Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers

GURINDAR S. SOHI, MEMBER, IEEE

Abstract—**The performance of pipelined processors is limited by data dependencies and branch instructions. In order to achieve high performance, mechanisms must exist to alleviate the effects of data dependencies and branch instructions. Furthermore, in many cases, for example the support of virtual memory, it is essential interrupts be precise. In multiple functional unit pipelined processors where the instructions can complete and update the state of the machine out of program order, hardware support must be provided to implement precise interrupts. In this paper, we combine the problems of data dependency resolution and precise interrupt implementation. We present a design for a hardware mechanism that resolves dependencies dynamically and, at the same time, guarantees precise interrupts. Simulation studies show that, by resolving dependencies, the proposed mechanism is able to obtain a significant speedup over a simple instruction issue mechanism as well as implement precise interrupts.**

Index Terms—**Dependency resolution, multiple functional units, out-of-order execution, pipelined computers, precise interrupts, register update unit, Tomasulo's algorithm.**

I. Introduction

THE CPU's of most supercomputers consist of several pipelined functional units connected together in some fashion. Such multiple functional unit, pipelined machines are able to achieve a considerable overlap in the execution of instructions. Unfortunately, pipelined CPU's have two major impediments to their performance: 1) *data dependencies* and 2) *branch instructions.* An instruction cannot begin execution until its operands are available. If an instruction is dependent upon a previous instruction, the instruction must wait until the previous instruction has completed execution. This waiting can degrade performance. The performance degradation due to branch instructions can be even more severe. Not only must a conditional branch instruction wait for the branch condition to be known, an additional penalty may be incurred when fetching an instruction from the taken branch path to the stage where the instruction is decoded and issued.

Pipelined CPU's suffer from another major problem—an interrupt can be *imprecise* [3], [12], [24]. This problem is especially severe in multiple functional unit computers in which instructions can complete execution out of program order even though they are issued in program order [1], [3], [21]. For a high-performance, pipelined CPU, an adequate solution must be found for the imprecise interrupt problem and means must be provided for overcoming the performance degradation due to data dependencies and branch instructions.

The detrimental effects of branch instructions can be alleviated by using *delayed* branch instructions. However, the utility of delayed branch instructions is limited for long pipelines. In such cases, other means must exist to alleviate the detrimental effects. A common approach is to use *branch prediction* [13], [22]. Using prediction techniques, the probable execution path of a branch instruction is determined. Instructions from the predicted path can then be fetched into instruction buffers or even executed in a *conditional mode* [3], [4], [7], [14], [19]. While the conditional mode of execution will generally result in a higher pipeline throughput, a mechanism to allow the machine to recover from an incorrect sequence of conditionally executed instructions must be provided.

Both hardware and software solutions exist to the data dependency problem. Software solutions use code scheduling techniques (combined with a large set of registers) to increase the distance between dependent instructions and to provide interlocks [6]. Most hardware solutions employ some form of *waiting stations* where an instruction can wait for its operands and allow subsequent instructions to proceed, thereby allowing instructions to issue out of program order. Examples of waiting stations include the *reservation stations* of the IBM 360/91 floating point unit [26] and the node tables of the HPS microarchitecture [17]. The waiting stations form the core of a *dependency-resolution* mechanism that must exist in order to preserve program dependencies. In this paper, a dependency-resolution mechanism is synonymous with an out-of-order instruction issue mechanism. Note the difference between out-of-order instruction issue (also called out-of-order instruction execution) and out-of-order instruction completion. Instructions can complete out of program order even though they were issued in program order.

In a pipelined machine, imprecise interrupts can be caused by instruction-generated traps such as arithmetic exceptions and page faults. An imprecise interrupt can leave the machine in an irrecoverable state. While the occurrence of arithmetic exceptions is rare, the occurrence of page faults in a ma-

Manuscript received August 8, 1987; revised July 25, 1989. This work was supported in part by the University of Wisconsin Graduate Research Committee and in part by NSF Grant CCR-8706722. A preliminary version of this paper appeared in the 14th International Symposium on Computer Architecture, Pittsburgh, PA, June 1987.

The author is with the Computer Sciences Department, University of Wisconsin, Madison, WI 53706.

IEEE Log Number 8932910.

0018-9340/90/0300-0349$01.00 © 1990 IEEE

chine that supports virtual memory is not. Therefore, if virtual memory is to be used with a pipelined CPU, it is crucial that interrupts be precise. Several hardware solutions to the problem are described in [24] and in [8]. We are unaware of any software solutions to the imprecise interrupt problem for multiple functional unit computers. A software solution will be extremely difficult, if not impossible. Not only must the software allow for the worst case execution time for any instruction, it must also keep track of instructions that have completed out of program order and generate an appropriate code sequence to undo the effects of those instructions. In any case, some hardware support must be provided to maintain run-time information.

The problems of out-of-order instruction issue and imprecise interrupts have been considered independent of one another by many researchers [2], [8], [24], [26], [27]. The solutions provided thus far attack each problem individually. For example, a recent microarchitecture, HPS, uses *register alias tables* and *node alias tables* to permit out-of-order instruction issue [8], [17], [18]. To provide precise interrupts, HPS uses a *checkpoint repair* mechanism [9], [10]. In this paper, we treat the problems of out-of-order instruction issue and imprecise interrupts simultaneously. If interrupts are to be precise, some hardware support is needed. In its simplest form, a precise-interrupt mechanism will aggravate dependencies [24]. Why not combine a simple mechanism that implements precise interrupts with an out-of-order instruction issue mechanism so that the aggravated dependencies (as well as other dependencies) can be tolerated?

The remainder of this paper is as follows. In Section II, we describe the model architecture that we use throughout this paper. In Section III, we discuss Tomasulo's out-of-order instruction issue algorithm and extend it, giving several variations, so that the cost of implementing it using discrete components is not very high even for a large number of registers. In Section IV, we discuss the problem of imprecise interrupts and review known solutions. Section V describes a unit, the register update unit (RUU), that resolves dependencies as well as implements precise interrupts. The precise interrupt and out-of-order instruction issue mechanisms mutually aid and simplify each other. An evaluation of the RUU is carried out in Section VI. Finally, we discuss how our mechanism can be used to alleviate the degradation due to branch instructions.

