PARALLEL PROCESSING, SUPERSCALAR AND VLIW PROCESSORS

All modern central processing units and most embedded processing units execute multiple instructions per cycle in parallel, by exploiting the implicit parallelism available in ordinary programs. Such instruction-level parallelism is regulated by many factors, including the algorithm, the implementation of the algorithm, and the efforts of the compiler. It is then the job of the processor to correctly execute the program to preserve the original program meaning. In general, parallel execution of instructions may be specified in the algorithm by employing explicit parallel directives to the hardware, in the program by employing parallel language constructs, in the compiler by employing parallelization techniques, or in the hardware by employing automated parallelization mechanisms. The latter two approaches—relegating the task to the compiler or the hardware—are termed *very long instruction word* (*VLIW*) and *superscalar* processing, respectively. This article will review these techniques, highlighting the key similarities and differences between these two approaches. First, superscalar processors are reviewed, followed by VLIW processors. The article concludes with comparisons between both processors.

INSTRUCTION-LEVEL PARALLELISM

Instruction-level parallelism, or ILP, is a phenomenon that exists in nearly all programs to varying degrees. An example is shown in Fig. 1. Here a pseudo machine language has been used to clarify the illustration. The instructions on the left side of Fig. 1 can be rearranged from their sequential order into a partial order without change. This issue is shown on the right side of Fig. 1 which shows a graph connecting with arc operations that must execute in sequence. Any pair of instructions not connected by an arc are free to execute in parallel without affecting the outcome of the program. The arcs in this case are termed *data dependencies*.

The ILP property of instruction streams can be determined before program execution by the programmer, by the compiler, by the assembler, or by other software in the system. If this is not done before run time, then it may be done during run time by the hardware. In either approach, the fact that the program is executing in parallel at the instruction level is entirely hidden from the user.

In addition to data dependencies, control dependencies can also force sequential execution of programs. This issue is shown in Fig. 2, where a simple control flow of an if-then-else statement has been converted to a directed graph. In this code, block B or C must wait until the decision in block A is determined. However, it is interesting to note that block D is independent of A's decision. In this case, any instructions in D that are independent of results from B's or C's instructions can execute alongside A's instructions. Although it is tempting to ignore this level of analysis and only execute code in parallel until branches (i.e., the branch at the end of A) are resolved, such a conservative decision reduces the benefits of ILP considerably. For nonscientific code (i.e., for so-called "integer" code), ILP has been measured at a high of approximately 2 to 3 instructions per cycle between branches. When control dependencies are resolved and ILP is searched for across branches, this empirical figure grows by an order of magnitude. Thus control dependencies must be dealt with in the hardware or software.

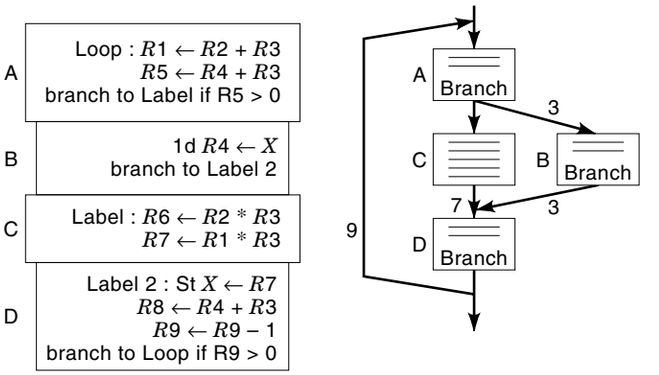

Figure 2. An illustration of a control flow graph for an if-then-else embedded in a loop.

VLIW AND SUPERSCALAR PROCESSORS

Both VLIW and superscalar processors are designed as instruction assembly lines or *pipelines* of stages. Stages are separated from each other by latches or flip-flops, which pass their contents onto the next stage based on the cycle of the processor's clock. (There are more complex pipelining techniques that compose multiple stages between latches, but that is an advanced topic for interested readers and is not covered here.) The responsibilities of the stages are what distinguish the two processors, and this is what this article focuses on. The following section describes the superscalar pipeline. Since a VLIW pipeline is relatively simple, it is described in terms of its associated compiler passes.

OVERVIEW OF SUPERSCALAR IMPLEMENTATION

A basic superscalar processor is depicted in Fig. 3. It is composed of the following stages:

1. *Instruction Fetch.* In this stage or stages, instructions are fetched from memory and decoded for future ease of

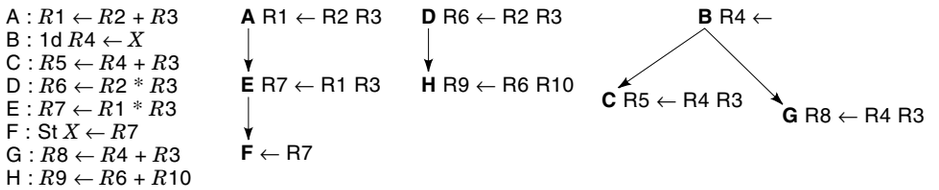

Figure 1. An example of instructions and their partial order that allows for instruction-level parallelism.

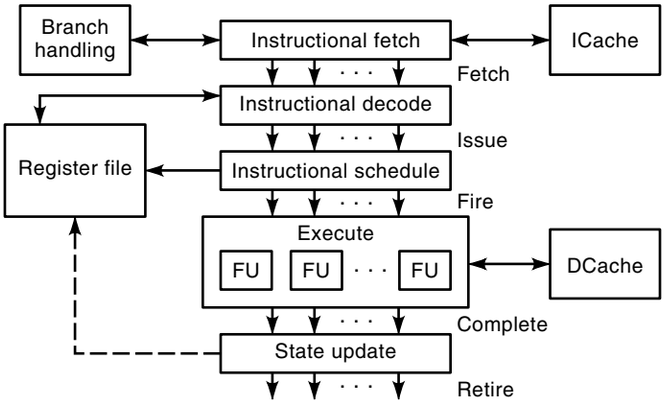

Figure 3. The anatomy of a simple superscalar processor pipeline.

execution. Branches are predicted at this point in the hardware (explained further below).

2. *Instruction Decode.* This stage reads register values, determines the hardware inside the processor that will be required to execute the instruction, and reserves this hardware when necessary.
3. *Instruction Scheduling.* Instruction scheduling is an important stage in a superscalar processor. It determines which instructions can execute in parallel and which must wait for later cycles.
4. *Execution.* The execution unit is actually a pool of several units, typically an ALU and an interface to the data cache, plus floating-point units (i.e., FP add, FP multiply) and special-purpose units (e.g., shifters, multiply-accumulate units, motion estimation for decoding digital video). There is relatively little novel in the design of these units that is specific to superscalar processors. The data cache may be multiported and allow nonblocking accesses while misses are repaired (1). The execution unit will be omitted from the detailed discussion below.
5. *State Update.* The state update unit has the responsibility of maintaining consistent sequential state so that interrupts (either internal or external) can be handled. A detailed example to illustrate the need for this unit is presented below. There are several techniques that have been developed to handle state update. The most common are explained here.

