Implementing Precise Interrupts in Pipelined Processors

JAMES E. SMITH, MEMBER, IEEE, AND ANDREW R. PLESZKUN, MEMBER, IEEE

Abstract—**This paper describes and evaluates solutions to the precise interrupt problem in pipelined processors. An interrupt is *precise* if the saved process state corresponds with a sequential model of program execution where one instruction completes before the next begins. In a pipelined processor, precise interrupts are difficult to implement because an instruction may be initiated before its predecessors have completed.**

The precise interrupt problem is described, and five solutions are discussed in detail. The first solution forces instructions to complete and modify the process state in architectural order. The other four solutions allow instructions to complete in any order, but additional hardware is used so that a precise state can be restored when an interrupt occurs. All the methods are discussed in the context of a parallel pipeline structure. Simulation results based on the CRAY-1S scalar architecture are used to show that the first solution results in a performance degradation of at least 16 percent. The remaining four solutions offer better performance, and three of them result in as little as a 3 percent performance loss. Several extensions, including vector architectures, virtual memory, and linear pipeline structures, are briefly discussed.

Index Terms—**Performance simulation, pipelined computers, precise interrupts, process checkpointing, process recovery, virtual memory.**

I. Introduction

MOST computer architectures are based on a sequential model of program execution in which an architectural program counter sequences through instructions one-by-one, finishing one before starting the next. In contrast, a high-performance implementation may be pipelined, permitting several instructions to be in some phase of execution at the same time. The use of a sequential architecture and a pipelined implementation clash at the time of an interrupt; pipelined instructions may modify the process state in an order different from that defined by the sequential architectural model. At the time an interrupt condition is detected, the hardware may not be in a state that is consistent with any specific program counter value.

When an interrupt occurs, the state of an interrupted process is typically saved by the hardware, the software, or by a combination of the two. The process state generally consists of the program counter, registers, and memory. If the saved process state is consistent with the sequential architectural model, then the interrupt is *precise*. To be more specific, the saved state should reflect the following conditions.

1) All instructions preceding the instruction indicated by the saved program counter have been executed and have modified the process state correctly.

2) All instructions following the instruction indicated by the saved program counter are unexecuted and have not modified the process state.

3) If the interrupt is caused by an exception condition raised by an instruction in the program, the saved program counter points to the interrupted instruction. The interrupted instruction may or may not have been executed, depending on the definition of the architecture and the cause of the interrupt. Whichever is the case, the interrupted instruction has either completed, or has not started execution.

If the saved process state is inconsistent with the sequential architectural model and does not satisfy the above conditions, then the interrupt is *imprecise*.

This paper describes and compares ways of implementing precise interrupts in pipelined processors. The methods used are designed to modify the state of an executing process in a carefully controlled way. The simple methods force all instructions to update the process state in the architectural order. Other, more complex methods save portions of the process state so that the proper state may be restored by the hardware at the time an interrupt occurs.

A. Classification of Interrupts

We consider interrupts belonging to two classes.

1) *Program interrupts,* sometimes referred to as "traps," result from *exception conditions* detected during fetching and execution of specific instructions. These exceptions may be due to software errors such as trying to execute an illegal opcode, numerical errors such as overflow, or they may be part of normal program execution as with page faults.

2) *External interrupts* are not caused by specific instructions and are often caused by sources outside the currently executing process, sometimes completely unrelated to it. I/O interrupts and timer interrupts are examples.

For a specific architecture, all interrupts may be defined to be precise or only a proper subset. Virtually every architecture, however, has some types of interrupts that must be precise. There are a number of conditions under which precise interrupts are either necessary or desirable.

1) For I/O and timer interrupts, a precise process state makes restarting possible.

2) In virtual memory systems, precise interrupts allow a process to be correctly restarted after a page fault has been serviced.

Manuscript received March 23, 1986. This work is supported by the National Science Foundation under Grant ECS-8207277.

J. E. Smith is with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706.

A. R. Pleszkun is with the Department of Computer Sciences, University of Wisconsin, Madison, WI 53706.

IEEE Log Number 8717725.