II. Model Architecture

The model architecture that we use for our studies is presented in Fig. 1. It has the same capabilities and executes the same instruction set as the scalar unit of the CRAY-1 [5], [21]. The CRAY-1 was chosen because it represents a state-of-the-art scalar unit and its execution can be modeled precisely. The author also had easy access to tools that could be used to generate instruction traces for the CRAY-1 scalar unit [16]. There are a few differences between the CRAY-1 scalar unit and our model architecture. First, in our model architecture, all instructions, whether they are composed of 1 parcel (16 bits) or 2 parcels (32 bits) can issue in a single cycle if issue conditions are favorable. Next, only one function can output data onto the result bus in any clock cycle. In contrast, the CRAY-1 scalar unit has separate result buses for the address and scalar functional units. Instructions are fetched by the *instruction fetch unit* and decoded and issued by the *decode and issue unit*. Once dependencies have been resolved in the decode and issue unit, instructions are forwarded to the functional units for execution. The results of the functional units are written directly into the register file. The register file consists of 8 A, 8 S, 64 B, and 64 T registers. In this paper, we shall focus on an issue unit that is capable of issuing only one instruction per clock cycle. Extensions to this work to allow the issue of multiple instructions per clock cycle can be found in [20].

A. Benchmark Programs

The benchmark programs used throughout this paper were the first 14 Lawrence Livermore loops [15]. The first 14 loops were chosen because they were readily available and also allow us to compare our results to previous studies that tackle similar problems [24], [27]. Henceforth, we shall refer to them as LLL1, LLL2, $\cdots$, LLL14. The simulations were carried out as follows. The benchmark programs, as compiled by the CFT compiler for the scalar unit, were fed into a CRAY-1 simulator [16]. The CRAY-1 simulator generates an instruction trace for each program. Vector instructions are not used. Each instruction trace was then fed into the appropriate simulator.

B. Simulation of the Model Architecture

We simulated the execution of the benchmark programs on the model architecture of Fig. 1. The number of instructions executed, the number of clock cycles taken for the execution of each benchmark program, and the number of instructions executed per cycle is given in Table I. In generating the results of Table I, we assumed that: 1) no memory bank conflicts occur, 2) all instruction references are serviced by the instruction buffers, and 3) the instructions are already present in the instruction buffers when the program is started. These assumptions do not affect the execution time considerably for the benchmark programs. These assumptions and a difference in the bus structure account for the difference between the data presented in Table I and in [27]. The instruction issue rate is the average number of instructions that are executed in a cycle, i.e., the total number of instructions executed in the benchmark divided by the total number of cycles to execute the benchmark. The instruction issue rate for the total of all 14 loops is calculated as the harmonic mean of the individual issue rates [23]. For reasons of brevity, we shall present all subsequent simulation results as a harmonic mean of all 14 loops rather than report the results for each individual loop.

As we can see from Table I, the performance of the model machine is far from the issue limit of 1 instruction per cycle. From our simulations, we determined that the main reason for this suboptimal performance is data dependencies. Therefore, we must find some way of alleviating the affects of data dependencies. We have two choices: 1) eliminating the dependencies or 2) tolerating the dependencies. Data dependencies can be eliminated by software code scheduling techniques. Hardware dependency resolution techniques allow the machine to tol-

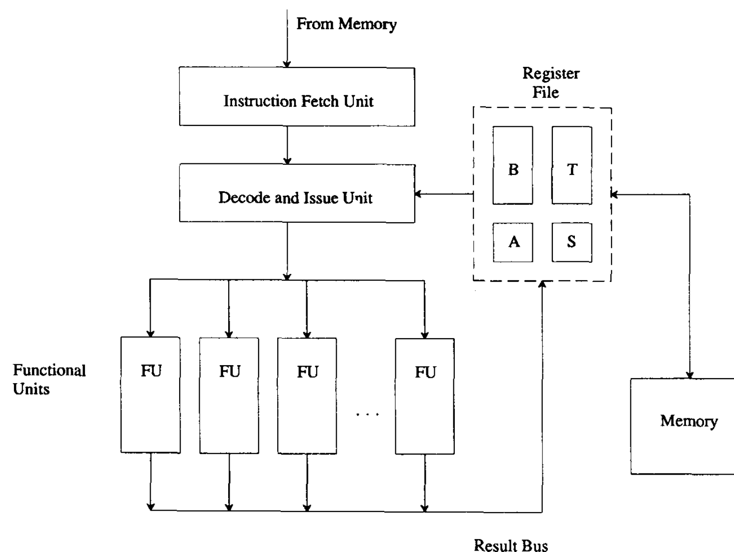


Fig. 1. The model architecture.

TABLE I
STATISTICS FOR THE BENCHMARK PROGRAMS

Benchmark Program	Instructions Executed	Clock Cycles to Execute Program	Instruction Issue Rate
LLL1	7217	17234	0.419
LLL2	8448	17102	0.494
LLL3	14015	36023	0.389
LLL4	9783	20643	0.474
LLL5	8347	20696	0.403
LLL6	9350	22034	0.424
LLL7	4573	10231	0.447
LLL8	4031	8026	0.502
LLL9	4918	10134	0.485
LLL10	4412	9420	0.468
LLL11	12002	28002	0.429
LLL12	11999	27991	0.429
LLL13	8846	17814	0.497
LLL14	9915	23573	0.421
Harmonic Mean			0.446

erate dependencies. Since we are mainly concerned with a hardware mechanism that allows the architecture to tolerate dependencies as well as implement precise interrupts, we can restrict our attention to hardware mechanisms for tolerating dependencies.

III. HARDWARE DEPENDENCY RESOLUTION

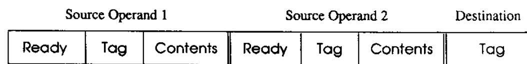
When an instruction reaches the decode and issue stage in the pipeline, checks must be made to determine if the operands for the instruction are available, i.e., if all dependencies for this instruction have been resolved. If an operand is not available, the instruction must wait in the decode and issue stage. Because the decode and issue stage of the pipeline is busy, subsequent instructions cannot proceed even though they may be ready to execute. Subsequent instructions can proceed if the waiting instruction "steps aside," thereby freeing the decode and issue stage and allowing other instructions to bypass the waiting instruction. In order to do so, some form of waiting stations or *reservation stations* must be provided [26]. Other mechanisms also exist in the literature [2]. Since our work is based on the concept of reservation stations, we shall focus our attention on mechanisms that employ reservation stations in some form.

