shall see, this approach was subsequently pursued by various researchers. In the meantime, loop scheduling moved off in a different direction, which, as is true of most VLIW scheduling work, had its roots in hardware design.

Researchers concerned with the design of pipelined functional units, most notably Davidson and coworkers, had developed the theory of and algorithms for the design of hardware controllers for pipelines to maximize the rate at which functions could be evaluated [Davidson 1971, 1974; Davidson et al. 1975; Patel 1976; Patel and Davidson 1976; Thomas and Davidson 1974]. The issues considered here were quite similar to those faced by individuals programming the innermost loops of signal processing algorithms [Cohen 1978; Kogge 1973, 1974, 1977a, 1977b; Kogge and Stone 1973] on the early peripheral array processors [Floating Point Systems 1979; IBM 1976; Ruggiero and Coryell 1969]. In both cases the objective was to sustain the initiation of successive function evaluations (loop iterations) before prior ones had completed. Since this style of computation is termed pipelining in the hardware context, it was dubbed *software pipelining* in the programming domain [Charlesworth 1981].

Early work in software pipelining consisted of ad hoc hand-coding techniques [Charlesworth 1981; Cohen 1978]. Both the quality of the schedules and the attempts at automating the generation of software pipelined schedules were hampered by the architecture of the early array processors. Nevertheless, Floating Point Systems developed, for the FPS-164 array processor, a compiler that could software pipeline a loop consisting of a single basic block [Touzeau 1984]. Weiss and Smith [1987] note that a limited form of software pipelining was present both in certain hand-coded libraries for the CDC 6600 and also as a capability in the Fortran compiler for the CDC 6600.

The general formulation of the software pipelining process for single basic block loops was stated by Rau and others [Rau and Glaeser 1981; Rau, Glaeser, and Picard 1982] drawing upon and generalizing the theory developed by Davidson and his coworkers on the design of hardware pipelines. This work identified the attributes of a VLIW architecture that make it amenable to software pipelining, most importantly, the availability of conflict-free access to register storage between the output of a functional unit producing a result and the functional unit that uses that result. This provides freedom in scheduling each operation and is in contrast to the situation in array processors where, due to limited register file bandwidth, achieving peak performance required that a majority of the operations be scheduled to start at the same instant that their predecessor operations completed so that they could pluck their operands right off the result buses.

Rau and others also presented a condition that has to be met by any legal software pipelined schedule—the *modulo constraint*—and derived lower bounds on the rate at which successive iterations of the loop can be started, that is, the *initiation interval* (II). (II is also the length of the software pipelined loop, measured in VLIW instructions, when no loop unrolling is employed.) This lower bound on II, the *minimum initiation interval* (MII), is the maximum of the lower bound due to the resource usage constraints (ResMII) and the lower bound due to the cyclic data dependence constraints caused by recurrences (RecMII). This lower bound is applicable both to vectorizable loops as well as those with arbitrary recurrences and for operation latencies of arbitrary length. A simple, deterministic software pipelining algorithm based on list scheduling, the modulo scheduling algorithm, was shown to achieve the MII, thereby yielding an asymptotically optimal schedule. This algorithm was restricted to DO loops whose body is a single basic block being scheduled on a machine in which each operation has a simple pattern of resource usage, viz., the resource usage of each operation can be abstracted to the use of a single resource for a single cycle (even though the latency of the operation is not restricted to a single cycle). The task of generating an optimal, resource-constrained schedule for loops with arbitrary recurrences is known to be NP-complete [Hsu 1986; Lam 1987] and any practical algorithm must utilize heuristics to guide a generally near-optimal process. These heuristics were only broadly outlined in this work.

Three independent sets of activity took this work and extended it in various directions. The first one was the direct continuation at Cydrome, over the period 1984–88, of the work done by Rau and others [Dehnert et al. 1989; Dehnert and Towle 1993]. In addition to enhancing the modulo scheduling algorithm to handle loops with recurrences and arbitrary acyclic control flow in the loop body, attention was paid to coping with the very complex resource usage patterns that were the result of compromises forced by pragmatic implementation considerations. Complex recurrences and resource usage patterns make it unlikely that a one-pass scheduling algorithm, such as list scheduling, will be able to succeed in finding a near-optimal modulo schedule, even when one exists, and performing an exhaustive search was deemed impractical. Instead, an iterative scheduling algorithm was used that could unschedule and reschedule operations. This iterative algorithm is guided by heuristics based on dynamic slack-based priorities. The initial attempt is to schedule the loop with the II equal to the MII. If unsuccessful, the II is incremented until a modulo schedule is achieved.

Loops with arbitrary acyclic control flow in the loop body are dealt with by performing IF-conversion [Allen et al. 1983] to replace all branching by predicated (guarded) operations. This transformation, which assumes the hardware capability of predicated execution [Rau 1988; Rau et al. 1989], yields a loop with a single basic block that is then amenable to the modulo scheduling algorithm [Dehnert et al. 1989]. A disadvantage of predicated modulo scheduling is that the ResMII must be computed as if all the operations in the body of the loop are executed each iteration, whereas, in reality, only those along one of

the control flow paths are actually executed. As a result, during execution, some fraction of the operations in an instruction are wasted. Likewise, the RecMII is determined by the worst-case dependence chain across all paths through the loop body. Both contribute to a degree of suboptimality that depends on the structure of the loop.

Assuming the existence of hardware to support both predicated execution and speculative execution [Mahlke, Chen et al. 1992], Cydrome's modulo scheduling algorithm has been further extended to handle WHILE loops and loops with conditional exits [Tirumalai et al. 1990]. The problem that such loops pose is that it is not known until late in one iteration whether the next one should be started. This eliminates much of the overlap between successive iterations. The solution is to start iterations speculatively, in effect, by moving operations from one iteration into a prior one. The hardware support makes it possible to avoid observing exceptions from operations that should not have been executed, without overlooking exceptions from nonspeculative operations.

Independently of the Cydrome work, Hsu [1986] proposed a modulo scheduling algorithm for single basic block loops with general recurrences that recognizes each strongly connected class (SCC) of nodes in the cyclic dependence graph as a distinct entity. Once the nodes in all the SCCs have been jointly scheduled at the smallest possible II using a combinatorial search, the nodes in a given SCC may only be rescheduled as a unit and at a time that is displayed by a multiple of II. This rescheduling is performed to enable the remaining nodes that are not part of any SCC to be inserted into the schedule. Hsu also described an II extension technique that can be used to take a legal modulo schedule for one iteration and trivially convert it into a legal modulo schedule for a larger II without performing any scheduling. This works with simple resource usage patterns. With complex patterns a certain amount of rescheduling would be required, but less than starting from scratch.

Lam's algorithm, too, utilizes the SCC structure but list schedules each SCC separately, ignoring the inter-iteration dependences [Lam 1987, 1988]. Thereafter, an SCC is treated as a single pseudo-operation with a complex resource usage pattern, employing the technique of hierarchical reduction proposed by Wood [1979]. After this hierarchical reduction has been performed, the dependence graph of the computation is acyclic and can be scheduled using modulo scheduling. With an initial value equal to the MII, the II is iteratively increased until a legal modulo schedule is obtained. By determining and fixing the schedule of each SCC in isolation, Lam's algorithm can result in SCCs that cannot be scheduled together at the minimum achievable II.

On the other hand, the application of hierarchical reduction enables Lam's algorithm to cope with loop bodies containing structured control flow graphs without any special hardware support such as predicated execution. Just as with the SCCs, structured constructs such as IF-THEN-ELSE are list scheduled and treated as atomic objects. Each leg of the IF-THEN-ELSE is list scheduled separately and the union of the resource usages represents that of the reduced IF-THEN-ELSE construct. This permits loops with structured flow of control to be modulo scheduled. After modulo scheduling, the hierarchically reduced IF-THEN-ELSE pseudo-operations must be expanded. Each portion of the schedule in which m IF-THEN-ELSE pseudo-operations are active must be expanded into 2^m control flow paths with the appropriate branching and merging between the paths.

Since Lam takes the union of the resource usages in a conditional construct while predicated modulo scheduling takes the sum of the usages, the former approach should yield the smaller MII. However, since Lam separately list schedules each leg of the conditional creating pseudo-operations with complex resource usage patterns, the II that she actually achieves should deviate from the MII to a greater extent. Warter and others have implemented both techniques and have observed that, on the average, Lam's approach results in smaller MIIs but larger IIs [Warter et al. 1992]. This effect increases for processors with higher issue rates. Warter and others go on to combine the best of both approaches in their enhanced modulo scheduling algorithm. They derive the modulo schedule as if predicated execution were available, except that two operations from the same iteration are allowed to be scheduled on the same resource at the same time if their predicates are mutually exclusive, that is, they cannot both be true. This is equivalent to taking the union of the resource usages. Furthermore, it is applicable to arbitrary, possibly unstructured, acyclic flow graphs in the loop body. After modulo scheduling, the control flow graph is regenerated much as in Lam's approach. Enhanced modulo scheduling results in MIIs that are as small as for hierarchical reduction, but as with predicated modulo scheduling, the achieved II is rarely more than the MII.

Yet another independent stream of activity has been the work of Su and his colleagues [Su et al. 1984; Su et al. 1986]. When limited to loops with a single basic block, Su's URPR algorithm is an ad hoc approximation to modulo scheduling and is susceptible to significant suboptimality when confronted by nonunit latencies and complex resource usage patterns. The essence of the URPR algorithm is to unroll and schedule successive iterations until the first iteration has completed. Next the smallest contiguous set of instructions, which contain at least one instance of each operation in the original loop, is identified. After deleting multiple instances of all operations, this constitutes the software pipelined schedule. This deletion process introduced "holes" in the schedule and the attendant suboptimality. Also, for nonunit latencies, there is no guarantee that the schedule, as constructed, can loop back on itself without padding the schedule out with no-op cycles. This introduces further degradation.