3) For software debugging, it is desirable for the saved state to be precise. This information can be helpful in isolating the exact instruction and circumstances that caused the exception condition.

4) For graceful recovery from arithmetic exceptions, software routines may be able to take steps, rescale floating point numbers for example, to allow a process to continue. Some end cases of modern floating point arithmetic systems might best be handled by software, gradual underflow in the proposed IEEE floating point standard [15], for example.

5) Unimplemented opcodes can be simulated by system software in a way transparent to the programmer if interrupts are precise. In this way, lower performance models of an architecture can maintain compatibility with higher performance models using extended instruction sets.

6) Virtual machines can be implemented if privileged instruction faults cause precise interrupts. Host software can simulate these instructions and return to the guest operating system in a user-transparent way.

B. Historical Survey

The precise interrupt problem is as old as the first pipelined computers [5]. The IBM 360/91 [3] was a well-known computer that produced imprecise interrupts under some circumstances, floating point exceptions, for example. Imprecise interrupts were a break with the IBM 360 architecture which made them even more noticeable. Subsequent IBM 360 and 370 implementations have used less aggressive pipeline designs where instructions modify the process state in strict program order, and interrupts are precise.[1] A more complete description of the method used in these "linear" pipeline implementations is in Section VIII-D.

Most pipelined implementations of general purpose architectures are similar to those used by IBM. These pipelines constrain all instructions to pass through the pipeline in order with a stage at the end where exception conditions are checked before the process state is modified. Examples include the Amdahl 470 and 580 [1], [2] and the Gould/SEL 32/87 [17].

The high-performance CDC 6600 [16], CDC 7600 [4], and Cray Research [8], [14] computers allow instructions to complete out of the architectural sequence. Consequently, they have some exception conditions that result in imprecise interrupts. In these machines, the advantages of precise interrupts have been sacrificed in favor of maximum parallelism and design simplicity. I/O interrupts in these machines are precise, and they do not implement virtual memory.

The CDC STAR-100 [11] and CYBER 200 [7] series machines also allow instructions to complete out of order, and they do support virtual memory. In these machines, the use of vector instructions further complicates the problem. The solution finally arrived at was the addition of an *invisible exchange package* [7]. The invisible exchange package resides in memory and captures machine-dependent state information resulting from partially completed instructions. A related approach is used in pipelined array processors [9] which contain instructions that permit interrupt handlers to explicitly dump and restore the contents of certain pipeline segments. A similar method has been suggested for MIPS [10] where pipeline information is dumped at the time of an interrupt and restored to the pipeline when the process is resumed. This solution makes a process restartable although it is arguable whether it has all the features and advantages of an architecturally precise interrupt. For example, it might be necessary to have implementation-dependent software sift through the machine-dependent state in order to provide complete debug information.

The recently announced CDC CYBER 180/990 [6] is a pipelined implementation of a new architecture that supports virtual memory, and offers roughly the same performance as a CRAY-1S. To provide precise interrupts, the CYBER 180/990 uses a history buffer, to be described later in this paper, where state information is saved just prior to being modified. When an interrupt occurs, this "history" information is used to back the system up into a precise state.

C. Paper Overview

This paper concentrates on explaining and discussing basic methods for implementing precise interrupts in pipelined processors. We emphasize scalar architectures (as opposed to vector architectures) because of their applicability to a wider range of machines. Section II defines the model architecture to be used in describing precise interrupt implementations. The model architecture is very simple so that the fundamentals of the methods can be clearly described. Sections III–VI contain methods for implementing precise interrupts. Section III describes a simple method that is easy to implement, but which reduces performance. Section IV describes a higher performance variation where results may be bypassed to other instructions before the results are used to modify the process state. Sections V and VI describe methods where instructions are allowed to complete in any order, but where state information is saved so that a precise state may be restored when an interrupt occurs. Section VII presents simulation results. Experimental results based on these CRAY-1S simulations are presented and discussed. Section VIII contains a brief discussion of 1) saving additional state information, 2) supporting virtual memory, 3) precise interrupts when a data cache is used, 4) linear pipeline structures, and 5) vector instructions. Finally, Section IX discusses ways to solve the precise interrupt problem architecturally rather than in the implementation. These methods are based on an architectural model that is parallel instead of sequential.