Superscalar Instruction Fetch

Instruction fetch for a superscalar is a difficult task. The processor must acquire a parallel stream of instructions from memory in order to support the parallel core of the superscalar. The alignment of instructions in memory, memory hierarchy (cache) performance, and the presence of branches complicate this.

Branch Prediction. This uses hardware to predict if instructions at a given location in memory are branch instructions that will transfer program control, preventing execution of the next instruction in program memory (2). Branch prediction hardware must also predict where program control will be transferred. This is often done by maintaining a record of where program control was transferred the last time the branch was executed (in the case of return instructions, a small hardware stack is often used). This article focuses on the prediction process itself. Branches that transfer control are said to be *taken branches,* whereas branches that do not transfer control are *not taken branches.* The prediction problem restated is, given a location in memory holding an instruction (which may or may not be a branch), predict whether the instruction is a taken branch or a not taken branch.

One-Level Branch Prediction. These schemes use the instruction address to index into a buffer that contains a small state machine. An example of such a machine is the Smith counter, which uses a small, saturating up/down counter (2). This counter is incremented when the branch is actually taken and decremented when it is not taken. If the counter is greater than or equal to half its range, the branch is predicted taken, otherwise it is predicted not taken. The counter saturates at each extreme, so that a three-bit counter value of 7 remains at 7 if a branch is actually taken. Empirically, the Smith counter performs fairly well, with a 85% to 90% accuracy for nonnumeric code (nonnumeric code is not heavily loop-based, and loop closing branches are relatively easy to predict).

Two-Level Branch Prediction. This uses additional information to access a predictor. The Yeh/Patt "PAs" scheme is a good example of this idea. The instruction address is used to index into a buffer that contains an N-bit shift register. This register is a *history* of what occurred at this address. A "0" indicates a branch was not taken, whereas a "1" indicates it was taken. The history register is used to index into a $2N$ table of Smith counters (other state machines are possible). These counters are then used to predict the branch. The interesting feature is that the counters "learn" to predict all branches with the same history, as opposed to learning about a branch at a particular address in memory. The accuracy for this scheme is higher than the one-level predictor described above—empirically 96% to 98% accurate.

Instruction Fetch Issues. The mechanics of how branch prediction is integrated into the instruction cache for a combined instruction fetch mechanism is beyond the scope of this article, since its proper treatment requires understanding of cache memory design. However, it is possible to explain the periphery of the instruction cache. Each cache holds multiple bytes for a given address range in a fixed-sized cache block. An instruction address is used to index into the cache block, which delivers multiple potential instructions. If instructions are fixed-length (as in most RISC architectures), then the number of instructions in each cache block is known. Note that each block contains a sequential list of instructions as they appear in memory. Branch prediction can be used to fetch multiple cache blocks from the instruction cache in order to circumvent this sequential limitation. The predicted instructions must then be pulled from these multiple blocks, as described in Ref. 3. An alternative is to store instructions in the cache in groupings according to their prediction, which has been called a *trace cache* (4). Since the predicted groupings are not known until the branch predictor has observed branch behavior, a trace cache is often combined with a traditional instruction cache as a backup.

Superscalar Instruction Decode

In terms of program execution, the decoder not only determines the meaning of the bit encodings of the instructions, but in a superscalar it also determines the meaning of a sequence of instructions. It is the last unit in the processor's pipeline before instructions are allowed to go out of order. The function of the decoder is, therefore, to interpret this sequential order and derive the partial order that the hardware must obey. Recall that superscalars execute several instructions at once, in parallel. Consider the situation shown in Fig. 4. In this figure, several instructions are generating values for register R3 (A, C, and E). Several instructions use these values (B, D, F). However, the value needed by F is different from that needed by D, which is, in turn, different from that needed by B. This implies that the register number (i.e., 3) is not sufficient for identifying the value when instructions are executed in parallel. Now consider Fig. 4(b). Here R3 has been replaced by a new numbering scheme, shown as Tx, where x is referred to as a *tag.* Now there is no ambiguity about value identities when instructions are executed in parallel.

The register file is modified to help with the renaming task. A traditional register file is indexed by the register number from the decoded instruction format, and it holds the register value. The modified register file also holds a *ready bit* and a *tag* (or *unique name*) for each register in addition to the register's value. The ready bit is a flag that, if true, indicates the value stored is the correct value and should be used in the computation. However, if the ready bit is set to false, then this means the value field is no longer valid. In this case, the instruction must wait for the valid register value. It uses the tag field in the register file to know the identity of the register to wait on. It is also the decoder's responsibility to assign unique tag values for all destination registers and store this assignment into the register file. Once this is done, the destination register is marked as not ready by assigning false to its ready bit.

Instructions are decoded into register values, register numbers, tags, and destination functional units. The decoded instruction is then written into a buffer for the scheduler. This buffer is called the *reservation buffer* and is discussed further below when the scheduler is presented.

In accordance with the Tomasulo algorithm (5) of the IBM 360 model 91, the decoder performs these tasks: For each instruction from the instruction fetch unit, examine the instructions in program order and:

1. Examine the source registers in the register file and, if the *ready bit* is true, copy the register value to the reservation buffer, otherwise copy the *tag* to the reservation buffer.
2. Assign a unique tag to the destination register of the instruction and copy it to the reservation buffer. Also, write this same tag into the register file, setting the *ready bit* to false. Copy the destination register number from the instruction format into the reservation buffer (the purpose of this is explained below).
3. Mark the destination functional unit (FU) type in the reservation buffer (the scheduler uses this to select a functional unit from the pool of FUs in the execution unit).

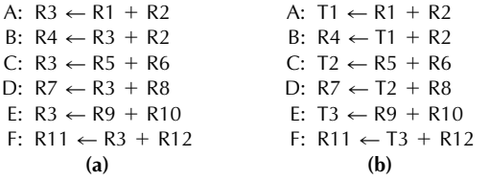

Figure 4. The effects of renaming registers to enhance ILP.

Once these steps have been followed, the instruction is said to be *issued* to the functional unit. The rate at which instructions can be issued is a figure of merit for superscalar processors since it limits how many instructions can execute in parallel. A typical value is an issue rate of four instructions per cycle.