Subsequently, Su extended URPR to the GURPR* algorithm for software pipelining loops containing control flow [Su et al. 1987; Su and Wang 1991a, 1991b]. GURPR* consists of first performing global scheduling on the body of the loop and then using a URPR-like procedure, as if each iteration was IF-converted, to derive the repeating pattern. Finally, as with enhanced modulo scheduling, a control flow graph is regenerated. The shortcomings of URPR are inherited by GURPR*. Warter and others, who have implemented GURPR* within the IMPACT compiler, have found that GURPR* performs significantly worse than hierarchical reduction, predicated modulo scheduling, or enhanced module scheduling [Warter et al. 1992].

The idea proposed by Fisher and others of incrementally unrolling and scheduling a loop until the pattern repeats [Fisher et al. 1981] was pursued by Nicolau and his coworkers, assuming unbounded resources, initially for single basic block loops [Aiken and Nicolau 1988b] and then, under the title of perfect pipelining, for multiple basic block loops [Aiken and Nicolau 1988a; Nicolau and Potasman 1990]. The latter was subsequently extended tc yield a more realistic algorithm assuming finite resources [Aiken and Nicolau 1991]. For single basic block loops the incremental unrolling yields a growing linear trace, the

expansion of which is terminated once a repeating pattern is observed. In practice there are complications since the various SCCs might proceed at different rates, never yielding a repeating pattern. For multiple basic block loops, the unrolling yields a growing tree of schedules, each leaf of which spawns two further leaves when a conditional branch is scheduled. A new leaf is not spawned if the (infinite) tree, of which it would be the root, is identical to another (infinite) tree (of which it might be the leaf) whose root has already been generated.

This approach addresses a shortcoming of all the previously mentioned approaches to software pipelining multiple basic block loops. In general, both RecMII and ResMII are dependent upon the specific control flow path followed in each iteration. Whereas the previous approaches had to use a single, constant, conservative value for each one of these lower bounds, the unrolling approach is able to take advantage of the branch history of previous iterations in deriving the schedule for the current one. However, there are some drawbacks as well. One handicap that such unrolling schemes have is a lack of control over the greediness of the process of initiating iterations. Starting successive iterations as soon as possible, rather than at a measured rate that is in balance with the completion rate, cannot reduce the average initiation interval but can increase the time to enter the repeating pattern and the length of the repeating pattern. Both contribute to longer compilation times and larger code size. A second problem with unrolling schemes lies in their implementation; recognizing that one has arrived at a previously visited state, to which one can wrap back instead of further expanding the search tree, is quite complicated, especially in the context of finite resources, nonunit latencies, and complex resource usage patterns.

The cyclic scheduling algorithm developed by the IBM VLIW research project [Ebcioglu and Nakatani 1989; Gasperoni 1989; Moon and Ebcioglu 1992; Nakatani and Ebcioglu 1990] might represent a good compromise between the ideal and the practical. Stripped to the essentials, this algorithm applies a cut set, termed a *fence*, to the cyclic graph, which yields an acyclic graph. This reduces the problem to that of scheduling a general, acyclic graph—a simpler problem. Once this is done the fence is moved and the acyclic scheduling is repeated. As this process is repeated, all the cycles in the control flow graph acquire increasingly tight schedules. The acyclic scheduling algorithm used by Ebcioglu and others is a resource-constrained version of percolation scheduling [Ebcioglu and Nicolau 1989; Moon and Ebcioglu 1992].

Software pipelining was also implemented in the compiler for the product line marketed by another minisupercomputer company, Culler Scientific. Unfortunately, we do not believe that any publication describing their implementation of software pipelining exists. Quite recently, software pipelining has been implemented in the compilers for HP's PA-RISC line of computers [Ramakrishnan 1992].

3.2.1.4. Scheduling for RISC and Superscalar Processors. Seemingly conventional scalar processors can sometimes benefit from scheduling techniques. This is due to small amounts of ILP in the form of, for instance, branch delay slots and shallow pipelines. Scheduling for such processors, whether RISC or CISC, has generally been less ambitious and more ad hoc than that for VLIW processors [Auslander and Hopkins 1982; Gross and Hennessy 1982; Hennessy and Gross 1983; Hsu 1987; McFarling and Hennessy 1986]. This was a direct consequence of the lack of parallelism in those machines and the corresponding lack of opportunity for the scheduler to make a big difference. Furthermore, the limited number of registers in those architectures made the use of aggressive scheduling rather unattractive. As a result, scheduling was viewed as rather peripheral to the compilation process, in contrast to the central position it occupied for VLIW processors and, to a lesser extent, for more highly pipelined processors [Rymarczyk 1982; Sites 1978; Weiss and Smith 1987]. Now, with superscalar processors growing in popularity, the importance of scheduling, as a core part of the compiler, is better appreciated and a good deal of activity has begun in this area [Bernstein and Rodeh 1991; Bernstein et al. 1991; Golumbic and Rainish 1990; Jain 1991; Smotherman et al. 1991], unfortunately, sometimes unaware of the large body of literature that already exists.

3.2.2. Register Allocation. In conventional, sequential processors, instruction scheduling is not an issue. The program's execution time is barely affected by the order of the instruction, only by the number of instructions. Accordingly, the emphasis of the code generator is on generating the minimum number of instructions and using as few registers as possible [Aho and Johnson 1976; Aho et al. 1977a, 1977b; Bruno and Sethi 1976; Sethi 1975; Sethi and Ullman 1970]. However, in the context of pipelined or multiple-issue processors, where instruction scheduling is important, the issue of the phase-ordering between it and register allocation has been a topic of much debate. There are advocates both for performing register allocation before scheduling [Gibbons and Muchnick 1986; Hennessy and Gross 1983; Jain 1991] as well as for performing it after scheduling [Auslander and Hopkins 1982; Chang, Lavery, and Hwu 1991; Goodman and Hsu 1988; Warren 1990]. Each phase-ordering has its advantages and neither one is completely satisfactory.

The most important argument in favor of performing register allocation first is that whereas a better schedule may be desirable, code that requires more registers than are available is just unacceptable. Clearly, achieving a successful register allocation must supersede the objective of constructing a better schedule. The drawback of performing scheduling first, oblivious of the register allocation, is that shorter schedules tend to yield greater register pressure. If a viable allocation cannot be found, spill code must be inserted. At this point, in the case of a statically scheduled processor, the schedule just constructed may no longer be correct. Even if it is, it may be far from the best one possible, for either a VLIW or superscalar machine, since the schedule was built without the spill code in mind. In machines whose load latency is far greater than that of the other operations, the time penalty of the spill code may far exceed the benefits of the better schedule obtained by performing scheduling first.

Historically, the merit of performing register allocation first was that processors had little instruction-level parallelism and few registers, so whereas there was much to be lost by a poor register allocation, there was little to be gained by good scheduling. It was customary, therefore, to perform register allocation first, for instance using graph coloring [Chaitin 1982; Chow and Hennessy 1984, 1990] followed by a postpass scheduling step that considered individual basic blocks [Gibbons and Muchnick 1986; Hennessy and Gross 1983].

From the viewpoint of instruction-level parallel machines, the major problem with performing register allocation first is that it introduces antidependences and output dependences that can constrain parallelism and the ability to construct a good schedule. To some extent

this is inevitable; the theoretically optimal combination of schedule and allocation might contain additional arcs due to the allocation. The real concern is that, when allocation is done first, an excessive number of ill-advised and unnecessary arcs might be introduced due to the insensitivity of the register allocator to the scheduling task. On pipelined machines, whose cache access time is as short as or shorter than the functional unit latencies, the benefits of a schedule unconstrained by register allocation may outweigh the penalties of the resulting spill code.

Scheduling prior to register allocation, known as prepass scheduling, was used in the PL.8 compiler [Auslander and Hopkins 1982]. In evolving this compiler to become the compiler for the superscalar IBM RISC System/6000, the suboptimality of inserting spill code after the creation of the schedule became clear and a second, postpass scheduling step was added after the register allocation [Warren 1990]. During the postpass the scheduler honors all the dependences caused by the register allocation, which in turn was aware of the preferred instruction schedule provided by the prepass scheduler. The IMPACT project at the University of Illinois has demonstrated the effectiveness of this strategy for multiple-issue processors [Chang, Lavery, and Hwu 1991]. Instead of employing the graph coloring paradigm, Hendren and others make use of the richer information present in interval graphs, which are a direct temporal representation of the span of the lifetimes [Hendren et al. 1992]. This assumes that the schedule or, at least, the instruction order has already been determined and that a postpass scheduling step will follow.

Irrespective of which one goes first, a shortcoming of all strategies discussed so far is that the first phase makes its decisions with no consideration of their impact on the subsequent phase. Goodman and Hsu [1988] have addressed this problem by developing two algorithms—one, a scheduler that attempts to keep the register pressure below a limit provided to it, and the second, a register allocation algorithm that is sensitive to its effect on the critical path length of the DAG and thus to the effect on the eventual schedule.