II. Preliminaries

A. Model Architecture

For describing the various techniques, a model architecture is chosen so that the basic methods are not obscured by details and unnecessary complications brought about by a specific architecture.

We choose a register–register architecture where all memory accesses are through registers and all functional operations involve registers. In this respect, it bears some similarity to the

[1] Except for the models 95 and 195 which were derived from the original model 91 design. Also, the models 85 and 165 had imprecise interrupts for the case of protection exceptions and addressing exceptions caused by store operations.

CDC and Cray architectures, but has only one set of registers. The load instructions are of the form $Ri = (Rj + \text{disp})$. That is, the content of register Rj plus a displacement given in the instruction are added to form an effective address. The content of the addressed memory location is loaded into register Ri. Similarly, a store is of the form $(Rj + \text{disp}) = Ri$, where register Ri is stored at the address found by adding the content of register Rj and a displacement. The functional instructions are of the form $Ri = Rj$ op Rk, where op is the operation being performed. For unary operations, the degenerate form $Ri =$ op Rk is used. Conditional instructions are of the form $P = \text{disp}:Ri$ op Rj, where P is the program counter, the disp is the address of the branch target, and op is a relational operator, $=$, $>$, $<$, etc.

The only process state in the model architecture consists of the program counter, the general purpose registers, and main memory. The architecture is simple, has a minimal amount of process state, can be easily pipelined, and can be implemented in a straightforward way with parallel functional units like the CDC and Cray architectures.

Initially, we assume no operand cache. Similarly, condition codes are not used. They add other problems beyond precise interrupts when a pipelined implementation is used. Extensions for operand cache and condition codes are discussed in Section VIII.

A parallel pipeline implementation for the simple architecture is shown in Fig. 1. It uses an instruction fetch/decode pipeline which processes instructions in order. The final stage of the fetch/decode pipeline is an issue register where all register interlock conditions are checked. If there are no register conflicts, an instruction issues to one of the parallel functional units. Here, the memory access function is implemented as one of the functional units. The operand registers are read at the time an instruction issues. There is a single result bus that returns results to the register file. This bus may be reserved at the time an instruction issues or when an instruction is approaching completion. This assumes the functional unit times are deterministic. A new instruction can issue every clock period in the absence of register or result bus conflicts. Unless stated otherwise, the parallel pipeline structure of Fig. 1 is used throughout this paper.

Example 1: To demonstrate how an imprecise process state may occur in our model architecture, consider the following section of code which sums the elements of arrays A and B into array C.

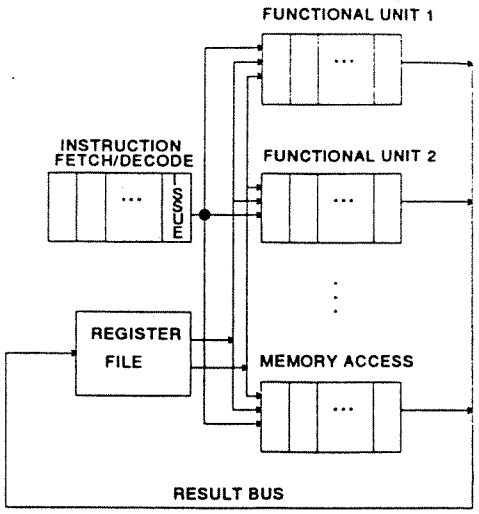


Fig. 1. Pipelined implementation of our model architecture. Not shown is the results shift register used to control the result bus.

Consider the instructions in statements 6 and 7. Although the integer add which increments the loop count will be issued after the floating point add, it will complete before the floating point add. The integer add will therefore change the process state before an overflow condition is detected in the floating point add. In the event of such an overflow, there is an imprecise interrupt.

B. Interrupts Prior to Instruction Issue

Before proceeding with the various precise interrupt methods, we first consider interrupts that occur prior to instruction issue because they are handled the same way by all the methods.