Superscalar Instruction Scheduling

The instruction-scheduling unit of a superscalar processor manages the reservation buffer, guaranteeing that instructions execute in the partial order dictated by their dependencies. This work happens in two phases: before instructions begin execution and after they complete execution. Any instruction for which all of its source registers are marked as ready in the reservation buffer is ready to execute. It may still have to await an available functional unit in the execution unit's pool of FUs. However, once a unit is free, the instruction can proceed with no fear of violating program dependencies. This action is termed *firing* an instruction for execution. This first phase is, therefore,

> *Phase I (Before Instruction Execution).* For all instructions in the reservation buffer, if any entry has all of its source registers marked as ready and its associated FU is not busy, then begin executing the instruction (*fire* the instruction).

It is important to note that the instruction is not deleted from the reservation buffer when it is fired. This is because the reservation buffer holds the destination tag, which is needed when the instruction completes. This is explained further below.

Firing instructions have hardware complications. Instructions must be shipped to their destination functional units. If there are M entries in the reservation buffer and N FUs, then this operation requires an $M \times N$ crossbar interconnect. Interconnect is inherently slow and may result in a pipeline stage worth of communication time between the reservation buffer and the destination FUs. The original IBM 360 model 91 removed this constraint by placing the $M \times N$ crossbar interconnect between the decode unit and the reservation buffer, then subdividing the reservation buffer into N subbuffers, one per each FU. Tomasulo's original name for these buffers was a *reservation station,* and this notation is preserved here (5). The notion of one, unified reservation buffer will be preserved for the remainder of this discussion, without loss of generality.

As instructions complete from their respective FUs, the scheduling unit in phase II must check their tags against all of the tags of the source registers, of all instructions in the reservation buffer. If there are any matches, the reservation buffer entry for those registers are marked as completed. The tag of the destination register of the completing instruction is taken from its own reservation buffer entry. Thus, the reser-

vation buffer is searched in a fully associative manner (i.e., as a content addressable memory), using the completing instruction's tag as the name of the item being searched for and the set of tags of source registers of waiting instructions as the names being compared against. One method to reduce the complexity of this search is discussed below.

As a hardware consideration, it is important to note that the FUs and the reservation buffer are often in separate parts of the processor's layout. Thus a bus is needed to broadcast the completing instruction to the reservation buffer. This bus must hold the tag of the destination register. It is convenient for it to also hold the result of the computation, so that the register file does not need to be examined in phase I for every ready instruction. If this extra information is added, then the register file may watch this bus as well as the reservation buffer. The register file can thereby use the broadcast of the value on the bus to update the register's contents. Thus the bus must also hold the number of the destination register. To avoid problems with the scenario of Fig. 4, the register file should ignore any broadcast for a register whose tag value does not match the tag value stored in the register file by the decode unit. To summarize, this bus [called the *common data bus* in Tomasulo's description of the IBM 360 model 91 (5)] is written to by the completing instruction as it exits the FU, and read by the reservation buffer and the register file.

A summary for phase II is:

Phase II (After Instruction Completion)

1. For each entry in the reservation buffer, if the tag of the completing instruction's destination register matches a source register tag, then copy the value (i.e., the result of the completing instruction's computation) from the common data bus and set the source register's ready bit to true.
2. Look up the destination register in the register file. If the tag stored in the register file matches the tag from the common data bus, then copy the value from the bus into the register file and set the register's ready bit to true.
3. Delete the completed instruction from the reservation buffer.

It is important to note that the rate of instructions completing is limited by the bandwidth of the common data bus. In most modern processors, this bus is replaced by a set of identical buses, often referred to as *result buses.* They serve the same function as the common data bus, but the scheduling unit must check all of these buses in parallel.

As noted above, the reservation buffer must be searched for every completing instruction's destination tag value. One method to avoid this search is to update the register file only and then periodically (i.e., once per cycle) update the reservation buffer by checking every source register of every instruction against the contents of the register file. To avoid the problem of Fig. 4, each tag must have a unique entry in the register file. This changes the meaning of tags slightly: in this scheme they are commonly referred to as *virtual registers* or *rename registers.* Thus the tag is no longer a phantom name for the value being produced by an instruction. Rather, it is a physical location in the register file. It is not, however, a location that the program/programmer can access directly, and as such they are often also referred to as *nonarchitected registers.* It is important to note that this variation is not needed for correct execution of the Tomasulo algorithm, rather it enhances the ease of implementation of the algorithm (5).

Superscalar Interrupt Precision and Speculation

Superscalar processors suffer from a problem with exceptions and interrupts. Consider the example instruction sequence in Fig. 5(a). This sequence is shown executing on a superscalar processor in Fig. 5(b). A *load* instruction causes a page fault. But when this fault occurs, instructions I1, I3, and I4 have completed and written back their results to the register file. For the *load* instruction to be reexecuted (after the page fault is repaired by the operating system), the original values of registers R3 and R4 need to be restored. In the original IBM 360 model 91, this problem was referred to as the imprecise interrupt problem, since it was sometimes impossible to find a PC value to resume to when an interrupt (or program error) occurred. As such, it retains that name today.

The state update unit in a superscalar processor repairs the imprecise interrupt problem. One technique is to provide a *reorder buffer,* which preserves the sequential state of operations and writes back these operations in program order. In order for operations to still execute out-of-order, two copies of the register file are kept—the normal *Tomasulo* copy and a *future file,* which holds a known sequential state (6). A variation on this technique uses the reorder buffer itself as a mechanism for renaming and avoids the second register file (7). Another alternative technique, which has been implemented in some commercial processors, uses three copies of the register file—one current copy and two backups. The backup copies are periodically built from the current state of the completing instructions (8).

Branch speculation can be implemented using the logic that the state update unit employs to handle exceptions. Whenever a branch is predicted by the instruction fetch unit, the state update unit can make note of this and allow the code after the branch to execute *speculatively* until the outcome of the branch is known. When the outcome is known, the state update unit either commits the results of the branch to the archival state (i.e., in the future file or in a backup copy of the execution), or flushes the incorrectly speculated results from the machine and resumes execution down the correct path through the program. In this way, incorrectly predicted branches can be treated as small exceptions. Depending on

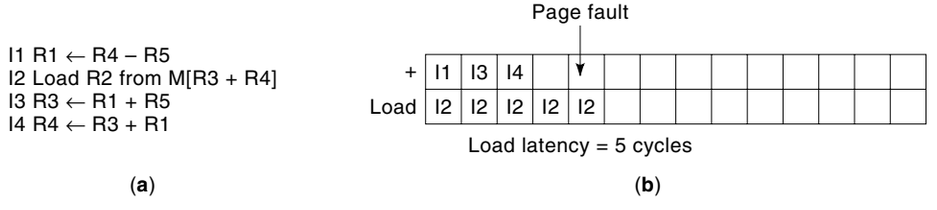

Figure 5. An example instruction sequence (a); and the problem of imprecise interrupts when it is executed out-of-order in (b).

the accuracy of the branch prediction, the speculative work can turn into nonspeculative work and enhance ILP.