For any piece of code on a given processor, there is some optimal schedule for which register allocation is possible. Scheduling twice, once before and then after register allocation, is an approximation of achieving this ideal. Simultaneous scheduling and register allocation is another strategy for attempting to find a near-optimal schedule and register allocation. Simultaneous scheduling and register allocation is currently understood only in the context of acyclic code, specifically, a single basic block or a linear trace of basic blocks. The essence of the idea is that each time an operation is scheduled, an available register is allocated to hold the result. Also, if this operation constitutes the last use of the contents of one of the source registers, that register is made available once again for subsequent allocation. When no register is available to receive the result of the operation being scheduled, a register must be spilled. The register holding the datum whose use is furthest away in the future is spilled. This approach was used in the FPS-164 compiler at the level of individual basic blocks [Touzeau 1984] as well as across entire traces [Ellis 1985; Freudenberger and Ruttenberg 1992; Lowney et al. 1993]. An important concept developed by the ELI project at Yale and by Multiflow was that of performing hierarchical, profile-driven, integrated global scheduling and register allocation. Traces are picked in decreasing order of frequency and integrated scheduling and allocation are performed on each. The scheduling and allocation decisions made for traces that have been processed form constraints on the corresponding decisions for the remaining code. This is a far more systematic approach than other ad hoc, priority-based schemes with the same objective. A syntax-based hierarchical approach to global register allocation has been suggested by Callahan and Koblenz [1991].

If a loop is unrolled some number of times and then treated as a linear trace of basic blocks [Fisher et al. 1981], simultaneous trace scheduling and register allocation can be accomplished, but with some loss of performance due to the emptying of pipelines across the back edge. In the case of modulo scheduling, which avoids this performance penalty, no approach has yet been advanced for simultaneous register allocation. Since doing register allocation in advance is unacceptably constraining on the schedule, it must be performed following modulo scheduling. A unique situation encountered with modulo scheduled loops is that the lifetimes are often much longer than the initiation interval. Normally, this would result in a value being overwritten before its last use has occurred. One solution is to unroll the kernel of a modulo scheduled loop a sufficient number of times to ensure that no lifetime is longer than the length of the replicated kernel [Lam 1987, 1988]. This is known as *modulo variable expansion*. In addition to techniques such as graph coloring, the heuristics proposed by Hendren and others [1992] and by Rau and others [1992] may be applied after modulo variable expansion. The other solution for register allocation is to assume the dynamic register renaming provided by the rotating register capability of the Cydra 5. The entity that the register allocator works with are vector lifetimes, that is, the entire sequence of (scalar) lifetimes defined by a particular operation over the execution of the loop [Dehnert and Towle 1993; Dehnert et al. 1989; Rau et al. 1992]. Lower bounds on the number of registers needed for a modulo scheduled loop have been developed by Mangione-Smith and others [1992]. The strategy for recovering from a situation, in which no allocation can be found for the software pipelined loop, is not well understood. Some options have been outlined [Rau et al. 1992], but their detailed implementation, effectiveness, and relative merits have as yet to be investigated.

3.2.3. Other ILP Compiler Topics. Although scheduling and register allocation are at the heart of ILP compilation, a number of other analyses, optimizations, and transformations are crucial to the generation of high-quality code. Currently, schedulers treat a procedure call as a barrier to code motion. Thus, in-lining of intrinsics and user procedures is very important in the high frequency portions of the program [Dehnert and Towle 1993; Linn 1988; Lowney et al. 1993].

Certain loop-oriented analyses and optimizations are specific to modulo scheduling. IF-conversion and the appropriate placement of predicate-setting operations are needed to modulo schedule loops with control flow [Allen et al. 1983; Dehnert and Towle 1993; Dehnert et al. 1989; Park and Schlansker 1991]. The elimination of subscripted loads and stores that are redundant across multiple iterations of a loop can have a significant effect upon both the ResMII and the RecMII [Callahan et al. 1990; Dehnert and Towle 1993; Rau 1992]. This is important for trace scheduling unrolled loops as well [Lowney et al. 1993]. Recurrence back-substitution, and other transformations that reduce the RecMII have a major effect on the performance of all software pipelined loops [Dehnert and Towle 1993]. Most of these transformations and analyses are facilitated by the dynamic single-assignment representation for inner loops [Dehnert and Towle 1993; Rau 1992].

On machines with multiple, identical clusters, such as the Multiflow Trace machines, it is necessary to decide which part of the computation will go on each cluster. This is a

nontrivial task; whereas increased parallelism argues in favor of spreading the computation over the clusters, this also introduces intercluster move operations into the computation, whose latency can degrade performance if the partitioning of the computation across clusters is not done carefully. An algorithm for performing this partitioning was developed by Ellis [1986] and was incorporated into the Multiflow compiler [Lowney et al. 1993].

An issue of central importance to all ILP compilation is the disambiguation of memory references, that is, deciding whether two memory references definitely are to the same memory location or definitely are not. Known as dependence analysis, this has become a very well developed topic in the area of vector computing over the past twenty years [Zima and Chapman 1990]. For vector computers the compiler is attempting to prove that two references in different iterations are *not* to the same location. No benefit is derived if it is determined that they *are* to the same location since such loops cannot be vectorized. Consequently, the nature of the analysis, especially in the context of loops containing conditional branching, has been approximate. This is a shortcoming from the point of view of ILP processors that can benefit both if the two references are or are not to the same location. A more precise analysis than dependence analysis, involving data flow analysis, is required. Also, with ILP processors, memory disambiguation is important outside of loops as well as within them. Memory disambiguation within traces was studied in the ELI project [Ellis 1985; Nicolau 1984] and was implemented in the Multiflow compiler [Lowney et al. 1993]. Memory disambiguation, in the context of innermost loops, was implemented in the Cydra 5 compiler [Dehnert and Towle 1993; Rau 1992] and was studied by Callahan and Koblenz [1991].

4. Available ILP

4.1. Limit Studies and Their Shortcomings

Many experimenters have attempted to measure the maximum parallelism available in programs. The goal of such limit studies is to

> *throw away all considerations of hardware and compiler practicality and measure the greatest possible amount of ILP inherent in a program.*

Limit studies are simple enough to describe: Take an execution trace of the program, and build a data precedence graph on the operations, eliminating false antidependences caused by the write-after-read usage of a register or other piece of hardware storage. The length in cycles of the serial execution of the trace gives the serial execution time on hardware with the given latencies. The length in cycles of the critical path though the data dependence graph gives the shortest possible execution time. The quotient of these two is the available speedup. (In practice, an execution trace is not always gathered. Instead, the executed stream is processed as the code runs, greatly reducing the computation or storage required, or both.)

These are indeed maximum parallelism measures in some sense, but they have a critical shortcoming that causes them to miss accomplishing their stated goal; they do not consider transformations that a compiler might make to enhance ILP. Although we mostly mean transformations of a yet-unknown nature that researchers may develop in the future, even current state-of-the-art transformations are rarely reflected in limit studies. Thus we have had, in recent years, the anomalies of researchers stating an "upper limit" on available parallelism in programs that is lower than what has already been accomplished with those same programs, or of new results that show the maximum available parallelism to be significantly higher than it was a few years ago, before a new set of code transformations was considered.

There is a somewhat fatuous argument that demonstrates just how imprecise limit studies must be; recalling that infinite hardware is available, we can replace computations in the code with table lookups. In each case we will replace a longer—perhaps very long—computation with one that takes a single step. While this is obviously impractical for most computations with operands that span the (finite, but large) range of integers or floating point numbers representable on a system, it is only impractical in the very sense in which practicality is to be discarded in limit studies. And even on practicality grounds, one cannot dismiss this argument completely; in a sense it really does capture what is wrong with these experiments. There are many instances of transformations, some done by hand, others automatically, that reduce to this concept. Arithmetic and transcendental functions are often sped up significantly by the carefully selected use of table lookups at critical parts of the computation. Modern compilers can often replace a nested set of IF-THEN tests with a single lookup in which hardware does an indirect jump through a lookup table. Limit studies have no way of capturing these transformations, the effect of which could be a large improvement in available ILP.

Even in current practice the effect of ignoring sophisticated compiling is extreme. Transformations such as tree height reduction, loop conditioning, loop exchange, and so forth can have a huge effect on the parallelism available in code. A greater unknown is the research future of data structure selection to improve ILP. A simple example can show this effect. The following code finds the maximum element of a linked list of data:

```
this-ptr = head-ptr;
max-so-far = most-neg-number;
while this-ptr {
        if this-ptr.data > max-so-far
             then max-so-far = this-ptr.data;
        this-ptr = this-ptr.next }
```

From simple observation the list of elements chained from `head-ptr` cannot be circular. If the compiler had judged it worthwhile, it could have stored these elements in an array and done the comparisons pairwise, in parallel, without having to chase the pointers linearly. This example is not as farfetched as it might seem. Vectorization took 20 years to go from the ability to recognize the simplest loop to the sophisticated vectorizers we have today. There has been virtually no work done on compiler transformations to enhance ILP.

Limit studies, then, are in some sense finding the maximum parallelism available, but in other ways are finding the minimum. In these senses they find the maximum parallelism:

- Disambiguation can be done perfectly, well beyond what is practical.
- There are infinitely many functional units available.
- There are infinitely many registers available.
- Rejoins can be completely unwound.

In other senses, they represent a minimum, or an existence proof that at least a certain amount of parallelism exists, since potentially important processes have been left out:

- Compiler transformations to enhance ILP have not been done.
- Intermediate code generation techniques that boost ILP have not been done.

Perhaps it is more accurate to say that a limit study shows that the maximum parallelism available, in the absence of practicality considerations, is *at least* the amount measured.