A. Tomasulo's Algorithm

Tomasulo's hardware dependency-resolution (or out-of-order instruction issue) algorithm was first presented for the floating point unit of the IBM 360/91 [26]. Extensions of this algorithm for the CRAY-1 scalar unit are presented in [27] and for the HPS microarchitecture in [8]. The algorithm operates as follows. An instruction whose operands are not available when it enters the decode and issue stage is forwarded to a *reservation station* (*RS*) associated with the functional unit that it will be using. It waits in the RS until its data dependencies have been resolved and its operands are available. Once at a reservation station, an instruction can resolve its dependencies by monitoring the common data bus (the result bus in our model architecture). When all the operands for an instruction are available, it is dispatched to the functional unit for execution. The result bus can be reserved either when the instruction is dispatched to the functional unit [27] or before it is about the leave the functional unit [26].

Each source register is assigned a *busy bit*. A register is busy if it is the destination of an instruction that is still in execution. Each destination register (also called a *sink register*) is assigned a tag which identifies the result that will be written into the register. Since any register in the register file can be a destination register, each register must be assigned a tag. The fields in each reservation station are shown in Fig. 2.

If a source register is busy when the instruction reaches the issue stage, the tag for the source register is obtained and the instruction is forwarded to a reservation station. The appropriate ready bit in the reservation station is set to indicate that the source operand is unavailable. If the source register is not busy, the contents of the register are read into the reservation station and the ready bit is reset to indicate that



Source Operand 1			Source Operand 2			Destination
Ready	Tag	Contents	Ready	Tag	Contents	Tag

Fig. 2. A reservation station in Tomasulo's algorithm.

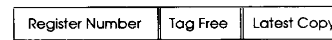
the source operand is available. The instruction obtains a tag for the destination register, updates the old tag of the destination register, and proceeds to a reservation station. When all source operands are available in a reservation station, the instruction is dispatched to the functional unit for execution and the reservation station is released for reuse by future instructions. Memory is treated as a special functional unit. When an instruction has completed execution, the result (along with its tag) appears on the result bus. The registers as well as the reservation stations monitor the result bus and update their contents (and ready/busy bits) when a matching tag is found. Details of the algorithm can be found in [26] and [27].

While this algorithm is straightforward and effective, it can be expensive to implement especially using discrete components since each register needs to be tagged and each tag needs associative comparison hardware to carry out the tag-matching process. This may not be practical if the number of *possible destination registers*, i.e., the number of registers is large. For our model architecture which has 8 *A*, 8 *S*, 64 *B*, and 64 *T* registers, an implementation of this dependency-resolution mechanism for all the registers would require 144 tag-matching units. The use of such a large number of hardware units may not be practical in many technologies.

B. Extensions to Tomasulo's Algorithm

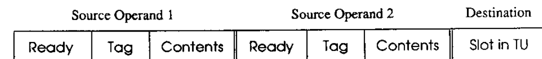
1) A Separate Tag Unit: On closer inspection, we see that very few of all *possible* destination registers may actually be active, i.e., be waiting for a result, at any given time. Therefore, if we associate a tag with *each possible* destination register, a lot of associative tag-matching hardware will be idle at any given time. Why not have a common tag pool and assign a tag only to a *currently active* destination register rather than associating a tag with each possible destination register? In Tomasulo's algorithm, a currently active register is one whose busy bit is on.

We consolidate the tags from all *currently active* registers into a *tag unit* (*TU*). Each register has only a single busy bit. At instruction issue time, if a source register is busy, the TU is queried for the current tag of the source register and the tag is forwarded to the reservation stations. A new tag is obtained for the destination register of the instruction. If the destination register is not busy, acquiring such a tag from the TU is straightforward. If the destination register is busy, i.e., the TU already holds a tag for the register, a new tag is obtained and the instruction holding the old tag is informed that, while it may update the register, it may not *unlock* the register, i.e., clear the busy bit when it completes execution. In order to do so, we associate a bit (the latest copy bit) with each TU entry. This bit indicates whether the tag is the latest tag for the register and if the instruction holding the tag has a *key* to *unlock* the register and clear the busy bit. Instruction issue blocks if no tag can be obtained, i.e., the TU is full. The fields of each entry in the TU are shown in Fig. 3. The reservation stations are modified so that the result can



Register Number	Tag Free	Latest Copy

Fig. 3. An entry in the tag unit.



Source Operand 1			Source Operand 2			Destination
Ready	Tag	Contents	Ready	Tag	Contents	Slot in TU

Fig. 4. A reservation station with a tag unit.

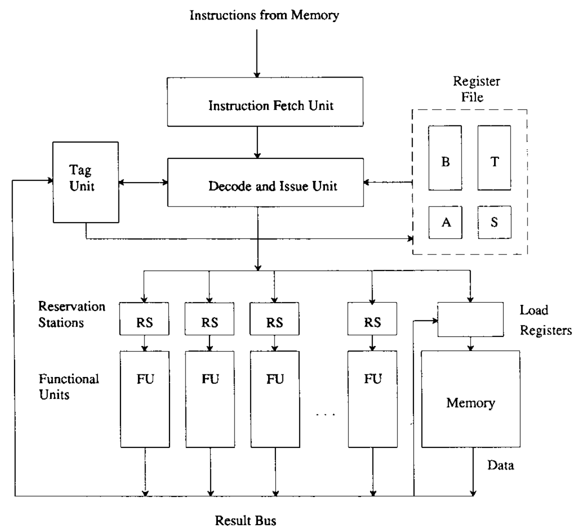


Fig. 5. The model architecture with a tag unit and distributed reservation stations.

be forwarded to the appropriate slot in the TU. The fields in the modified reservation stations are shown in Fig. 4.

As before, the instruction along with its associated tags/operands is forwarded to a reservation station where it waits for its operands to become ready. The result from a functional unit (along with its tag) is broadcast to all reservation stations and is also forwarded to the TU. Reservation stations monitor the result bus and gate in the result if the tag of the data on the result bus matches the tag stored in the reservation station. The TU forwards the result to the register specified in the appropriate slot of the TU. All registers are, therefore, updated only by the TU when their data are available and no direct connection is needed between the functional units and the register file. When the register has been updated by the TU, the corresponding tag is released and is marked free in the TU. The modified architecture that incorporates a tag unit and reservation stations associated with each functional unit is shown in Fig. 5.

a) Example: The operation of the tag unit is best illustrated by an example. Consider a TU that has six entries as shown in Fig. 6. Each entry in the TU has a bit indicating if the tag is free (tag free), i.e., available for use by the issue logic, a bit indicating if the tag is the latest tag for the register (latest copy), and a field for the number of the destination register (register number) as in Fig. 3. The TU is indexed by the tag number.