OVERVIEW OF VLIW IMPLEMENTATION

The difference between superscalar and VLIW approaches to exploiting ILP is illustrated in Fig. 6. Superscalar assumes that the code is initially unscheduled (not parallel). It then forms a parallel schedule in hardware, using methods such as the Tomasulo algorithm, as explained above. However, the parallelism in Fig. 1 is relatively simple for software to analyze. Although a programmer could do this work when the code is written, it is tedious. The compiler is the most often employed software used to determine parallel execution. This is shown in Fig. 6(b). Here the compiler has expressed the parallel nature of the code to the processor as a set of operations to execute in each cycle. Because the instruction now contains multiple operations, the instruction format is itself very long. Hence this is termed a *very long instruction word* or VLIW processor (9).

The instructions of Fig. 1, shown in Fig. 7, are scheduled for a VLIW machine (with a 2-cycle load pipeline, a 3-cycle multiplier, and all other units are single cycle). Empty spaces in the figure represent cycles in which no operations are executed. These spaces do not need to be explicitly encoded in the instructions, as described in Ref. 10. Because the term *instruction* is ambiguous in a VLIW (i.e., does it refer to the entire row or one entry in that row?), the term *MultiOp will be used* for a row in the schedule and the term *Op* will be used for a single operation scheduled on a unit in a particular MultiOp. [This terminology is derived from the Cydra 5 (11).]

In a strict VLIW processor design, hardware performs no interlocking whatsoever. Instead, it is the responsibility of the compiler to schedule code so that dependencies between operations are never violated. This shift in emphasis from hardware to software is the main advantage of VLIW processors over superscalar processors. This section discusses the architecture of VLIW processors by focusing on the compiler techniques developed for their use.

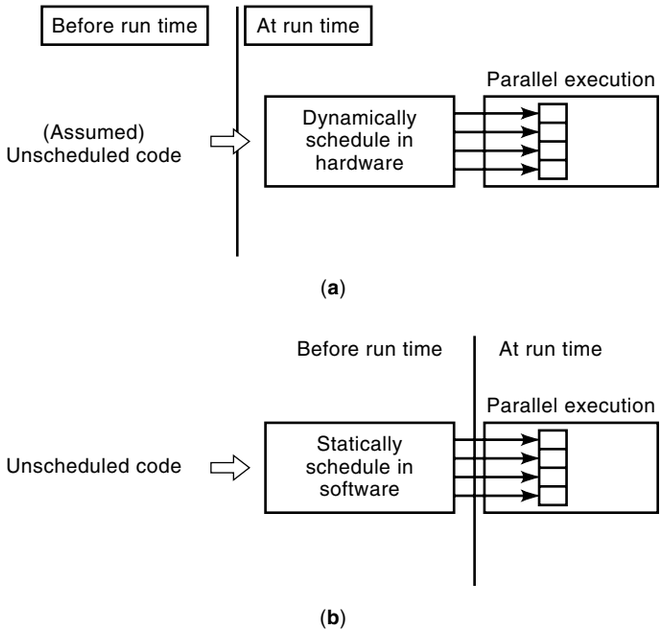

Figure 6. A perspective on the differences between superscalar (a) and VLIW (b) methods for exploiting ILP. (a) a view of superscalar; (b) a view of VLIW.

VLIW Compiler Considerations

A traditional compiler is composed of three phases: (1) parse the source language into an intermediate representation, (2) optimize the intermediate language, and (3) generate code of the target architecture from the intermediate language. The scheduling of resources is implicit in the code-generation phase. For a VLIW architecture, scheduling becomes a primary function and is often merged with optimization.

A compiler views a program in a way that is distinct. For example, the program is represented as *intermediate code,* which is typically an instruction set of a very simple architecture. The operation repertoire includes rudimentary arithmetic and logical operations, floating-point operations, control flow, and memory load/store operations. Each intermediate-language operation has one destination register and several source registers. An exception to this is control transfers (e.g., branches), which specify a destination address but no destination register. The only access to memory is through load and store operations. There is also an unlimited supply of registers available. These registers are often termed *virtual registers* for this reason.

In addition to the intermediate language, a compiler also maintains two primary data structures that describe the program. They are the *control flow* and *data flow graphs,* presented at the beginning of this article. As the names imply, these graphs describe the flow of control and data in the program. They do not completely characterize the program. That is to say, that these two graphs do not contain enough information to describe the computation without the addition of the intermediate-language operations.

Figure 2(a) shows a short example list of intermediate-language operations. These operations can be partitioned into blocks of guaranteed-sequential operations. These groupings are known as *basic blocks.* To form basic blocks, the following procedure is used: the code is scanned and a new basic block is started immediately after a branch or a code label. Operations are added to the block until another branch or code label is reached. Basic blocks are typically numbered sequentially, starting from the beginning of the source file or function being compiled. If destinations of branches at the bottom of basic blocks are connected to their target blocks by arcs, a basic block graph or *control flow graph* is formed [see Fig. 2(b) for an example].

A data flow graph is formed as described above (see the section titled "Instruction-level parallelism"). Recall that the Tomasulo algorithm removed false dependencies of Fig. 4 using hardware renaming via tags. The principal goal of renaming is to decouple the register names from their values so that register reuse in the program does not enforce a sequential execution order on the operations. For the compiler, the unlimited number of virtual registers accomplishes the same task, since each operation defines the value of a new virtual register name.