4.1.1. Early Experiments. The very first ILP limit studies demonstrated the effect we wrote of above: The experimenters' view of the techniques by which one could find parallelism was limited to the current state of the art, and the experimenters missed a technique that is now known to provide most of the available ILP, the motion of operations between basic blocks of code. Experiments done by Tjaden and Flynn [1970] and by Foster and Riseman [1972] (and, anecdotally, elsewhere) found that there was only a small amount (about a factor of two to three) of improvement due to ILP available in real programs. This was dubbed the *Flynn bottleneck*. By all accounts, these pessimistic and, in a sense, erroneous experiments had a tremendous dampening effect on the progress of ILP research. The experiments were only erroneous in the sense of missing improvements; certainly they did correctly what they said they did.

Interestingly, one of the research teams doing these experiments saw that under the hypothesis of free and infinite hardware, one would not necessarily have to stop finding ILP at basic block boundaries. In a companion paper to the one mentioned above, Riseman and Foster [1972] put forward a hardware-intensive solution to the problem of doing operations speculatively: They measured what would happen if one used duplicate hardware at conditional jumps, and disregarded the one that went in the wrong direction. They found a far larger amount of parallelism: Indeed, they found more than an order of magnitude more than they could when branches were a barrier. Some of the programs they measured could achieve arbitrarily large amounts of parallelism, depending only on data set size. But in an otherwise insightful and visionary piece of work, the researchers lost sight of the fact that they were doing a limit study, and in their tone and abstract emphasized how impractical it would be to implement the hardware scheme they had suggested. (They found that to get a factor-of-ten ILP speedup, one had to be prepared to cope with 16 unresolved branches at the worst point of a typical program. Their scheme would require, then, 2^{16} sets of hardware to do so. Today, as described in most of the papers in this issue, we try to get much of the benefit of the same parallelism without the hardware cost by doing code motions that move operations between blocks and having the code generator make sure that the correct computation is ultimately done once the branches settle.)

4.1.2. Contemporary Experiments. We know of no other ILP limit studies published between then and the 1980s. In 1981 Nicolau and Fisher [1981, 1984] used some of the apparatuses being developed for the Yale Bulldog compiler to repeat the experiment done by Riseman and Foster, and found virtually the same results.

In the late 1980s architects began to look at superscalar microprocessors and again started a series of limit studies. Interestingly, the most notorious of these [Jouppi and Wall 1989] again neglected the possibility of code motions between blocks. Unsurprisingly, the Flynn bottleneck appeared again, and only the factor of 2–3 parallelism found earlier was found. Two years later Wall [1991] did the most thorough limit study to date and accounted for speculative execution, memory disambiguation, and other factors. He built an elaborate model and published available ILP speedup under a great many scenarios, yielding a wealth of valuable data but no simple answers. The various scenarios allow one to try to bracket what really might be practical in the near future, but are subject to quite a bit of interpretation. In examining the various scenarios presented, we find that settings that a sophisticated compiler might approach during the coming decade could yield speedups ranging from 7 to 60 on the sample programs, which are taken from the SPEC suite and other standard benchmarks. (It is worth noting that Wall himself is much more pessimistic. In the same results he sees an average ceiling of about 5, and the near impossibility of attaining even that much.) Lam and Wilson [1992] did an experiment to measure the effects of different methods of eliminating control flow barriers to parallelism. When their model agreed with Wall's, their results were similar. Butler and Patt [Butler et al. 1991] considered models with a large variety of numbers of functional units and found that with good branch prediction schemes and speculative execution, a wide range of speedup was available.

4.2. Experiments That Measure Attained Parallelism

In contrast to the limit studies, some people have built real or simulated ILP systems and have measured their speedup against real or simulated nonparallel systems. When simulated systems have been involved, they have been relatively realistic systems, or systems that the researchers have argued would abstract the essence of realistic systems in such a way that the system realities should not lower the attained parallelism. Thus the experiments represent something closer to true lower bounds on available parallelism.

Ellis [1986] used the Bulldog compiler to generate code for a hypothetical machine. His model was unrealistic in several aspects, most notably the memory system, but realistic implementations should have little difficulty exploiting the parallelism he found. Ellis measured the speedups obtained on 12 small scientific programs for both a "realistic" machine (corresponding to one under design at Yale) and an "ideal" machine, with limitless hardware and single-cycle functional units. He found speedups ranging from no speedup to 7.6 times speedup for the real model, and a range of 2.7 to 48.3 for the ideal model.

In this issue there are three papers that add to our understanding of the performance of ILP systems. The paper by Hwu and others [1993] considers the effect of a realistic compiler that uses superblock scheduling. Lowney and others [1993] and Schuette and Shen [1993] compare the performance of the Multiflow TRACE 14/300 with current microprocessors from MIPs and IBM, respectively.

Fewer studies have been done to measure the attained performance of software pipelining. Warter and others [1992] consider a set of 30 *doall* loops with branches found in the Perfect

and SPEC benchmark sets. Relative to a single-issue machine without modulo scheduling, they find a 6-time speedup on a hypothetical 4-issue machine and a 10-time speedup on a hypothetical 8-issue machine. Lee and others [1993] combined superblock scheduling and software pipelining for a machine capable of issuing up to seven operations per cycle. On a mix of loop-intensive (e.g., LINPACK) and "scalar" (e.g., Spice) codes, they found an average of one to four operations issued per cycle, with two to seven operations in flight.

5. An Introduction to This Special Issue

In this special issue of *The Journal of Supercomputing* we have attempted to capture the most significant work that took place during the 1980s in the area of instruction-level parallel processing. The intent is to document both the theory and the practice of ILP computing. Consequently, our emphasis is on projects that resulted in implementations of serious scope, since it is this reduction to practice that exposes the true merit and the real problems of ideas that sound good on paper.

During the 1980s the bulk of the advances in ILP occurred in the form of VLIW processing, and this special issue reflects it with papers on Multiflow's Trace family and on Cydrome's Cydra 5. The paper by Lowney and others [1993] provides an overview of the Trace hardware and an in-depth discussion of the compiler. The paper by Schuette and Shen [1993] reports on an evaluation performed by the authors of the TRACE 14/300 and a comparison of it to the superscalar IBM RS/6000. The Cydra 5 effort is documented by two papers: one by Beck, Yen, and Anderson [1993] on the Cydra 5 architecture and hardware implementation, and the other by Dehnert and Towle [1993] on the Cydra 5 compiler. (While reading the descriptions of these large and bulky minisupercomputers, it is worthwhile to bear in mind that they could easily fit on a single chip in the near future!) The only important superscalar product of the 1980s was Astronautics' ZS-1 minisupercomputer. Although we wanted to include a paper on it in this special issue, that did not come to pass. The paper by Hwu and others [1993] reports on IMPACT, the most thorough implementation of an ILP compiler that has occurred in academia.

Notes

1. The first machines of this type that were built in the 1960s were referred to as *look-ahead* processors. Subsequently, machines that performed out-of-order execution, while issuing multiple operations per cycle, came to be termed *superscalar* processors. Since look-ahead processors are only quantitatively different from superscalar processors, we shall drop the distinction and refer to them, too, as superscalar processors.
2. We shall consistently refer to this code generation activity as *scheduling*.

References

Acosta, R.D., Kjelstrup, J., and Torng, H.C. 1986. An instruction issuing approach to enhancing performance in multiple function unit processors. *IEEE Trans. Comps.*, C-35, 9 (Sept.): 815–828.

Adam, T.L., Chandy, K.M., and Dickson, J.R. 1974. A comparison of list schedules for parallel processing systems. *CACM*, 17, 12 (Dec.): 685–690.

Advanced Micro Devices. 1989. *Am29000 Users Manual*. Pub. no. 10620B, Advanced Micro Devices, Sunnyvale, Calif.

Agerwala, T. 1976. Microprogram optimization: A survey. *IEEE Trans. Comps.*, C-25, 10 (Oct.): 962–973.

Agerwala, T., and Cocke, J. 1987. High performance reduced instruction set processors. Tech. rept. RC12434 (#55845), IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y.

Aho, A.V., and Johnson, S.C. 1976. Optimal code generation for expression trees. *JACM*, 23 3 (July): 488–501.

Aho, A.V., Johnson, S.C., and Ullman, J.D. 1977a. Code generation for expressions with common subexpressions. *JACM*, 24, 1 (Jan.): 146–160.

Aho, A.V., Johnson, S.C., and Ullman, J.D. 1977b. Code generation for machines with multiregister operations. In *Proc., Fourth ACM Symp. on Principles of Programming Languages*, pp. 21–28.

Aiken, A., and Nicolau, A. 1988a. Optimal loop parallelization. In *Proc., SIGPLAN'88 Conf. on Programming Language Design and Implementation* (Atlanta, June), pp. 308–317.

Aiken, A., and Nicolau, A. 1988b. Perfect pipelining: A new loop parallelization technique. In *Proc., 1988 European Symp. on Programming*, Springer Verlag, New York, pp. 221–235.

Aiken, A., and Nicolau, A. 1991. A realistic resource-constrained software pipelining algorithm. In *Advances in Languages and Compilers for Parallel Processing* (A. Nicolau, D. Gelernter, T. Gross, and D. Padua, eds.), Pitman/MIT Press, London, pp. 274–290.

Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J. 1983. Conversion of control dependence to data dependence. In *Proc., Tenth Annual ACM Symp. on Principles of Programming Languages* (Jan.): pp. 177–189.

Anderson D.W., Sparacio, F.J., and Tomasulo, R.M. 1967. The System/360 Model 91: Machine philosophy and instruction handling. *IBM J. Res. and Dev.*, 11, 1 (Jan.): 8–24.