In the pipeline implementation of Fig. 1, instructions stay in sequence until the time they are issued. Furthermore, the process state is not modified by an instruction before it issues. This makes precise interrupts a simple matter when an exception condition can be detected prior to issue. Examples of such exceptions are privileged instruction faults and unimplemented instructions. This class also includes external interrupts which can be checked at the issue stage.

When such an interrupt condition is detected, instruction issuing is halted. Then, there is a wait while all previously issued instructions complete. After they have completed, the process is in a precise state, with the program counter value corresponding to the instruction being held in the issue

Statement			Comments	Execution Time
0		$R2 \leftarrow 0$	Init. loop index	
1		$R0 \leftarrow 0$	Init. loop count	
2		$R5 \leftarrow 1$	Loop inc. value	
3		$R7 \leftarrow 100$	Maximum loop count	
4	Loop:	$R1 \leftarrow (R2 + A)$	Load $A(I)$	11 clock periods
5		$R3 \leftarrow (R2 + B)$	Load $B(I)$	11 clock periods
6		$R4 \leftarrow R1 + fR3$	Floating add	6 clock periods
7		$R0 \leftarrow R0 + R5$	Inc. loop count	2 clock periods
8		$(R0 + C) \leftarrow R4$	Store $C(I)$	
9		$R2 \leftarrow R2 + R5$	Inc. loop index	2 clock periods
10		$P = \text{Loop}:R0 \mathrel{!}= R7$	cond. branch not equal	

register. The registers and main memory are in a state consistent with this program counter value.

Because exception conditions detected prior to instruction can be handled easily as described above, we will not consider them any further. Rather, we will concentrate on exception conditions detected after instruction issue.

III. In-Order Instruction Completion

With this method, instructions modify the process state only when all previously issued instructions are known to be free of exception conditions. This section describes a strategy that is most easily implemented when pipeline delays in the parallel functional units are fixed. That is, they do not depend on the operands, only on the function. Thus, the result bus can be reserved at the time of issue.

First, we consider a method commonly used to control the pipelined organization shown in Fig. 1. This method may be used regardless of whether precise interrupts are to be implemented. The precise interrupt methods described in this paper are integrated into this basic control strategy, however. To control the result bus, a "result shift register" is used; see Fig. 2. Here, the stages are labeled 1 through n, where n is the length of the longest functional unit pipeline. An instruction that takes i clock periods reserves stage i of the result shift register at the time it issues. If the stage already contains valid control information, then issue is held until the next clock period, and stage i is checked once again. An issuing instruction places control information in the result shift register. This control information identifies the functional unit that will be supplying the result and the destination register of the result. This control information is also marked "valid" with a validity bit. Each clock period, the control information is shifted down one stage toward stage one. When it reaches stage one, it is used during the next clock period to control the result bus so that the functional until result is placed in the correct result register.

Still disregarding precise interrupts, it is possible for a short instruction to be placed in the result pipeline in stage i when previously issued instructions are in stage $j, j > i$. This leads to instructions finishing out of the original program sequence. If the instruction at stage j eventually encounters an exception condition, the interrupt will be imprecise because the instruction placed in stage i will complete and modify the process state even though the sequential architecture model says i does not begin until j completes.

Example 2: If one considers the section of code presented in Example 1, and an initially empty result shift register (all the entries invalid), the floating point add would be placed in stage 6 while the integer add would be placed in stage 2. The result shift register entries shown in Fig. 2 reflect the state of the result shift register after the integer add issues. Notice that the floating point add entry is in stage 5 since one clock period has passed since it issued. As described above, this situation leads to instructions finishing out of the original program sequence.

A. Registers

To implement precise interrupts with respect to registers using the above pipeline control structure, an issuing instruction using stage j should "reserve" stages $i < j$ as well as stage j. That is, the stages $i < j$ that were not previously reserved by other instructions are reserved, and they are loaded with null control information so that they do not affect the process state. This guarantees that instructions modifying registers finish in order.

STAGE	FUNCTIONAL UNIT SOURCE	DESTN. REGISTER	VALID	PROGRAM COUNTER
1			0	
2	INTEGER ADD	0	1	7
3			0	
4			0	
5	FLT PT ADD	4	1	6
⋮	⋮	⋮	⋮	⋮
N			0	

DIRECTION OF MOVEMENT

Fig. 2. Result shift register.