Compiler-Based Scheduling

Algorithms for VLIW scheduling are heuristic based. This is because the resource-constrained scheduling problem is NP-

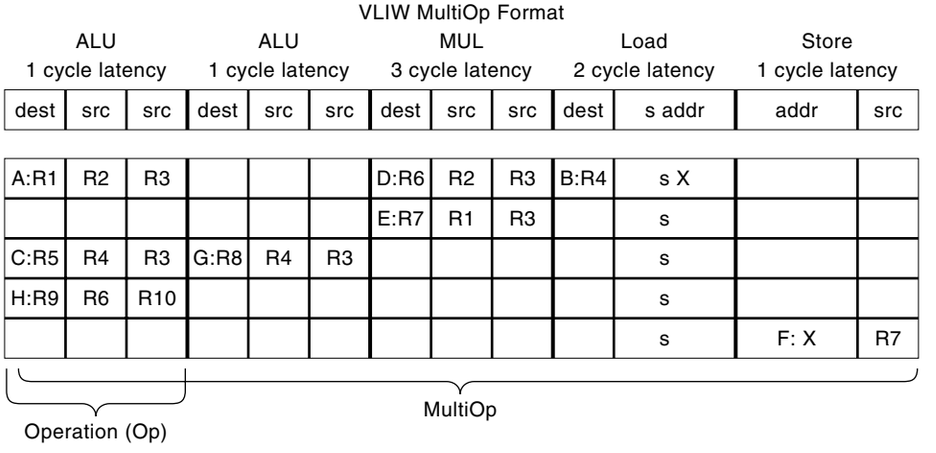

Figure 7. A VLIW schedule for the instructions of Fig. 4.

complete (i.e., requiring exponential time in the number of inputs) when the optimal schedule is sought. A very common heuristic algorithm is *list scheduling* (12). When the algorithm is applied to a single basic-block, it is termed *local scheduling* (its converse, *global scheduling,* is discussed below). In list scheduling, the dependence graph is first sorted into a list using a heuristic priority function. An example function might be the depth of the intermediate-language operation in the dependence graph. An empty schedule is created and the first operation is scheduled in the first cycle of the schedule. The scheduler then tries to schedule all additional operations in the list subject to available functional units (i.e., fields in the VLIW MultiOp) and dependencies imposed by the dependence graph. Any successfully scheduled operation is deleted from the list. After the entire list has been searched, the cycle pointer is incremented to the next cycle and the process repeats itself for the remaining operations on the list. Once the list is empty, the scheduling process is complete. In one variation of the list scheduling algorithm, the list is reordered based on the priority function at the end of each cycle. This is sometimes referred to as using a dynamic priority function, since the value of the operations' priorities depend on the current cycle pointer.

It is possible for the compiler to schedule operations such that there would be no difference in timing between execution of the operations of Fig. 1 on a superscalar employing the Tomasulo algorithm and the execution of the equivalent VLIW MultiOps. In essence, the very long instruction words are entire scripts for the functional units to follow in each cycle of execution. The dynamic responsibilities of the hardware have been reduced to obeying the dictates of the instruction format, without any hardware support to enforce dependencies. Nonetheless, the VLIW architecture can achieve the same or greater performance as a superscalar architecture. There are many reasons for this, including the cycle time advantage of simpler hardware, and the compiler's ability to find more parallelism before execution than hardware can find during execution.

The above discussion illustrates *local scheduling.* Unfortunately, the size of basic blocks is typically only four to six operations long. This limits the amount of parallelism a VLIW can extract in much the same way as branches limit the parallelism of superscalar processors. To extract more parallelism, *global scheduling* is used. This technique first builds larger blocks for scheduling out of basic blocks, then invokes a sometimes modified version of list scheduling on this larger scheduling scope. Since operations may be moved above a branch (i.e., between basic blocks), it is the VLIW analogue of branch prediction for speculative execution in a superscalar.

Global scheduling can be broadly classified into *acyclic* and *cyclic* scheduling. Acyclic scheduling deals with sequential lists of blocks with control flow containing no loops. When loops occur, acyclic techniques can still be used by breaking one of the arcs and scheduling the loop body as a sequential code. However, better results are often obtained when cyclic scheduling techniques are employed. The following section reviews several techniques for acyclic and cyclic scheduling.

Acyclic Global Scheduling

The first step in many acyclic scheduling techniques is to form larger groups of blocks out of basic blocks. There are many techniques for performing this grouping, including *Trace selection* (13), *superblock formation* (14), *hyperblock formation* (15), and *treegion formation* (16).

Trace selection was first used by pioneer VLIW researchers at Yale University in the *Bulldog* compiler (17), then later extended for inclusion in the commercial compilers of Multiflow (18), one of the early VLIW processor vendors. Superblock formation is an evolution of trace selection, used in the Illinois IMPACT project (19). The algorithms are similar. Superblock formation is described here, and then contrasted with trace selection, hyperblock formation, and treegion formation.

Figure 8(a) shows a control flow graph composed of basic blocks. The numbers or *weights* beside the blocks and arcs are execution counts for each arc. Obtaining the weights can be performed using information from profiled runs of the program, via software estimates, or by use of specially designed performance monitoring hardware. A control flow graph annotated in this way is often referred to as a *weighted control flow graph.*

Superblocks are formed first by grouping blocks together that tend to execute sequentially. Such groupings were termed *traces* by Fisher (13). The result of trace selection is shown in Fig. 8(b). The traces are represented as dashed rectangles in the figure. A superblock is a trace that has only one

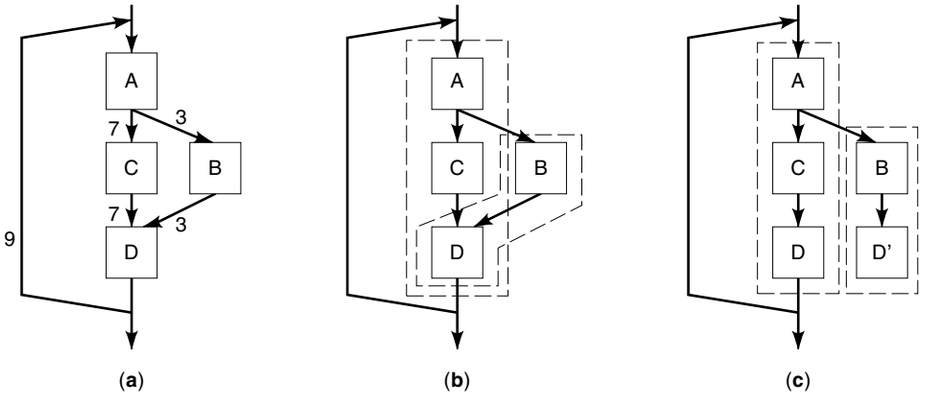

Figure 8. Superblock formation from a control flow graph. Note that D′ is a copy of all instructions inside basic-block D. (a) original control flow graph; (b) after trace selection; (c) after code duplication to remove side entrance B to D. (Numbers are execution frequencies along arcs of control flow graph.)

entrance at the top, but any number of multiple exits. No side entrance into a superblock is allowed. Notice that the larger trace in Fig. 8(b) is not a superblock because **B** transfers control into the middle of the trace. To solve this problem, *tail duplication* is performed. Specifically, **D** is duplicated. The overall result is shown in Fig. 8(c). Here notice that **D** has been duplicated so that execution after **B** now flows to **D**′. [An excellent description of the complete superblock algorithm by its inventors is presented in Ref. 14.]