Apollo Computer. 1988. *The Series 10000 Personal Supercomputer: Inside a New Architecture*. Publication no. 002402-007 2-88, Apollo Computer, Inc., Chelmsford, Mass.

Arvind and Gostelow, K. 1982. The U-interpreter. *Computer*, 15, 2 (Feb.): 12–49.

Arvind and Kathail, V. 1981. A multiple processor dataflow machine that supports generalised procedures. In *Proc., Eighth Annual Symp. on Computer Architecture* (May): pp. 291–302.

Auslander, M., and Hopkins, M. 1982. An overview of the PL.8 compiler. In *Proc., ACM SIGPLAN Symp. on Compiler Construction* (Boston, June), pp. 22–31.

Bahr, R., Ciavaglia, S., Flahive, B., Kline, M., Mageau, P., and Nickel, D. 1991. The DN10000TX: A new high-performance PRISM processor. In *Proc., COMPCON '91*, pp. 90–95.

Baker, K.R. 1974. *Introduction to Sequencing and Scheduling*. John Wiley, New York.

Beck, G.R., Yen, D.W.L., and Anderson T.L. 1993. The Cydra 5 minisupercomputer: Architecture and implementation. *The J. Supercomputing*, 7, 1/2: 143–180.

Bell, C.G., and Newell, A. 1971. *Computer Structures: Readings and Examples*. McGraw-Hill, New York.

Bernstein, D., and Rodeh, M. 1991. Global instruction scheduling for superscalar machines. In *Proc., SIGPLAN '91 Conf. on Programming Language Design and Implementation* (June), pp. 241–255.

Bernstein, D., Cohen, D., and Krawczyk, H. 1991. Code duplication: An assist for global instruction scheduling. In *Proc., 24th Annual Internat. Symp. on Microarchitecture* (Albuquerque, N.Mex.), pp. 103–113.

Blanck, G., and Krueger, S. 1992. The SuperSPARC™ microprocessor. In *Proc., COMPCON '92*, pp. 136–141.

Bloch, E. 1959. The engineering design of the STRETCH computer. In *Proc., Eastern Joint Computer Conf.*, pp. 48–59.

Bruno, J.L., and Sethi, R. 1976. Code generation for a one-register machine. *JACM*, 23, 3 (July): 502–510.

Buchholz, W., ed. 1962. *Planning a Computer System: Project Stretch*. McGraw-Hill, New York.

Butler, M., Yeh, T., Patt., Y., Alsup, M., Scales, H., and Shebanow, M. 1991. Single instruction stream parallelism is greater than two. In *Proc., Eighteenth Annual Internat. Symp. on Computer Architecture* (Toronto), pp. 276–286.

Callahan, D., and Koblenz, B. 1991. Register allocation via hierarchical graph coloring. In *Proc., SIGPLAN '91 Conf. on Programming Language Design and Implementation* (Toronto, June), pp. 192–203.

Callahan, D., Carr, S., and Kennedy, K. 1990. Improving register allocation for subscripted variables. In *Proc., ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, (White Plains, N.Y., June), pp. 53–65.

Carpenter, B.E., and Doran, R.W., eds. 1986. *A.M. Turing's ACE Report of 1946 and Other Papers*. MIT Press, Cambridge, Mass.

Chaitin, G.J. 1982. Register allocation and spilling via graph coloring. In *Proc., ACM SIGPLAN Symp. on Compiler Construction* (Boston, June), pp. 98–105.

Chang, P.P., and Hwu, W.W. 1988. Trace selection for compiling large C application programs to microcode. In *Proc., 21st Annual Workshop on Microprogramming and Microarchitectures* (San Diego, Nov.), pp. 21–29.

Chang, P.P., and Hwu, W.W. 1992. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22, 5 (May): 349–376.

Chang, P.P., Lavery, D.M., and Hwu, W.W. 1991. The importance of prepass code scheduling for superscalar and superpipelined processors. Tech. Rept. no. CRHC-91-18, Center for Reliable and High-Performance Computing, Univ. of Ill, Urbana-Champaign, Ill.

Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., and Hwu, W.W. 1991. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proc., 18th Annual Internat. Symp. on Computer Architecture* (Toronto, May), pp. 266–275.

Charlesworth, A.E. 1981. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *Computer*, 14, 9: 18–27.

Chen, T.C. 1971. Parallelism, pipelining, and computer efficiency. *Computer Design*, 10, 1 (Jan.): 69–74.

Chen, T.C. 1975. Overlap and pipeline processing. In *Introduction to Computer Architecture* (H.S. Stone, ed.), Science Research Associates, Chicago, pp. 375–431.

Chow, F., and Hennessy, J. 1984. Register allocation by priority-based coloring. In *Proc., ACM SIGPLAN Symp. on Compiler Construction* (Montreal, June), pp. 222–232.

Chow, F.C., and Hennessy, J.L. 1990. The priority-based coloring approach to register allocation. *ACM Trans. Programming Languages and Systems*, 12 (Oct.): 501–536.

Coffman, J.R., ed. 1976. *Computer and Job-Shop Scheduling Theory.* John Wiley, New York.

Coffman, E.G., and Graham, R.L. 1972. Optimal scheduling for two processor systems. *Acta Informatica*, 1, 3: 200–213.

Cohen, D. 1978. A methodology for programming a pipeline array processor. In *Proc., 11th Annual Microprogramming Workshop* (Asilomar, Calif., Nov.), pp. 82–89.

Colwell, R.P., Nix, R.P., O'Donnell, J.J., Papworth, D.B., and Rodman, P.K. 1988. A VLIW architecture for a trace scheduling compiler. *IEEE Trans. Comps.*, C-37, 8 (Aug.): 967–979.

Colwell, R.P., Hall, W.E., Joshi, C.S., Papworth, D.B., Rodman, P.K., and Tornes, J.E. 1990. Architecture and implementation of a VLIW supercomputer. In *Proc., Supercomputing '90* (Nov.), pp. 910–919.

Cotten, L.W. 1965. Circuit implementation of high-speed pipeline systems. In *Proc., AFIPS Fall Joint Computing Conf.*, pp. 489–504.

Cotten, L.W. 1969. Maximum-rate pipeline systems. In *Proc., AFIPS Spring Joint Computing Conf.*, 581–586.

Danelutto, M., and Vanneschi, M. 1990. VLIW in-the-large: A model for fine grain parallelism exploitation of distributed memory multiprocessors. In *Proc., 23rd Annual Workshop on Microprogramming and Microarchitecture* (Nov.), pp. 7–16.

Dasgupta, S., and Tartar, J. 1976. The identification of maximal parallelism in straight-line microprograms. *IEEE Trans. Comps.*, C-25, 10 (Oct.): 986–991.

Davidson, E.S. 1971. The design and control of pipelined function generators. In *Proc., 1971 Internat. IEEE Conf. on Systems, Networks, and Computers* (Oaxtepec, Mexico, Jan.), pp. 19–21.

Davidson, E.S. 1974. Scheduling for pipelined processors. In *Proc., 7th Hawaii Conf. on Systems Sciences*, pp. 58–60.

Davidson, S., Landskov, D., Shriver, B.D., and Mallett, P.W. 1981. Some experiments in local microcode compaction for horizontal machines. *IEEE Trans. Comps.*, C-30, 7: 460–477.

Davidson, E.S., Shar, L.E., Thomas, A.T., and Patel, J.H. 1975. Effective control for pipelined computers. In *Proc., COMPCON '90* (San Francisco, Feb.), pp. 181–184.

Dehnert, J.C., and Towle, R.A. 1993. Compiling for the Cydra 5. *The J. Supercomputing*, 7, 1/2: 181–227.

Dehnert, J.C., Hsu, P.Y.-T., and Bratt, J.P. 1989. Overlapped loop support in the Cydra 5. In *Proc., Third Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Apr.), pp. 26–38.

DeLano, E., Walker, W., Yetter, J., and Forsyth, M. 1992. A high speed superscalar PA-RISC processor. In *Proc., COMPCON '92* (Feb.), pp. 116–121.

DeWitt, D.J. 1975. A control word model for detecting conflicts between microprograms. In *Proc., 8th Annual Workshop on Microprogramming* (Chicago, Sept.), pp. 6–12.

Diefendorff, K., and Allen, M. 1992. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12, 2 (Apr.): 40–63.

Dongarra, J.J. 1986, A survey of high performance computers. In *Proc., COMPCON '86* (Mar.), pp. 8–11.

Dwyer, H., and Torng, H.C. 1992. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. In *Proc., 25th Annual Internat. Symp. on Microarchitecture* (Portland, Ore., Dec.), pp. 272–281.

Ebcioglu, K. 1988. Some design ideas for a VLIW architecture for sequential-natured software. In *Parallel Processing (Proc., IFIP WG 10.3 Working Conf. on Parallel Processing, Pisa, Italy)* (M. Cosnard, M.H. Barton, and M. Vanneschi, eds.), North-Holland, pp. 3–21.

Ebcioglu, K., and Nakatani, T. 1989. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing* (D. Gelernter, A. Nicolau, and D. Padua, eds.), Pitman/MIT Press, London, pp. 213–229.

Ebcioglu, K., and Nicolau, A. 1989. A *global* resource-constrained parallelization technique. In *Proc., 3rd Internat. Conf. on Supercomputing* (Crete, Greece, June), pp. 154–163.

Eckert, J.P., Chu, J.C., Tonik, A.B., and Schmitt, W.F. 1959. Design of UNIVAC-LARC System: I. In *Proc., Eastern Joint Computer Conf.*, pp. 59–65.

Ellis, J.R. 1986. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, Mass.