Consider the execution of an instruction I_1 that adds the contents of registers *S*0 and *S*7 and puts the result in *S*4. Assume that the state of the TU is as shown in Fig. 6 and that

Tag Number	Register Number	Tag Free	Latest Copy
1	A0	No	Yes
2	S0	No	Yes
3	NIL	Yes	Yes
4	S4	No	Yes
5	S0	No	No
6	S3	No	Yes

Fig. 6. A tag unit with six entries.

S7 is free (indeed a register must be free if it does not have an entry in the TU). When the issue logic decodes I_1, it attempts to get a *new* tag for the destination register S4 from the TU and obtains tag 3. Since the TU already has a tag for S4, the old tag (4) is updated to indicate that it no longer represents the latest copy of the register. Since S7's contents are valid, they can be read from the register file and forwarded to the reservation stations directly. However, since the contents of S0 are not valid, the latest tag for S0 (tag 2) must be obtained from the TU. The issue unit forwards a packet to the reservation station associated with the add functional unit. The packet contains the contents of S7, a tag (2) for S0 and a tag (3) for the destination register S4. I_1 waits in the reservation station until that tag 2 appears on the result bus. At this point, the reservation station reads the value for S0 and I_1 is ready to execute. When I_1 completes execution and leaves the add functional unit, the result is forwarded to all reservation stations that have a matching tag (3) and also to the TU. The TU forwards the result to the register file to be written into S4. Since tag 3 is the latest tag for S4, S4's busy bit can be reset when the data have been written into S4. Tag 3 is then marked free and is available for reuse by the issue logic.

b) Interactions with Memory: Load/store operations that interact with memory pose a challenge to architectures that allow out-of-order instruction issue (the reader is referred to [18] for a discussion of and some solutions to the problem). In our model, we handle memory dependencies in a fashion similar to the way register dependencies are handled in the TU. A set of *load registers* contains the addresses of "currently active" memory locations. Each load register has tags to allow for multiple instances of a memory address just as the TU allows multiple instances of registers.

The reservation stations associated with the memory functional unit are managed in a pseudoqueue fashion to satisfy dependencies. A load operation needs a memory address before it can be issued to the memory whereas a store operation needs both a memory address and a data value. If the address of a load/store operation is unavailable, subsequent load/store instructions are not allowed to proceed. This prevents a possible violation of dependencies.

When the memory address required by the operation is known, checks are made to see if the address matches an address in the load registers. A match indicates that there is a pending operation to the same memory address. If no match results, a free load register is obtained. Instruction issue is blocked if no free load register is available.

If the current operation is a load operation and a match results, the load operation need not be submitted to memory. This is because the pending operation to the same address can also satisfy the load operation. In this case, the tag of the appropriate load register is returned to the reservation station. If there is no pending request to the same address, the tag is returned to the reservation station and the load operation is submitted to the memory. In either case, the load operation completes when a matching tag appears on the result bus.

If the current operation is a store operation and a match results, the tag of the load register is updated and the tag returned to the reservation station. By doing so, a new instance of the memory location is provided. If no match results, a free load register is obtained and the tag returned to the reservation station. When the data for the store operation are available, they are forwarded (along with the tag) via the load registers to the memory and the store operation is complete.

When the load/store operation is complete, the reservation station is freed. The corresponding load register is also freed if the tags match, i.e., there is no pending operation to the same memory address. Note that the above scheme allows load operations to bypass store operations as long as the addresses of all the operations are known. Also note that the load registers need to be searched associatively. However, for a small number of load registers, this associative search is not very wide.

2) Merging the Reservation Stations: If each functional unit has a separate set of reservation stations, it is likely that some functional unit will run out of reservation stations while the reservation stations associated with another functional unit are idle. As suggested in [27], we can combine all the reservation stations into a common *RS pool* rather than having disjoint pools of reservation stations associated with each functional unit. All instructions that were previously issued to distributed reservation stations associated with the functional units now go to the common RS pool. Instruction issue is blocked if the RS pool is full. As instructions become ready in the RS pool, they are issued to the functional units. All the other functions are as before.

3) Merging the RS Pool and the Tag Unit: In the tag unit, there is one entry for every instruction that is present in either the RS pool or in the functional units. Therefore, at any time, there is a one-to-one correspondence between the entries in the TU and the instructions in the reservation stations or the functional units. This suggests that we can combine the RS pool and the tag unit into a single *RS tag unit* (*RSTU*). Of course, a reservation station is wasted if it is associated with an instruction that is in a functional unit. However, as we shall see in Section V, this organization can easily be extended to allow for the implementation of precise interrupts.

In the RSTU, a reservation station is reserved at the same time that a tag is reserved. When an instruction issues, it obtains a tag from the RSTU and in doing so automatically reserves a reservation station. All the other functions, including interactions with the memory, are as before. The architecture with an RSTU is shown in Fig. 7 and an entry in the RSTU is shown in Fig. 8. Since the reservations stations are merged, a functional unit field is needed to identify the functional unit to which the instruction occupying the RSTU entry will be issued.

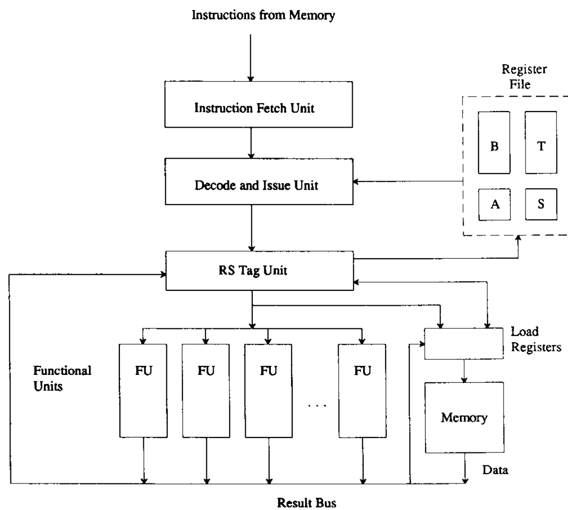


Fig. 7. The model architecture with an RSTU.

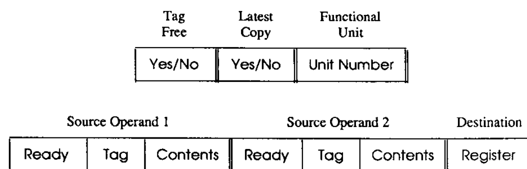


Fig. 8. An entry in the RSTU.