The interesting property of superblocks is that operations can be moved upwards in a superblock across the boundaries of basic blocks. This is a direct consequence of the no-side-entrances rule for superblock formation. It allows code motion that can extend the scope of local scheduling. This motion is limited by data dependencies. Consider a branch and an operation, X, from the fall-through path to be moved above this branch. For the purpose of the example, say the operation writes its results to register R1. If any operations along the taken path of the branch use R1, then X cannot be moved above the branch unless its destination register is changed from R1 to another register. Another alternative is for the compiler to insert patch-up code into basic blocks not in the superblock (i.e., on the taken path of the branch) to undo the effects of X's speculation.

Some additional hardware modifications are also required to enable speculative execution of potentially excepting operations. An extra bit is used in the VLIW encoding of each operation, to indicate the operation is being executed speculatively. If the speculative operation generates an exception, an imprecise interrupt problem exists in much the same way as it does for superscalar processors. This is solved via slight modifications to the register file to signal an exception when the result of an excepting operation is used. This modification to handle interrupts is referred to as *sentinel scheduling* (20).

Work has been done on scheduling algorithms that avoid code duplication and allow for code motion in both directions. The most notable of these techniques is the *hyperblock scheduling* technique of the Illinois IMPACT project (15). This technique relies on the use of *if-conversion* (21) and *predicated execution* (11).

Predicates are one-bit registers that control whether the results of an operation are retired or discarded. Support for predicated execution requires the addition of a predicate register field to all operations in the VLIW MultiOp encoding. There are two ways to use these registers. In one technique, the predicate register is checked when an operation is about to begin execution. If the register holds the value **false,** the operation is abandoned, otherwise it is executed normally. In the second technique, the specified predicate register is checked when the corresponding operation completes execution on its functional unit. If the predicate holds the value **false,** the results of the operation are discarded instead of being written back. Some proposed VLIW architectures support the former technique, others support the latter.

Predicated execution is a mechanism for removing conditional, acyclic branches entirely from code sequences. To see this, consider the example of Fig. 2(a). Figure 9 shows a predicated version. The predicate specifier is represented by the keyword "if P2" in the intermediate language. Note the operation "P2 = cmpp(R1 = 0)": This is a *predicate-define* operation. It tests the condition (e.g., R1 equals zero) and, if the condition is true, sets the predicate register P2 to **true,** else sets P2 to **false.** Note how the inner loop branch has now been converted into a data dependence on the predicate register P2. This observation is the reason that conversion of code from branch-based control flow to predicated form is termed *if-conversion.*

Hyperblock formation uses if-conversion and predicated execution to remove short forward branches and create larger blocks of sequential code. Consider again the example of Fig. 8, where **D** had to be duplicated. If instead a predicate were used to merge **B** into the superblock, the resulting *hyperblock* would not require any code duplication. In addition, the scheduler can move operations in a hyperblock in any direction, as long as the dependencies are obeyed. As mentioned above, the control flow arc in the control flow graph is converted into a dependence arc in the data flow graph by if-

```
Loop : R1 ← R2 + R3
       R5 ← R4 + R3
  pr2 = cmpp(R5 > 0)
  1d R4 ← X if not pr2
  R6 ← R2 * R3 if pr2
  R7 ← R1 * R3 if pr2
       St X ← R7
       R8 ← R4 + R3
       R9 ← R9 –1
  pr3 = cmpp(R9 > 0)
  branch to Loop if pr3
```

Figure 9. The example of Fig. 2(a) *if-converted* using predicates. (Affected region is shown in the rectangle.) The entire loop may now be considered as a hyperblock.

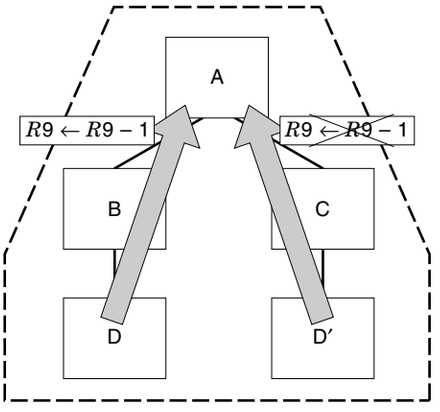

Figure 10. A treegion scheduling decision wherein an operation for D and D′ (the decrement of register R9) can be combined into one operation.

conversion. One drawback of hyperblock scheduling is visible from the example. Scheduled code that executes when p2 is **false** competes for resources with code that executes when p2 is **true.** Thus, although the resulting schedule may appear dense, this may not be the case when the code is executed. But predication does serve an important role in removing hard-to-predict branches, thereby allowing speculation when otherwise none could have been performed (22).

One very recent technique for global code scheduling uses a *tree-shaped region,* referred to as a *treegion* (16). In this technique, code is grouped into trees, then code motion is performed. Code duplication is used to enhance the number of trees in the control flow graph. One advantage of treegion scheduling is illustrated in Fig. 10. Here an operation is moved both from block D to block A, and its copy is moved from block D′ into A. The copy can be deleted as a result of the control independence discussed at the beginning of the chapter. This is not possible in superblock scheduling. It is possible in hyperblock scheduling, but only if the code is completely if-converted. Thus treegion scheduling can achieve levels of speculation similar to hyperblock scheduling without the need for predication.

Cyclic Scheduling

Cyclic scheduling efficiently schedules loops to achieve high parallelism. An example loop is shown in Fig. 11(a). The first step in most cyclic scheduling algorithms is to unroll the loop. An unrolled version of the loop is shown in Fig. 11(b). Here the body of the loop has been replicated four times. It can then be scheduled using any of the acyclic scheduling algorithms described above. However, any operations at the bottom of the loop cannot be overlapped with operations at the top of the loop. This can only be resolved by completely unrolling the loop (which is not always possible, depending on the conditions for looping).

```
Loop: Load R1 ← M[R2++]
      R3 ← R1 * 2
      Store M[R4++] ← R3
      branch to Loop if R4 < 100
```

```
Loop: Load R1 ← M[R2++]
      R3 ← R1 * 2
      Store M[R4++] ← R3
      exit loop if R4 ≥ 100
      Load R1 ← M[R2++]
      R3 ← R1 * 2
      Store M[R4++] ← R3
      exit loop if R4 ≥ 100
      Load R1 ← M[R2++]
      R3 ← R1 * 2
      Store M[R4++] ← R3
      exit loop if R4 ≥ 100
      Load R1 ← M[R2++]
      R3 ← R1 * 2
      Store M[R4++] ← R3
      branch to Loop if R4 < 100
```

Figure 11. An example-to-illustrate cyclic scheduling. (a) original loop; (b) loop unrolled four times for scheduling.