Fawcett, B.K. 1975. Maximal clocking rates for pipelined digital systems. M.S. thesis, Univ. of Ill., Urbana-Champaign, Ill.

Fernandez, E.B., and Bussel, B. 1973. Bounds on the number of processors and time for multiprocessor optimal schedule. *IEEE Trans. Comps.*, C-22, 8 (Aug.): 745–751.

Fisher, J.A. 1979. The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources, Ph.D. thesis, New York Univ., New York.

Fisher, J.A. 1980. 2^N-way jump microinstruction hardware and an effective instruction binding method. In *Proc., 13th Annual Workshop on Microprogramming* (Colorado Springs, Colo., Nov.), pp. 64–75.

Fisher, J.A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comps.*, C-30, 7 (July): 478–490.

Fisher, J.A. 1983. Very long instruction word architectures and the ELI-512. In *Proc., Tenth Annual Internat. Symp. on Computer Architecture* (Stockholm, June), pp. 140–150.

Fisher, J.A. 1992. Trace Scheduling-2, an extension of trace scheduling. Tech. rept., Hewlett-Packard Laboratories.

Fisher, J.A., and Freudenberger, S.M. 1992. Predicting conditional jump directions from previous runs of a program. In *Proc., Fifth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Oct.), pp. 85–95.

Fisher, J.A., Landskov, D., and Shriver, B.D. 1981. Microcode compaction: Looking backward and looking forward. In *Proc., 1981 Nat. Computer Conf.*, pp. 95–102.

Fisher, J.A., Ellis, J.R., Ruttenberg, J.C., and Nicolau, A. 1984. Parallel processing: A smart compiler and a dumb machine. In *Proc., ACM SIGPLAN '84 Symp. on Compiler Construction* (Montreal, June), pp. 37–47.

Floating Point Systems. 1979. *FPS AP-120B Processor Handbook.* Floating Point Systems, Inc., Beaverton, Ore.

Foster, C.C., and Riseman, E.M. 1972. Percolation of code to enhance parallel dispatching and execution. *IEEE Trans. Comps.*, C-21, 12 (Dec.): 1411–1415.

Franklin, M., and Sohi, G.S. 1992. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proc. 19th Annual International Symp. on Computer Architecture* (Gold Coast, Australia, May), pp. 58–67.

Freudenberger, S.M., and Ruttenberg, J.C. 1992. Phase ordering of register allocation and instruction scheduling. In *Code Generation—Concepts, Tools, Techniques: Proc., Internat. Workshop on Code Generation*, May 1991 (R. Giegerich, and S.L. Graham, eds.), Springer-Verlag, London, pp. 146–172.

Gasperoni, F. 1989. Compilation techniques for VLIW architectures. Tech. rept. RC 14915, IBM Research Div., T.J. Watson Research Center, Yorktown Heights, N.Y.

Gibbons, P.B., and Muchnick, S.S. 1986. Efficient instruction scheduling for a pipelined architecture. In *Proc., ACM SIGPLAN '86 Symp. on Compiler Construction* (Palo Alto, Calif., July), pp. 11–16.

Golumbic, M.C., and Rainish, V. 1990. Instruction schedulig beyond basic blocks. *IBM J. Res. and Dev.*, 34, 1 (Jan.): 93–97.

Gonzalez, M.J. 1977. Deterministic processor scheduling. *ACM Computer Surveys*, 9, 3 (Sept.): 173–204.

Goodman, J.R., and Hsu, W.-C. 1988. Code scheduling and register allocation in large basic blocks. In *Proc., 1988 Internat. Conf. on Supercomputing* (St. Malo, France, July), pp. 442–452.

Grishman, R., and Su, B. 1983. A preliminary evaluation of trace scheduling for global microcode compaction. *IEEE Trans. Comps.*, C-32, 12 (Dec.): 1191–1194.

Gross, T.R., and Hennessy, J.L. 1982. Optimizing delayed branches. In *Proc., 15th Annual Workshop on Microprogramming* (Oct.), pp. 114–120.

Gross, T., and Ward, M. 1990. The suppression of compensation code. In *Advances in Languages and Compilers for Parallel Computing* (A. Nicolau, D. Gelernter, T. Gross, and D. Padua, eds.), Pitman/MIT Press, London, pp. 260–273.

Gurd, J., Kirkham, C.C., and Watson, I. 1985. The Manchester prototype dataflow computer. *CACM*, 28, 1(Jan.): 34–52.

Hallin, T.G., and Flynn, M.J. 1972. Pipelining of arithmetic functions. *IEEE Trans. Comps.*, C-21, 8 (Aug.): 880–886.

Hendren, L.J., Gao, G.R., Altman, E.R., and Mukerji, C. 1992. Register allocation using cyclic interval graphs: A new approach to an old problem. ACAPS Tech. Memo 33, Advanced Computer Architecture and Program Structures Group, McGill Univ., Montreal.

Hennessy, J.L., and Gross, T. 1983. Post-pass code optimization of pipelined constraints. *ACM Trans. Programming Languages and Systems*, 5, 3 (July): 422–448.

Hennessy, J., Jouppi, N., Baskett, F., Gross, T., and Gill, J. 1982. Hardware/software tradeoffs for increased performance. In *Proc., Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar.) pp. 2–11.

Hennessy, J. Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., and Gill, J. 1982. MIPS: A microprocessor architecture. In *Proc., 15th Annual Workshop on Microprogramming* (Palo Alto, Calif., Oct.), pp. 17–22.

Hintz, R.G., and Tate, D.P. 1972. Control Data STAR-100 processor design. In *Proc., COMPCON '72* (Sept.), pp. 1–4.

Hsu, P.Y.T. 1986. Highly concurrent scalar processing. Ph.D. thesis, Univ. of Ill., Urbana-Champaign, Ill.

Hsu, P.Y.T., and Davidson, E.S. 1986. Highly concurrent scalar processing. In *Proc., Thirteenth Annual Internat. Symp. on Computer Architecture*, pp. 386–395.

Hsu, W.-C. 1987. Register allocation and code scheduling for load/store architectures. Comp. Sci. Tech. Rept. no. 722, Univ. of Wisc., Madison.

Hu, T.C. 1961. Parallel sequencing and assembly line problems. *Operations Research*, 9, 6: 841–848.

Hwu, W.W., and Chang, P.P. 1988. Exploiting parallel microprocessor microarchitectures with a compiler code generator. In *Proc., 15th Annual Internat. Symp. on Computer Architecture* (Honolulu, May), pp. 45–53.

Hwu, W.W., and Patt, Y.N. 1986. HPSm, a high performance restricted data flow architecture having minimal functionality. In *Proc., 13th Annual Internat. Symp. on Computer Architecture* (Tokyo, June), pp. 297–306.

Hwu, W.W., and Patt, Y.N. 1987. Checkpoint repair for out-of-order execution machines. *IEEE Trans. Comps.*, C-36, 12 (Dec.): 1496–1514.

Hwu, W.W., Conte, T.M., and Chang, P.P. 1989. Comparing software and hardware schemes for reducing the cost of branches. In *Proc., 16th Annual Internat. Symp. on Computer Architecture* (May), pp. 224–233.

Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *The J. Supercomputing*, 7, 1/2: 229–248.

IBM. 1967. *IBM J. Res. and Dev.*, 11, 1 (Jan.). Special issue on the System/360 Model 91.

IBM. 1976. *IBM 3838 Array Processor Functional Characteristics*. Pub. no. 6A24-3639-0, file no. S370-08, IBM Corp., Endicott, N.Y.

IBM. 1990. *IBM J. Res. and Dev.*, 34, 1 (Jan.). Special issue on the IBM RISC System/6000 processor.

Intel. 1989a. *i860 64-Bit Microprocessor Programmer's Reference Manual*. Pub. no. 240329-001, Intel Corp., Santa Clara, Calif.

Intel. 1989b. *80960CA User's Manual*. Pub. no. 270710-001, Intel Corp., Santa Clara, Calif.

Jain, S. 1991. Circular scheduling: A new technique to perform software pipelining. In *Proc., ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation* (June), pp. 219–228.

Johnson, M. 1991. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, N.J.

Jouppi, N.P. 1989. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Trans. Comps.*, C-38, 12 (Dec.): 1645–1658.

Jouppi, N.P., and Wall, D. 1989. Available instruction level parallelism for superscalar and superpipelined machines. In *Proc., Third Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Apr.), pp. 272–282.

Kasahara, H., and Narita, S. 1984. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. Comps.*, C-33, 11 (Nov.): 1023–1029.

Keller, R.M. 1975. Look-ahead processors. *Computing Surveys* 7, 4 (Dec.): 177–196.

Kleir, R.L. 1974. A representation for the analysis of microprogram operation. In *Proc., 7th Annual Workshop on Microprogramming* (Sept.), pp. 107–118.

Kleir, R.L., and Ramamoorthy, C.V. 1971. Optimization strategies for microprograms. *IEEE Trans. Comps.*, C-20, 7 (July): 783–794.

Kogge, P.M. 1973. Maximal rate pipelined solutions to recurrence programs. In *Proc., First Annual Symp. on Computer Architecture* (Univ. of Fla., Gainesville, Dec.), pp. 71–76.

Kogge, P.M. 1974. Parallel solution of recurrence problems. *IBM J. Res. and Dev.*, 18, 2 (Mar.): 138–148.

Kogge, P.M. 1977a. Algorithm development for pipelined processors. In *Proc., 1977 Internat. Conf. on Parallel Processing* (Aug.), p. 217.