TABLE II
RELATIVE SPEEDUP AND ISSUE RATE WITH AN RSTU

Number of Entries in RSTU	Relative Speedup	Instruction Issue Rate
3	0.984	0.439
4	1.157	0.516
5	1.312	0.585
6	1.429	0.637
7	1.483	0.661
8	1.547	0.690
9	1.593	0.710
10	1.634	0.723
15	1.721	0.768
20	1.748	0.780
25	1.773	0.791

a) Simulation Analysis of the RSTU: In order to evaluate the effectiveness of the RSTU, we carried out a simulation analysis of the RSTU using the first 14 Lawrence Livermore loops as a benchmark. The results obtained for the execution of all 14 loops are presented in Table II. The relative speedup is the speedup compared to the simple instruction issue mechanism of Table I and the instruction issue rate is the harmonic mean of the individual issue rates. The number of load registers in these simulations was six. This guarantees that, for our benchmark programs, instruction issue is never blocked because of an unavailable load register.

From Table II, it is quite clear that the RSTU is able to achieve a significant speedup over a simple instruction issue mechanism with a reasonable amount of hardware. The RSTU is also quite close to achieving the issue limit of 1 instruction per clock cycle for our model architecture. Indeed, all non-branch instructions are able to achieve the limit of 1 instruction per cycle. The only cycles in which no useful instruction is executed are the dead cycles following each branch instruction. The degradation due to such cycles could be reduced by using delayed branch instructions or by conditionally executing instructions. The results presented in Table II compare favorably to the results presented in [27]. Because the RSTU can implement the dependency-resolution mechanism for the B and T register files, it can achieve a better speedup than a mechanism that is somewhat restricted as in [27].

TABLE III
RELATIVE SPEEDUP AND ISSUE RATE WITH AN RSTU AND TWO DATA PATHS

Number of Entries in RSTU	Relative Speedup	Instruction Issue Rate
3	0.991	0.442
4	1.174	0.524
5	1.330	0.593
6	1.447	0.645
7	1.523	0.679
8	1.580	0.701
9	1.624	0.724
10	1.656	0.739
15	1.747	0.779
20	1.772	0.790
25	1.783	0.795

At first glance, it may seem that an organization with merged reservation stations (such as the RSTU of Fig. 8) is at a disadvantage when compared to an organization with distributed reservation stations (such as Fig. 5) since only one instruction can issue from the reservation stations to the functional units in a clock cycle unless multiple paths are provided between the RSTU and the functional units. On the other hand, a better use of the reservations stations results since the reservation stations can be shared among several functional units. In order to evaluate the effectiveness of multiple data paths between the RSTU and the functional units, we simulated an architecture with two paths from the RSTU to the functional units, but only a single issue unit, a single result bus, and single path from the RSTU to the register file. The results are presented in Table III.

As is evident from Table III, the presence of a duplicate path from the RSTU to the functional units makes little difference. This result is not counterintuitive. We use an argument based on instruction flow to convince the reader. The RSTU is essentially a reservoir of instructions that is filled by the decode and issue logic and drained by the functional units. Since the decode and issue logic can fill this reservoir at a maximum rate of 1 instruction per cycle, having a drain that is capable of draining more than 1 instruction per cycle will not be very useful in a steady state. Of course, if the decode and issue unit itself could submit more than 1 instruction per clock cycle to the RSTU, additional paths from the RSTU to the functional units would be needed [20].

IV. IMPLEMENTATION OF PRECISE INTERRUPTS

We now address the issue of precise interrupts. A complete description of several schemes that implement precise interrupts is given in [24]. An alternate scheme that uses checkpoint repair is presented in [10].

The mechanisms described in [24] include a simple *reorder buffer*, a more complex reorder buffer with *bypass logic*, a *history buffer*, and a *future file*. The simple reorder buffer allows instructions to finish execution out of order but up-

dates the state of the machine, i.e., *commits* the instructions, in the order that the instructions arrived at the decode and issue stage. This ensures that a precise state of the machine is recoverable at any time. However, by forcing an ordering of commitment among the instructions, the reorder buffer aggravates data dependencies. This is because the value of a register cannot be read until it has been updated by the reorder buffer, even though the instruction that computed a value for the register may have already completed and the new value is in the reorder buffer. If bypass logic is associated with the reorder buffer, an instruction does not have to wait for the reorder buffer to update a source register; it can fetch the value from the reorder buffer (if it is available) and can issue. With a bypass mechanism, the issue rate of the machine is not degraded considerably if the size of the buffer is reasonably large [24]. However, a bypass mechanism is expensive to implement since it requires a search capability and additional data paths for each buffer entry. A history buffer has the same performance as a reorder buffer with bypass logic. It does not need bypass logic but the register file needs another read port. A future file achieves the same performance as a reorder buffer with bypass logic at the expense of duplicating the entire register file. The checkpoint repair mechanism described in [10] maintains three copies of the register file. We shall not discuss these mechanisms in more detail in this paper. The interested reader is referred to the original papers.

V. Merging Dependency Resolution and Precise Interrupts

We note that the RSTU of Section III-B3 can be modified to behave like a reorder buffer if it is forced to update the state of the machine in the order that the instructions are encountered by the decode and issue unit. This is easily accomplished by managing the RSTU as a queue. Therefore, all that we have to do to implement precise interrupts in an architecture with an RSTU is to manage the RSTU like a queue. We call the modified logic the *register update unit* (*RUU*). The RUU is essentially the RSTU constrained to commit instructions in the order that the instructions were received by the decode and issue logic (and consequently by the RUU). The functional units remain unchanged. The modified architecture that uses an RUU to execute instructions out of program order and to ensure a precise state of the machine is given in Fig. 9. Let us consider the operation of the RUU in some more detail.

A. The Register Update Unit (RUU)

The RUU performs four major functions in each clock cycle. First, it accepts new instructions from the decode and issue logic. Second, it monitors the result bus to resolve dependencies. Third, it determines which instruction should be issued to the functional units for execution, reserves the result bus, and dispatches the instruction to the selected functional unit for execution. Fourth, it determines if an instruction can commit, i.e., update the registers, and commits the instruction if it can. Below, we see how the RUU accomplishes these tasks.

First, the RUU must accept an instruction from the decode and issue logic. The RUU is managed like a queue using RUU_Head and RUU_Tail pointers. If RUU_Head = RUU_Tail, the RUU is full. RUU_Tail points to the slot that will be used by the decode and issue logic and RUU_Head points to the next instruction that must commit to ensure a precise state. When an instruction is decoded, the issue logic requests an entry in the RUU. If the RUU is full, instruction issue is blocked. If an entry is available, the issue logic obtains the position of the entry (using the RUU_Tail pointer) and updates the RUU_Tail pointer. Simultaneously, it forwards the contents of the source registers (if they are available) or a register tag to the selected reservation station in the RUU.