If the loop is unrolled completely, as shown in Fig. 12, a pattern emerges. This pattern is evident in the boxed iterations between the heavy lines. This repeating pattern is termed the *kernel* of the loop. The loop can be rewritten using the kernel. This is shown in Fig. 13, where each line corresponds to operations that may be executed in the same cycle (specifics of the VLIW encoding have been omitted for clarity). The boxed region is the kernel of the loop. Figure 13 shows the loop rewritten in a compact form. Note that when this loop is scheduled in this way then executed, the effect is as though the loop was unrolled completely. The boxed central region of the loop is identical to that of Fig. 12. The MultiOps above the kernel are termed the *prologue,* and those after the kernel are termed the *epilogue.* If each iteration of the loop is viewed as a single macro instruction, this kind of cyclic scheduling is equivalent to a pipelined execution of these macroinstructions. In this case, the pipeline is composed of three stages: stage one performs a *load* operation, stage two performs the *multiplication,* and stage three performs the *store* of the results. The prologue loads the pipeline with the macroinstructions; they are then performed in an overlapped fashion. When the end of the loop is near, the pipeline is drained by the epilogue. Because the pipeline does not exist in hardware, but rather is a construct of the compiler, this kind of cyclic scheduling is sometimes termed *software pipelining.* Because multiple iterations are executed at once, it is also referred to as *polycyclic scheduling.*

There were several conditions that make polycyclic scheduling of the above example quite simple. These included the constant upper bound of 100 iterations, the lack of conditional code in the loop, and the well-behaved register usage patterns and functional unit requirements. When any of these conditions is not present, polycyclic scheduling can become extremely complicated. In many cases, nonconstant upper bound on the number of iterations and conditional code in the loop body can be handled using predicated execution via if-conversion (similar to hyperblock formation, see above). Decoupling cross-iteration dependencies in a loop can be done either by using additional registers or by hardware support, as in the Cydra 5 (11).

An excellent summary of how to implement polycyclic scheduling in a compiler is presented by Rau (23).

Compatibility Between Generations

Although VLIW processors have simplified hardware, their implementations have commercial problems. Because the binary executable file is written in such a way that it can be executed without dependency checking, the executable file can only execute on one generation of the hardware. Any changes in operation latency or the number of functional units would require new scheduling of the operations. Rau proposed performing this scheduling in hardware via a superscalar-like mechanism in a technique he termed *split issue*

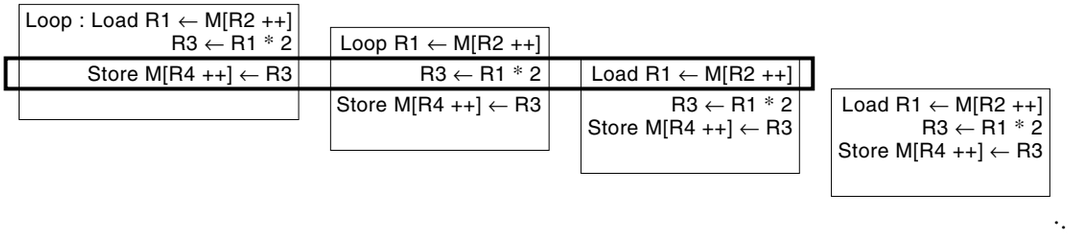

Figure 12. An example of how polycyclic scheduling is derived from unrolling a loop. The region enclosed in heavy lines is the kernel of the loop.

(24). However, Conte and Sathaye (25) proposed a less hardware-centric technique that moves the scheduler from the compiler into the page fault handler of the operating system. This technique (called *dynamic rescheduling*) reschedules code originally scheduled for a different generation of VLIW processor. The scheduler is only invoked on first-time page faults, not when a page is replaced and faulted back into memory. Methods to cache rescheduled pages between program runs can reduce the overhead to near-zero for most code (26). Dynamic rescheduling appears to solve the VLIW compatibility problem in the spirit of VLIW—by employing software to schedule for the hardware.

COMPARISONS AND CONCLUSIONS

Superscalar and VLIW designs both exploit instruction-level parallelism to achieve high performance from a single stream of execution. The techniques each architecture uses to do this however, are vastly different. Superscalar designs handle the challenge of register reuse via methods such as the Tomasulo algorithm. For VLIWs, this same problem is solved by use of a large number of registers, thereby approximating the virtual registers of the compiler's intermediate code. Once registers are renamed, superscalar uses hardware constructs such as the reservation buffer, the scoreboard, the common data bus, and multiple instruction issue to execute instructions out of program order. The VLIW approach uses compiler-based local and global scheduling techniques, the latter including both acyclic (trace-, superblock-, hyperblock- and treegion-scheduling), and cyclic (software pipelining/polycyclic scheduling) techniques. As with dependencies, where superscalar uses a purely hardware-based solution, the VLIW solution is to rely on the compiler. As for speculative execution, superscalars use hardware-based branch predictors. VLIW processors use profile information to construct a weighted control flow graph, which is then scheduled. The analog of superscalar state update hardware (e.g., the reorder buffer, future file, or checkpoint-repair) is the VLIW sentinel scheduling technique. Both aid in speculative execution. VLIWs also take advantage of predicated execution via if-conversion.

Which is better, then, a VLIW or a superscalar processor? Compiler-based scheduling is superior to hardware scheduling techniques alone, since it can consider the entire program rather than the contents of the reservation buffer. However, often old executables cannot be recompiled to take advantage of new compiler scheduling techniques. In such situations, hardware scheduling has an advantage. In addition, the scheduling techniques used for VLIWs are not limited to VLIWs. A superscalar with simple interlocking can be viewed as a "forgiving VLIW," where it correctly executes unscheduled code, but can achieve more substantial speedups for scheduled code. What separates VLIW from superscalar is the programmer's view of the processor. The latencies of functional units in a superscalar processor are not part of the instruction set architecture. For a VLIW, a programmer or compiler must know the latencies to correctly schedule code. This turns into an advantage for a VLIW, since known latencies result in accurate and highly parallel code schedules.

In 1997, Hewlett-Packard and Intel announced details of their new *EPIC* instruction set. From the descriptions, it appears that EPIC is VLIW-like. Each "bundle" in EPIC holds three operations. Bundles can be connected into larger groupings of independent operations. Although no details of the first implementation were given, the announcement is seen by many to forecast the introduction of another commercial VLIW processor by the end of the century. It may be too early to declare that VLIW is the clear winner over superscalar. Industry is likely to mix both ideas into a hybrid that supports a degree of code compatibility in hardware, with a very wide instruction word interface for the compiler to exploit.