Kogge, P.M. 1977b. The microprogramming of pipelined processors. In *Proc., 4th Annual Symp. on Computer Architecture* (Mar.), pp. 63–69.

Kogge, P.M. 1981. *The Architecture of Pipelined Computers*. McGraw-Hill, New York.

Kogge, P.M., and Stone, H.S. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comps.*, C-22, 8 (Aug.): 786–793.

Kohler, W.H. 1975. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Trans. Comps.*, C-24, 12 (Dec.): 1235–1238.

Kohn, L., and Margulis, N. 1989. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, 9, 4 (Aug.): 15–30.

Kunkel, S.R., and Smith, J.E. 1986. Optimal pipelining in supercomputers. In *Proc., 13th Annual Internat. Symp. on Computer Architecture* (Tokyo, June), pp. 404–411.

Labrousse, J., and Slavenburg, G.A. 1988. CREATE-LIFE: A design system for high performance VLSI circuits. In *Proc., Internat. Conf. on Circuits and Devices*, pp. 365–360.

Labrousse, J., and Slavenburg, G.A. 1990a. A 50 MHz microprocessor with a VLIW architecture. In *Proc., ISSCC '90* (San Francisco), pp. 44–45.

Labrousse, J., and Slavenburg, G.A. 1990b. CREATE-LIFE: A modular design approach for high performance ASICs. In *Proc., COMPCON '90* (San Francisco), pp. 427–433.

Lam, M.S.-L. 1987. A systolic array optimizing compiler. Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh.

Lam. M. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc., ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation* (Atlanta, June), pp. 318–327.

Lam, M.S., and Wilson, R.P. 1992. Limits of control flow on parallelism. In *Proc., Nineteenth Internat. Symp. on Computer Architecture* (Gold Coast, Australia, May), pp. 46–57.

Landskov, D., Davidson, S., Shriver, B., and Mallett, P.W. 1980. Local microcode compaction techniques. *ACM Computer Surveys*, 12, 3 (Sept.): 261–294.

Lee, J.K.F., and Smith, A.J. 1984. Branch prediction strategies and branch target buffer design. *Computer*, 17, 1 (Jan.): 6–22.

Lee, M., Tirumalai, P.P., and Ngai, T.-F. 1993. Software pipelining and superblock scheduling: Compilation techniques for VLIW machines. In *Proc., 26th Annual Hawaii Internat. Conf. on System Sciences* (Hawaii, Jan.), vol. 1, pp. 202–213.

Linn, J.L. 1988. Horizontal microcode compaction. In *Microprogramming and Firmware Engineering Methods* (S. Habib, ed.), Van Nostrand Reinhold, New York, pp. 381–431.

Lowney, P.G., Freudenberger, S.M., Karzes, T.J., Lichtenstein, W.D., Nix, R.P., O'Donnell, J.S., and Ruttenburg, J.C. 1993. The Multiflow trace scheduling compiler. *The J. Supercomputing*, 7, 1/2: 51–142.

Mahlke, S.A., Chen, W.Y., Hwu, W.W., Rau, B.R., and Schlansker, M.S. 1992. Sentinel scheduling for VLIW and superscalar processors. In *Proc., Fifth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Oct.), pp. 238–247.

Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., and Bringmann, R.A. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proc., 25th Annual Internat. Symp. on Microarchitecture* (Dec.), pp. 45–54.

Mallett, P.W. 1978. Methods of compacting microprograms. Ph.D. thesis, Univ. of Southwestern La., Lafayette, La.

Mangione-Smith, W., Abraham, S.G., and Davidson, E.S. 1992. Register requirements of pipelined processors. In *Proc., Internat. Conf. on Supercomputing* (Washington, D.C., July).

McFarling, S., and Hennessy, J. 1986. Reducing the cost of branches. In *Proc., Thirteenth Internat. Symp. on Computer Architecture* (Tokyo, June), pp. 396–403.

Moon, S.-M., Ebcioglu, K. 1992. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proc., 25th Annual Internat. Symp. on Microarchitecture* (Portland, Ore., Dec.), pp. 55–71.

Nakatani, T., and Ebcioglu, K. 1990. Using a lookahead window in a compaction-based parallelizing compiler. In *Proc., 23rd Annual Workshop on Microprogramming and Microarchitecture* (Orlando, Fla., Nov.), pp. 57–68.

Nicolau, A. 1984. Parallelism, memory anti-aliasing and correctness for trace scheduling compilers. Ph.D. thesis, Yale Univ., New Haven, Conn.

Nicolau, A. 1985a. Percolation scheduling: A parallel compilation technique. Tech. Rept. TR 85-678, Dept. of Comp. Sci., Cornell, Ithaca, N.Y.

Nicolau, A. 1985b. Uniform parallelism exploitation in ordinary programs. In *Proc., Internat. Conf. on Parallel Processing* (Aug.), pp. 614–618.

Nicolau, A., and Fisher, J.A. 1981. Using an oracle to measure parallelism in single instruction stream programs. In *Proc., Fourteenth Annual Microprogramming Workshop* (Oct.), pp. 171–182.

Nicolau, A., and Fisher, J.A. 1984. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. Comps.*, C-33, 11 (Nov.): 968–976.

Nicolau, A., and Potasman, R. 1990. Realistic scheduling: Compaction for pipelined architectures. In *Proc., 23rd Annual Workshop on Microprogramming and Microarchitecture* (Orlando, Fla., Nov.), pp. 69–79.

Oehler, R.R., and Blasgen, M.W. 1991. IBM RISC System/6000: Architecture and performance. *IEEE Micro*, 11, 3 (June): 14.

Papadopoulos, G.M., and Culler, D.E. 1990. Monsoon: An explicit token store architecture. In *Proc., Seventeenth Internat. Symp. on Computer Architecture* (Seattle, May), pp. 82–91.

Park, J.C.H., and Schlansker, M.S. 1991. On predicated execution. Tech. Rept. HPL-91-58, Hewlett Packard Laboratories.

Patel, J.H. 1976. Improving the throughput of pipelines with delays and buffers. Ph.D. thesis, Univ. of Ill., Urbana-Champaign, Ill.

Patel, J.H., and Davidson, E.S. 1976. Improving the throughput of a pipeline by insertion of delays. In *Proc., 3rd Annual Symp. on Computer Architecture* (Jan.), pp. 159–164.

Patterson, D.A., and Sequin, C.H. 1981. RISC I: A reduced instruction set VLSI computer. In *Proc., 8th Annual Symp. on Computer Architecture* (Minneapolis, May), pp. 443–450.

Peterson, C., Sutton, J., and Wiley, P., 1991. iWarp: A 100-MOPS, LIW microprocessor for multicomputers. *IEEE Micro*, 11, 3 (June): 26.

Popescu, V., Schultz, M., Spracklen, J., Gibson, G., Lightner, B., and Isaman, D. 1991. The Metaflow architecture. *IEEE Micro*, 11, 3 (June): 10.

Radin, G. 1982. The 801 minicomputer. In *Proc., Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar.), pp. 39–47.

Ramakrishnan, S. 1992. Software pipelining in PA-RISC compilers. *Hewlett-Packard J.* (July): 39–45.

Ramamoorthy, C.V., and Gonzalez, M.J. 1969. A survey of techniques for recognizing parallel processable streams in computer programs. In *Proc., AFIPS Fall Joint Computing Conf.*, pp. 1–15.

Ramamoorthy, C.V., and Tsuchiya, M. 1974. A high level language for horizontal microprogramming. *IEEE Trans. Comps.*, C-23: 791–802.

Ramamoorthy, C.V, Chandy, K.M., and Gonzalez, M.J. 1972. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. Comps.*, C-21, 2 (Feb.): 137–146.

Rau, B.R. 1988. Cydra 5 Directed Dataflow architecture. In *Proc., COMPCON '88* (San Francisco, Mar.), pp. 106–113.

Rau, B.R. 1992. Data flow and dependence analysis for instruction level parallelism. In *Fourth Internat. Workshop on Languages and Compilers for Parallel Computing* (U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds.), Springer-Verlag, pp. 236–250.

Rau, B.R., and Glaeser, C.D. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc., Fourteenth Annual Workshop on Microprogramming* (Oct.), pp. 183–198.

Rau, B.R., Glaeser, C.D., and Greenawalt, E.M. 1982. Architectural support for the efficient generation of code for horizontal architectures. In *Proc., Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar.), pp. 96–99.

Rau, B.R., Glaeser, C.D., and Picard, R.L. 1982. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proc., Ninth Annual Internat. Symp. on Computer Architecture* (Apr.), pp. 131–139.

Rau, B.R., Lee, M., Tirumalai, P., and Schlansker, M.S. 1992. Register allocation for software pipelined loops. In *Proc., SIGPLAN '92 Conf. on Programming Language Design and Implementation* (San Francisco, June 17–19), pp. 283–299.

Rau, B.R., Yen, D.W.L., Yen, W., and Towle, R.A. 1989. The Cydra 5 departmental supercomputer: Design philosophies, decisions and trade-offs. *Computer*, 22, 1 (Jan.): 12–34.

Riseman, E.M., and Foster, C.C. 1972. The inhibition of potential parallelism by conditional jumps. *IEEE Trans. Comps.*, C-21, 12 (Dec.): 1405–1411.

Ruggiero, J.F., and Coryell, D.A. 1969. An auxiliary processing system for array calculations. *IBM Systems J.*, 8, 2: 118–135.

Russell, R.M. 1978. The CRAY-1 computer system. *CACM*, 21: 63–72.

Rymarczyk, J. 1982. Coding guidelines for pipelined processors. In *Proc., Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar.), pp, 12–19.