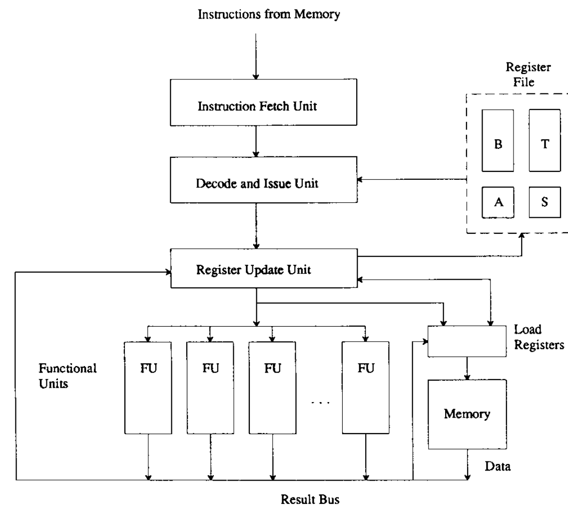


Fig. 9. The model architecture with an RUU.

Managing the RSTU like a queue has a very important side effect—the logic for obtaining tags for source operands and generating tags for destination operands, i.e., for dependency resolution, is greatly simplified. Recall that in the RSTU, the issue logic had to search the RSTU associatively to obtain the correct tag for the source operand and to update the latest copy field for the destination register. If multiple instances of the same destination register are disallowed, i.e., instruction issue is blocked if the destination register is busy, no associative logic is necessary since the register number itself serves as the tag. An *instance* of a register is a new copy of the register. By providing multiple instances of a destination register, the architecture can process several instructions with the same destination register simultaneously, i.e., resolve write-after-write hazards [11]. Disallowing multiple instances of a destination register can degrade performance [27]. As noted in [26], it is possible to eliminate the associative search and use a counter to provide multiple instances and source operand tags for each register *if we can guarantee that results return to the registers in order*. This is precisely the situation in the RUU. The implementation of precise interrupts, therefore, simplifies the out-of-order instruction issue mechanism.

The scheme we use to provide multiple instances of a destination register and to provide source operand tags associates two n-bit counters with each register in the register file (this includes the B and T register files). There is no busy bit. The counters, the *number of instances* (*NI*) and the *latest instance* (*LI*), represent the number of instances of a register in

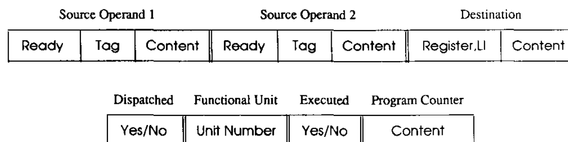


Fig. 10. An entry in the RUU.

the RUU and the number of the latest instance, respectively. When an instruction with a destination register Ri is issued to the RUU, both NI and LI associated with Ri are incremented. LI is incremented modulo n. Up to $2^n - 1$ instances of a register can be present in the RUU at any time; issue is blocked if NI for a destination register is $2^n - 1$. When an instruction leaves the RUU and updates the value of Ri, the associated NI is decremented (since n is small, the incrementing/decrementing process is fast). A register is free if $\mathrm{NI} = 0$, i.e., there is no instruction in the RUU that is going to write into the register.

The register tag sent to the RUU consists of the register number Ri appended with the LI counter. This guarantees that future instructions access the latest instance, i.e., obtain the latest copy of the register contents and that instructions already present in the RUU get the correct version of the data. In our experiments, each of these counters was 3 bits wide. This allowed up to seven instances of a destination register. A 3-bit counter ensured that, for our benchmark programs, an instruction never blocked in the decode and issue stage because an instance of a register was unavailable. Since we had a total of 144 registers, the tag field was 11 $(8 + 3)$ bits wide.

To accomplish its second task of resolving dependencies, the RUU must monitor the result bus. To do so, each source operand field in the RUU has a ready bit, a tag subfield, and a content subfield. If the operand is not ready, the tag subfield monitors the result bus for a matching tag. If a match is detected, the data on the bus are gated into the content field. This task of the RUU corresponds to the task carried out by the reservation stations in Tomasulo's algorithm. Note that there is no need for a latest copy field in the RUU and no associative search logic is needed in the RUU to generate and maintain the tags. However, associative comparison logic is still needed for all the reservation stations in the RUU so that they can gate in the value of source operands when available.

An entry in the RUU is shown in Fig. 10. The dispatched field indicates if the instruction has been dispatched for execution to the functional unit specified in the functional unit field. The executed field indicates if the instruction has finished execution and is ready to update the register file. The program counter field is needed for the implementation of precise interrupts [24]. We have omitted the details of extra information that must be carried around with each instruction since the details of such information are straightforward.

The RUU accomplishes its third task by monitoring the ready bits of the source operands. When the operands of an instruction in the RUU are ready, the instruction can issue to the functional units. The RUU issues the highest priority instruction and sets the dispatched bit to indicate that the instruction has been dispatched for execution and should not be selected again by the dispatching algorithm. Priority is first given to load/store instructions and then to an instruction which entered the RUU earlier. The RUU reserves the result bus when it issues an instruction to the functional units.

The final RUU task of committing an instruction is accomplished by monitoring the executed bit of the RUU entry at the head of the RUU. If the executed bit of the instruction at the head of the RUU is set, the results of its destination register are forwarded to the register file. The associated NI counter in the register file is decremented and RUU_Head updated.

As is obvious from the above discussion, each of the tasks of the RUU can be carried out in parallel in each clock cycle and each task is simple enough that it is not likely to penalize the clock cycle. Instructions that interact with the memory are handled as in Section III-B1b. The reservation stations for the memory are provided by the RUU. Note that the load registers still need to be searched associatively for memory addresses. However, the hardware needed for this comparison is not very great for a small number of load registers. In our simulations, we used six load registers, although four were sufficient for most cases.

VI. Evaluation of the RUU

In order to evaluate the effectiveness of the RUU, we simulated three RUU organizations, 1) an RUU with bypass logic for source operand values, 2) an RUU without bypass logic, and 3) an RUU with a limited bypass logic. The results presented in this section differ from results presented previously [25]. The main reason for the difference is a different pipeline structure and a different issue mechanism for load and store instructions.