Today, the majority of all processors used in systems ranging from personal-computer uniprocessors to massively parallel multiprocessors exploit instruction-level parallelism. It is clear is that instruction-level parallelism will continue to be the most common and most general form of parallel processing in use.

To Probe Further

The design of superscalar and VLIW processor architectures is an active research topic and most of the ideas are first pre-

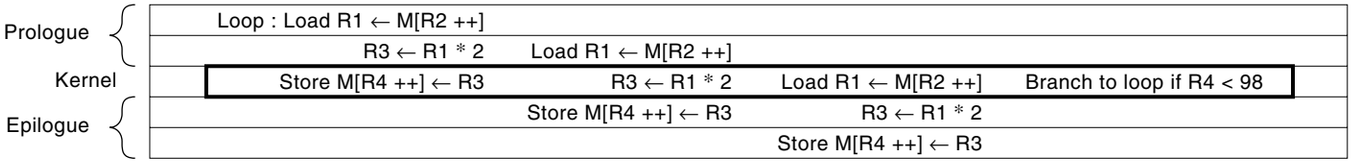

Figure 13. An illustration of the final polycyclic (software pipeline) scheduled loop.

sented at annual conferences. A leading conference is the *International Symposium on Microarchitecture,* which is held every year and organized by the *Association for Computing Machinery SIGMICRO* special interest group and the *IEEE TC-MICRO* technical committee. Many of the ideas discussed in this chapter were first presented at this conference. Other conferences include the *International Symposium on Computer Architecture* and the *International Conference on Architectural Support for Programming Languages and Operating Systems,* both organized by the *ACM SIGARCH* special interest group and the *IEEE Computer Architecture Technical Committee.* In addition, readers interested in VLIW should also examine the journal *Software Practice & Experience,* and the *Conference on Programming Language Design and Implementation* (organized by the *ACM SIGPLAN* special interest group). The proceedings of the above conferences are published by the IEEE and the ACM.

BIBLIOGRAPHY

1. D. Kroft, Lockup-free instruction fetch/prefetch cache organization, *Proc. 8th Annu. Int. Symp. Comput. Archit.,* 1981, pp. 81–87.
2. J. E. Smith, A study of branch prediction strategies, *Proc. 8th Annu. Int. Symp. Comput. Archit.,* 1981, pp. 135–148.
3. T. M. Conte et al., Optimization of instruction fetch mechanisms for high issue rates, *Proc. 22nd Annu. Int. Symp. Comput. Archit.,* 1995, pp. 333–344.
4. E. Rotenberg et al., A low latency approach to high bandwidth instruction fetching, *Proc. 29th Annu. Int. Symp. Microarchit.,* 1996, pp. 24–34.
5. R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM J. Res. Develop.,* **11**: 34–53, 1967.
6. J. E. Smith and A. Pleszkun, Implementation of precise interrupts in pipelined processors, *Proc. 12th Annu. Int. Symp. Comput. Archit.,* 1985.
7. M. Johnson, *Superscalar Microprocessor Design,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
8. W. W. Hwu and Y. N. Patt, Checkpoint repair for high-performance out-of-order execution machines, *IEEE Trans. Comput.,* **C-36**: 1496–1514, 1987.
9. J. A. Fisher et al., Parallel processing: A smart compiler and a dumb machine, *Proc. 1984 SIGPLAN Symp. Compiler Construct.,* 1984, pp. 37–47.
10. T. M. Conte et al., Instruction fetch mechanisms for VLIW architectures with compressed encodings, *Proc. 29th Annu. Int. Symp. Microarchit.,* 1996, pp. 201–211.
11. B. R. Rau et al., The Cydra 5 departmental supercomputer, *Computer,* **22**: 12–35, 1989.
12. E. G. Coffman, *Computer and Job-Shop Scheduling Theory,* New York: Wiley, 1976.
13. J. A. Fisher, Trace scheduling: A technique for global microcode compaction, *IEEE Trans. Comput.,* **C-30**: 478–490, 1981.
14. W. W. Hwu et al., The Superblock: An effective structure for VLIW and superscalar compilation, *J. Supercomput.,* **7**: 229–248, 1993.
15. S. A. Mahlke, *Exploiting instruction level parallelism in the presence of branches,* Ph.D. dissertation, Depart. Electri. Comput. Eng., Univ. Illinois, Urbana-Champaign, Urbana, IL, 1996.
16. W. A. Havanki, S. Banerjia, and T. M. Conte, Treegion scheduling for wide-issue processors, *Proc. 4th Int. Symp. High-Performance Comput. Archit.* (HPCA-4), Las Vegas, 1998.
17. J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures,* Cambridge, MA: MIT Press, 1986.
18. P. G. Lowney et al., The Multiflow Trace scheduling compiler, *J. Supercomput.,* **7**: 51–142, 1993.
19. P. P. Chang et al., IMPACT: An architectural framework for multiple-instruction-issue processors, *Proc. 18th Annu. Int. Symp. Comput. Archit.,* 1991, pp. 266–275.
20. S. A. Mahlke et al., Sentinel scheduling: A model for compiler-controlled speculative execution, *ACM Trans. Comput. Syst.,* **11**: 376–408, 1993.
21. J. R. Allen et al., Conversion of control dependence to data dependence, *Proc. 10th Annu. ACM Symp. Principles Programming Languages,* 1983, pp. 177–189.
22. S. A. Mahlke et al., A comparison of full and partial predicated execution support for ILP processors, *Proc. 22nd Annu. Int. Symp. Comput. Archit.,* 1995, pp. 138–150.
23. B. R. Rau, Iterative modulo scheduling: An algorithm for software pipelining loops, *Proc. 27th Annu. Int. Symp. Microarchit.,* 1994, pp. 63–74.
24. B. R. Rau, Dynamically scheduled VLIW processors, *Proc. 26th Annu. Int. Symp. Microarchit.,* 1993, pp. 80–90.
25. T. M. Conte and S. W. Sathaye, Dynamic rescheduling: A technique for object code compatibility in VLIW architectures, *Proc. 28th Annu. Int. Symp. Microarchit.,* 1995, pp. 208–218.
26. T. M. Conte, S. W. Sathaye, and S. Banerjia, A persistent rescheduled-page cache for low-overhead object-code compatibility in VLIW architectures, *Proc. 29th Annu. Int. Symp. Microarchit.,* 1996, pp. 4–14.

THOMAS M. CONTE
North Carolina State University