Schmidt, U., and Caesar, K. 1991. Datawave: A single-chip multiprocessor for video applications. *IEEE Micro*, 11, 3 (June): 22.

Schneck, P.B. 1987. *Supercomputer Architecture*. Kluwer Academic, Norwell, Mass.

Schuette, M.A., and Shen, J.P. 1993. Instruction-level experimental evaluation of the Multiflow TRACE 14/300 VLIW computer. *The J. Supercomputing*, 7, 1/2: 249–271.

Sethi, R. 1975. Complete register allocation problems. *SIAM J. Computing*, 4, 3: 226–248.

Sethi, R., and Ullman, J.D. 1970. The generation of optimal code for arithmetic expressions, *JACM*, 17, 4 (Oct.): 715–728.

Sites, R.L. 1978. Instruction ordering for the CRAY-1 computer. Tech. rept. 78-CS-023, Univ. of Calif., San Diego.

Smith, J.E. 1981. A study of branch prediction strategies. In *Proc., Eighth Annual Internat. Symp. on Computer Architecture* (May), pp. 135–148.

Smith, J.E. 1982. Decoupled access/execute architectures. In *Proc., Ninth Annual Internat. Symp. on Computer Architecture* (Apr.), pp. 112–119.

Smith, J.E. 1989. Dynamic instruction scheduling and the Astronautics ZS-1. *Computer*, 22, 1 (Jan.): 21–35.

Smith, J.E., and Pleszkun, A.R. 1988. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comps.*, C-37, 5 (May): 562–573.

Smith, J.E., Dermer, G.E., Vanderwarn, B.D., Klinger, S.D., Roszewski, C.M., Fowler, D.L., Scidmore, K.R., and Laudon, J.P. 1987. The ZS-1 central processor. In *Proc., Second Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Oct.), pp. 199–204.

Smith, M.D., Horowitz, M., and Lam, M. 1992. Efficient superscalar performance through boosting. In *Proc., Fifth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Oct.), pp. 248–259.

Smith, M.D., Lam, M.S., and Horowitz, M.A. 1990. Boosting beyond static scheduling in a superscalar processor. In *Proc., Seventeenth Internat. Symp. on Computer Architecture* (June), pp. 344–354.

Smotherman, M., Krishnamurthy, S., Aravind, P.S., and Hunnicutt, D. 1991. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proc., 24th Annual Internat. Workshop on Microarchitecture* (Albuquerque, N.M., Nov.), pp. 93–102.

Sohi, G.S., and Vajapayem, S. 1987. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proc., 14th Annual Symp. on Computer Architecture* (Pittsburgh, June), pp. 27–36.

Su, B., and Ding, S. 1985. Some experiments in global microcode compaction. In *Proc., 18th Annual Workshop on Microprogramming* (Asilomar, Calif., Nov.), pp. 175–180.

Su, B., and Wang, J. 1991a. GURPR*: A new global software pipelining algorithm. In *Proc., 24th Annual Internat. Symp. on Microarchitecture* (Albuquerque, N.M., Nov.), pp. 212–216.

Su, B., and Wang, J. 1991b. Loop-carried dependence and the general URPR software pipelining approach. In *Proc., 24th Annual Hawaii Internat. Conf. on System Sciences* (Hawaii, Jan.).

Su, B., Ding, S., and Jin, L. 1984. An improvement of trace scheduling for global microcode compaction. In *Proc., 17th Annual Workshop on Microprogramming* (New Orleans, Oct.), pp. 78–85.

Su, B., Ding, S., and Xia, J. 1986. URPR—An extension of URCR for software pipelining. In *Proc., 19th Annual Workshop on Microprogramming* (New York, Oct.), pp. 104–108.

Su, B., Ding, S., Wang, J., and Xia, J. 1987. GURPR—A method for global software pipelining. In *Proc., 20th Annual Workshop on Microprogramming* (Colorado Springs, Colo., Dec.), pp. 88–96.

Thistle, M.R., and Smith, B.J. 1988. A processor architecture for Horizon. In *Proc., Supercomputing '88*, (Orlando, Fla., Nov.), pp. 35–41.

Thomas, A.T., and Davidson, E.S. 1974. Scheduling of multiconfigurable pipelines. In *Proc., 12th Annual Allerton Conf. on Circuits and Systems Theory* (Allerton, Ill.), pp. 658–669.

Thornton, J.E. 1964. Parallel operation in the Control Data 6600. In *Proc., AFIPS Fall Joint Computer Conf.*, pp. 33–40.

Thornton, J.E. 1970. *Design of a Computer—The Control Data 6600.* Scott, Foresman, Glenview, Ill.

Tirumalai, P., Lee, M., and Schlansker, M.S. 1990. Parallelization of loops with exits on pipelined architectures. In *Proc., Supercomputing '90* (Nov.), pp. 200–212.

Tjaden, G.S., and Flynn, M.J. 1970. Detection and parallel execution of parallel instructions. *IEEE Trans. Comps.*, C-19, 10 (Oct.): 889–895.

Tjaden, G.S., and Flynn, M.J. 1973. Representation of concurrency with ordering matrices. *IEEE Trans. Comps.*, C-22, 8 (Aug.): 752–761.

Tokoro, M., Tamura, E., and Takizuka, T. 1981. Optimization of microprograms. *IEEE Trans. Comps.*, C-30, 7 (July): 491–504.

Tokoro, M., Takizuka, T., Tamura, E., and Yamaura, I. 1978. A technique of global optimization of microprograms. In *Proc., 11th Annual Workshop on Microprogramming* (Asilomar, Calif., Nov.), pp. 41–50.

Tokoro, M., Tamura, E., Takase, K., and Tamaru, K. 1977. An approach to microprogram optimization considering resource occupancy and instruction formats. In *Proc., 10th Annual Workshop on Microprogramming* (Niagara Falls, N.Y., Nov.), pp. 92–108.

Tomasulo, R.M. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. and Dev.*, 11, 1 (Jan.): 25–33.

Touzeau, R.F. 1984. A FORTRAN compiler for the FPS-164 scientific computer. In *Proc., ACM SIGPLAN '84 Symp. on Compiler Construction* (Montreal), pp. 48–57.

Tsuchiya, M., and Gonzalez, M.J. 1974. An approach to optimization of horizontal microprograms. In *Proc., Seventh Annual Workshop on Microprogramming* (Palo Alto, Calif.), pp. 85–90.

Tsuchiya, M., and Gonzalez, M.J. 1976. Toward optimization of horizontal microprograms, *IEEE Trans. Comps.*, C-25, 10 (Oct.): 992–999.

Uht, A.K. 1986. An efficient hardware algorithm to extract concurrency from general-purpose code. In *Proc., Nineteenth Annual Hawaii Conf. on System Sciences* (Jan.), pp. 41–50.

Wall, D.W. 1991. Limits of instruction-level parallelism. In *Proc., Fourth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, Calif., Apr.), pp. 176–188.

Warren, H.S. 1990. Instruction scheduling for the IBM RISC System/6000 processor. *IBM J. Res. and Dev.*, 34, 1 (Jan.): 85–92.

Warter, N.J., Bockhaus, J.W., Haab, G.E., and Subramanian, K. 1992. Enhanced modulo scheduling for loops with conditional branches. In *Proc., 25th Annual Internat. Symp. on Microarchitecture* (Portland, Ore., Dec.), pp. 170–179.

Watson, W.J. 1972. The TI ASC—A highly modular and flexible super computer architecture. In *Proc., AFIPS Fall Joint Computer Conf.*, pp. 221–228.

Wedig, R.G. 1982. Detection of concurrency in directly executed language instruction streams. Ph.D. thesis, Stanford Univ., Stanford, Calif.

Weiss, S., and Smith, J.E. 1984. Instruction issue logic for pipelined supercomputers. In *Proc., 11th Annual Internat. Symp. on Computer Architecture*, pp. 110–118.

Weiss, S., and Smith, J.E. 1987. A study of scalar compilation techniques for pipelined supercomputers. In *Proc., Second Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Oct.), pp. 105–109.

Wilkes, M.V. 1951. The best way to design an automatic calculating machine. In *Proc., Manchester Univ. Comp. Inaugural Conf.* (Manchester, England, July), pp. 16–18.

Wilkes, M.V., and Stringer, J.B. 1953. Microprogramming and the design of the control circuits in an electronic digital computer. In *Proc., The Cambridge Philosophical Society, Part 2* (Apr.), pp. 230–238.

Wolfe, A., and Shen, J.P. 1991. A variable instruction stream extension to the VLIW architecture. In *Proc., Fourth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, Calif., Apr.), pp. 2–14.

Wood, G. 1978. On the packing of micro-operations into micro-instruction words. In *Proc., 11th Annual Workshop on Microprogramming* (Asilomar, Calif., Nov.), pp. 51–55.

Wood, G. 1979. Global optimization of microprograms through modular control constructs. In *Proc., 12th Annual Workshop on Microprogramming* (Hershey, Penn.), pp. 1–6.

Yau, S.S., Schowe, A.C. and Tsuchiya, M. 1974. On storage optimization of horizontal microprograms. In *Proc., Seventh Annual Workshop on Microprogramming* (Palo Alto, Calif.), pp. 98–106.

Yeh, T.Y., and Patt, Y.N. 1992. Alternative implementations of two-level adaptive branch prediction. In *Proc., Nineteenth Internat. Symp. on Comp. Architecture* (Gold Coast, Australia, May), pp. 124–134.

Zima, H., and Chapman, B. 1990. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, Reading, Mass.