A. The RUU with Bypass Logic

Recall that the RUU forces the results to return to the registers in program order. In doing so, it aggravates data dependencies. Such a degradation could be eliminated if bypass logic for source operands was provided in some form. The simplest form could be associative comparison hardware with the destination field of each RUU entry. If a source operand for instruction I_j is provided by I_i and the destination operand of I_i is ready in the RUU, the operand can be read from the RUU and I_j is allowed to proceed with execution. Note that the history buffer and the future file [24] are alternate forms for bypass logic. The relative speedups (compared to the simple instruction issue mechanism of Table I) and the corresponding instruction issue rate for different sizes of an RUU with bypass logic are presented in Table IV.

The results of Table IV are quite promising. An RUU with a reasonable number of entries (10–12) not only speeds up execution but also provides precise interrupts. Moreover, for somewhat larger RUU sizes, the RUU is able to achieve a speedup that is quite similar to the RSTU. Note that the RSTU was not constrained to implement precise interrupts and it also requires additional associative logic.

B. The RUU without Bypass Logic

Since bypass logic is expensive to implement, we decided to evaluate an RUU without any bypass logic. Before we present

TABLE IV
RELATIVE SPEEDUP AND ISSUE RATE WITH AN RUU WITH BYPASS LOGIC

Number of Entries in RUU	Relative Speedup	Instruction Issue Rate
3	0.853	0.380
4	0.940	0.419
6	1.079	0.481
8	1.248	0.557
10	1.383	0.617
12	1.508	0.673
15	1.584	0.706
20	1.649	0.735
25	1.682	0.750
30	1.700	0.758
40	1.735	0.774
50	1.737	0.775

TABLE V
RELATIVE SPEEDUP AND ISSUE RATE WITH AN RUU WITHOUT BYPASS LOGIC

Number of Entries in RUU	Relative Speedup	Instruction Issue Rate
3	0.824	0.366
4	0.912	0.407
6	1.031	0.460
8	1.082	0.483
10	1.103	0.492
12	1.216	0.542
15	1.225	0.546
20	1.295	0.578
25	1.330	0.593
30	1.393	0.621
40	1.439	0.642
50	1.471	0.656

the results, let us see the situations where bypass logic is helpful.

Consider an instruction I_j that uses the result of a previous instruction I_i. Recall that the reservation stations associated with the RUU already have the capability to monitor the result bus. Therefore, if I_i completes execution *after* I_j is issued to the RUU, I_j can gate in the result from I_i when it appears on the result bus. In this case, bypass logic is not needed.

Bypass logic is helpful only in the cases where I_i has completed execution when I_j is issued. Rather that providing bypass logic for this case, we wait for the result of I_i to come out on the bus between the RUU and the register file in order to resolve I_j's dependency on I_i. If I_j is issued to the RUU before I_i completes, I_j's dependency on I_i can be resolved when I_i's result appears on the result bus if we extend the capabilities of the reservation stations to monitor both the result bus and the RUU to register file bus.

Table V presents the relative speedups and instruction issue rates for a RUU without bypass logic. From Table V we see that a RUU without any bypass logic at all is still able to achieve a substantial increase in speed over a simple instruction issue mechanism and implement precise interrupts at the same time. The speedup, however, is not as impressive as the speedup obtained if bypass logic were used. The difference arises mainly because of the ordering of code in the loops. Let us illustrate the problem with an example.

Consider the following section of code:

$$I_i \quad A_2 \leftarrow A1 + A3$$

$$\vdots$$

$$I_j \quad A0 \leftarrow A2 + 1$$

$$\vdots$$

$$I_k \quad \text{JAM loopstart.}$$

Conventional compilation techniques try and increase the distance between instructions I_i and I_j and instructions I_j and I_k so that when instructions I_j and I_k reach the issue stage, their respective operands are ready. Such an increase in dependency distance is in fact harmful to an RUU without bypass logic. If I_j was issued sufficiently before I_k and completed execution before I_k reached the decode and issue stage, I_k would be forced to wait until I_j left the RUU. If, on the other hand, I_j was issued soon before I_k, I_k could resolve its dependency on I_j when the result of I_j was available on the functional unit result bus. In our simulations, no attempt was made to improve the performance of the RUU without bypass logic by reordering the code for such cases.

C. The RUU with Limited Bypass Logic

Because of the problem illustrated above, we found that branch instructions were blocked for a long period of time in the decode and issue stage since the contents of the $A0$ register could not be read from the RUU (or were unavailable because of a dependency chain aggravated as above). The branch instruction has to wait in the decode and issue unit until the value of $A0$ appears on a bus. In order to eliminate this problem, we duplicated the A register file, effectively creating a limited bypass path for the A registers. The duplicate A register file acts as a future file for the A registers. The entire A register file (eight registers) was duplicated to prevent the unnecessary increase in the length of the dependency chain that affects the conditional branch instruction. All other functions are as before. Specifically, there is only 1 copy of the B, S, and T register files and there is no bypass logic in the RUU. As functions that affect the A registers are completed and appear on the result bus, the result is forwarded to the RUU and also to the A future file. The architectural register file contains a valid copy of registers at all time for recovering a precise state. Instructions that use A registers as source operands, fetch the data from the A future file, if it is available, and proceed. The results for an RUU with limited bypass logic is presented in Table VI. An RUU with limited bypass logic is able to overcome a significant portion of the performance penalty paid for eliminating bypass logic especially for small RUU sizes. For larger RUU sizes, however, the performance is not as good. This is because instructions that transfer data from a B register to an A register are still held up in the RUU (no bypass logic for the B register file). Since the destination A register of such transfer instructions eventually affects the branch condition (most branch instructions in the benchmark programs tested the value of the $A0$ register), instruction issue is blocked for longer periods of time. We are confident that the performance of an RUU without bypass logic and an RUU with limited bypass logic could be improved considerably and would come close to the speedups with bypass logic if the code was modified accordingly.

VII. BRANCH PENALTY AND CONDITIONAL INSTRUCTIONS

As mentioned earlier, the performance degradation due to branches can be reduced by conditionally executing instruc-

TABLE VI
RELATIVE SPEEDUP AND ISSUE RATE WITH AN RUU WITH LIMITED BYPASS LOGIC

Number of Entries in RUU	Relative Speedup	Instruction Issue Rate
3	0.845	0.377
4	0.930	0.415
6	1.064	0.477
8	1.119	0.499
10	1.283	0.572
12	1.310	0.584
15	1.393	0.621
20	1.457	0.650
25	1.463	0.652
30	1.484	0.662
40	1.491	0.665
50	1.525	0.680

tions from a predicted branch path. Several architectures employ this approach [3], [4], [8], [19]. To allow conditional execution of instructions, a hardware mechanism is needed that would allow the machine to recover from an incorrect branch prediction.

The RUU provides a very powerful mechanism for *nullifying* instructions, be the instructions valid instructions or instructions that executed in a conditional mode. Valid instructions may be nullified because of a trap caused by a previous instruction; conditionally executed instructions may be nullified if they are from an incorrect execution path. Therefore, the conditional execution of instructions with an RUU is very easy. If the decode and issue unit predicts the outcome of branches and actually executes instructions from a predicted path in a conditional mode, recovery from incorrect branch predictions can be achieved very easily without duplicating the register file. We can identify such instructions through the use of an additional field in the RUU and prevent them from being committed until they are proven to be from a correct path. Furthermore, there is no hard limit to the number of branches that can be predicted; the RUU can provide multiple instances of a register for the different paths. Extending the RUU to accommodate branch prediction and conditional execution is an ongoing research topic.

VIII. SUMMARY

In this paper, we have combined the issues of hardware dependency-resolution and implementation of precise interrupts. We devised a scheme that can resolve dependencies and thereby allows out-of-order instruction execution without associating tag-matching hardware with each register. Such a scheme can, therefore, be used even in the presence of a large number of registers without a substantial hardware cost. Then we extended the scheme to incorporate precise interrupts. The precise interrupt and the dependency-resolution mechanisms mutually aid and simplify each other. We evaluated the performance of the resulting hardware using 14 Livermore loops as the benchmark. The results are quite encouraging. The combined mechanism, called the RUU, is able to implement precise interrupts and is able to achieve a significant performance improvement over a simple instruction issue mechanism without a substantial cost in hardware.

ACKNOWLEDGMENT

The author would like to thank J. Goodman, A. Pleszkun, and J. Smith for their useful discussions during this research. The author would also like to thank S. Vajapeyam for his help with the simulations.

REFERENCES

[1] *CDC Cyber 200 Model 205 Computer System Hardware Reference Manual*, Control Data Corp., Arden Hills, MN, 1981.

[2] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An instruction issuing approach to enhancing performance in multiple functional unit processors," *IEEE Trans. Comput.*, vol. C-35, pp. 815–828, Sept. 1986.

[3] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine philosophy and instruction-handling," *IBM J. Res. Develop.*, pp. 8–24, Jan. 1967.

[4] P. Chow and M. Horowitz, "Architectural tradeoffs in the design of MIPS-X," in *Proc. 14th Annu. Symp. Comput. Architecture*, Pittsburgh, PA, June 1987, pp. 300–308.

[5] CRAY, *CRAY-1 Computer Systems, Hardware Reference Manual.* Chippewa Falls, WI: Cray Research, Inc., 1982.

[6] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, and J. Gill, "Hardware/software tradeoffs for increased performance," in *Proc. Int. Symp. Architectural Support Programming Languages Oper. Syst.*, Mar. 1982, pp. 2–11.

[7] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proc. 13th Annu. Symp. Comput. Architecture*, June 1986, pp. 386–395.

[8] W. Hwu and Y. N. Patt, "HPSm, A high performance restricted data flow architecture having minimal functionality," in *Proc. 13th Annu. Symp. Comput. Architecture*, June 1986, pp. 297–307.

[9] —, "Design choices for the HPSm microprocessor chip," in *Proc. 20th Annu. Hawaii Int. Conf. Syst. Sci.*, Kona, HI, Jan. 1987.

[10] —, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Trans. Comput.*, vol. C-36, pp. 1496-1514, Dec. 1987.

[11] R. M. Keller, "Look-ahead processors," *ACM Comput. Surveys*, vol. 7, pp. 66–72, Dec. 1975.

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

[13] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Comput. Mag.*, vol. 17, pp. 6–22, Jan. 1984.

[14] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proc. 13th Annu. Symp. Comput. Architecture*, Tokyo, Japan, June 1986, pp. 396–304.

[15] F. H. McMahon, *FORTRAN CPU Performance Analysis*, Lawrence Livermore Labs., 1972.

[16] N. Pang and J. E. Smith, "CRAY-1 simulation tools," Tech. Rep. ECE-83-11, Univ. of Wisconsin-Madison, Dec. 1983.

[17] Y. N. Patt, W.-M. Hwu, and M. Shebanow, "HPS, A new microarchitecture: Rationale and introduction," in *Proc. 18th Annu. Workshop Microprogramming*, Pacific Grove, CA, Dec. 1985, pp. 103–108.

[18] Y. N. Patt, S. W. Melvin, W.-M. Hwu, and M. Shebanow, "Critical issues regarding HPS, A high performance microarchitecture," in *Proc. 18th Annu. Workshop Microprogramming*, Pacific Grove, CA, Dec. 1985, pp. 109–116.

[19] A. Pleszkun, J. Goodman, W. C. Hsu, R. Joersz, G. Bier, P. Woest, and P. Schecter, "WISQ: A restartable architecture using queues," in *Proc. 14th Annu. Symp. Comput. Architecture*, Pittsburgh, PA, June 1987, pp. 290–299.

[20] A. R. Peszkun and G. S. Sohi, "The performance potential of multiple functional unit processors," in *Proc. 15th Annu. Symp. Comput. Architecture*, Honolulu, HI, June 1988, pp. 37–44.

[21] R. M. Russel, "The CRAY-1 computer system," *Commun. ACM*, vol. 21, pp. 63–72, Jan. 1978.
[22] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annu. Symp. Comput. Architecture*, May 1981, pp. 135–148.
[23] —, "Characterizing computer performance with a single number," *Commun. ACM*, vol. 31, pp. 1202–1206, Oct. 1988.
[24] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Comput.*, vol. 37, pp. 562–573, May 1988.
[25] G. S. Sohi and S. Vajapeyam, "Instruction issue logic for high-performance, interruptible pipelined processors," in *Proc. 14th Annu. Symp. Comput. Architecture*, Pittsburgh, PA, June 1987, pp. 27–34.
[26] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, pp. 25–33, Jan. 1967.
[27] S. Weiss and J. E. Smith, "Instruction issue logic in pipelined supercomputers," *IEEE Trans. Comput.*, vol. C-33, pp. 1013–1022, Nov. 1984.

Gurindar S. Sohi (S'85–M'85) received the B.E. (Hons.) degree in electrical engineering from the Birla Institute of Science and Technology, Pilani, India, in 1981 and the M.S. and Ph.D degrees in electrical engineering from the University of Illinois, Urbana-Champaign, in 1983 and 1985, respectively.

Since September 1985, he has been with the Computer Sciences Department at the University of Wisconsin–Madison where is currently an Assistant Professor. His interests are in computer architecture, parallel and distributed processing, and fault-tolerant computing.