There is logic on the result bus that checks for exception conditions in instructions as they complete. If an instruction contains a nonmasked exception condition, then control logic "cancels" all subsequent instructions coming on the result bus so that they do not modify the process state.

Example 3: For our sample section of code given in Example 1, assuming the the result shift register is initially empty, such a policy would have the floating point add instruction reserve stages 1–6 of the result shift register. When, on the next clock cycle, the integer add is in the issue register, it is prohibited from issuing because stage 2 is already reserved. Thus, the integer add must wait at the issue stage until stage 2 of the result shift register is no longer reserved. This would be five clock periods after the issue of the floating point add.

A generalization of this method is to determine, if possible, that an instruction is free of exception conditions prior to the time it is complete. Only result shift register stages that will finish before exceptions are detected need to be reserved (in addition to the stage that actually controls the result).

B. Main Memory

Store instructions modify the portion of process state that resides in main memory. To implement precise interrupts with respect to memory, one solution is to force store instructions to wait for the result shift register to be empty before allowing them to issue. Alternatively, stores can issue and be held in the load/store pipeline until all preceding instructions are known to be exception-free. Then the store can be released to memory.

To implement the second alternative, recall that memory can be treated as a special functional unit. Thus, as with any other instruction, the store can make an entry in the result shift register. This entry is defined as a *dummy* store. The dummy store does not cause a result to be placed in the register file, but is used for controlling the memory pipeline. The dummy store is placed in the result shift register so that it will not reach stage one until the store is known to be exception-free. When the dummy stores stage one, all previous instructions have completed without exceptions, and a signal is sent to the load/store unit to release the store to memory. If the store itself contains an exception condition, then the store is cancelled, all following load/store instructions are cancelled, and the store

unit signals the pipeline control so that all instructions issued subsequent to the store are cancelled as they leave the result pipeline.

C. Program Counter

To implement precise interrupts with respect to the program counter, the result shift register is widened to include a field for the program counter of each instruction (see Fig. 2). This field is filled as the instruction issues. When an instruction with an exception condition appears at the result bus, its program counter is available and becomes part of the saved state.

IV. The Reorder Buffer

The primary disadvantage of the above method is that fast instructions may sometimes get held up at the issue register even though they have no dependencies and would otherwise issue. In addition, they block the issue register while slower instructions behind them could conceivably issue.

This leads us to a more complex, but more general solution. Instructions are allowed to finish out of order, but a special buffer called the *reorder buffer* is used to reorder them before they modify the process state.

A. Basic Method

The overall organization is shown in Fig. 3(a). The reorder buffer, Fig. 3(b), is a circular buffer with head and tail pointers. Entries between the head and tail are considered valid. When an instruction issues, the next available reorder buffer entry, pointed to by the tail pointer, is given to the issuing instruction. The tail pointer value is used as a tag to identify the entry in the buffer reserved for the instruction. The tag is placed in the result shift register along with the other control information. The tail pointer is then incremented, modulo the buffer size. The result shift register differs from the one used earlier because there is a field containing a reorder tag instead of a field specifying a destination register.

When an instruction completes, both results and exception conditions are sent to the reorder buffer. The tag from the result shift register is used to guide them to the correct reorder buffer entry. When the entry at the head of the reorder buffer contains valid results (its instruction has finished), then its exceptions are checked. If there are none, the results are written into the registers. If an exception is detected, issue is stopped in preparation for the interrupt, and all further writes into the register file are inhibited.

Example 4: The entries in the reorder buffer and result shift register shown in Fig. 3(b) reflect their state after the integer add from Example 2 has issued. Notice that the result shift register entries are very similar to those in the Fig. 2. The integer add will complete execution before the floating point add and its results will be placed in entry 5 of the reorder buffer. These results, however, will not be written into $R0$ until the floating point result, found in entry 4, has been placed in $R4$.

B. Main Memory

Preciseness with respect to memory is maintained in a manner similar to that in the in-order completion scheme

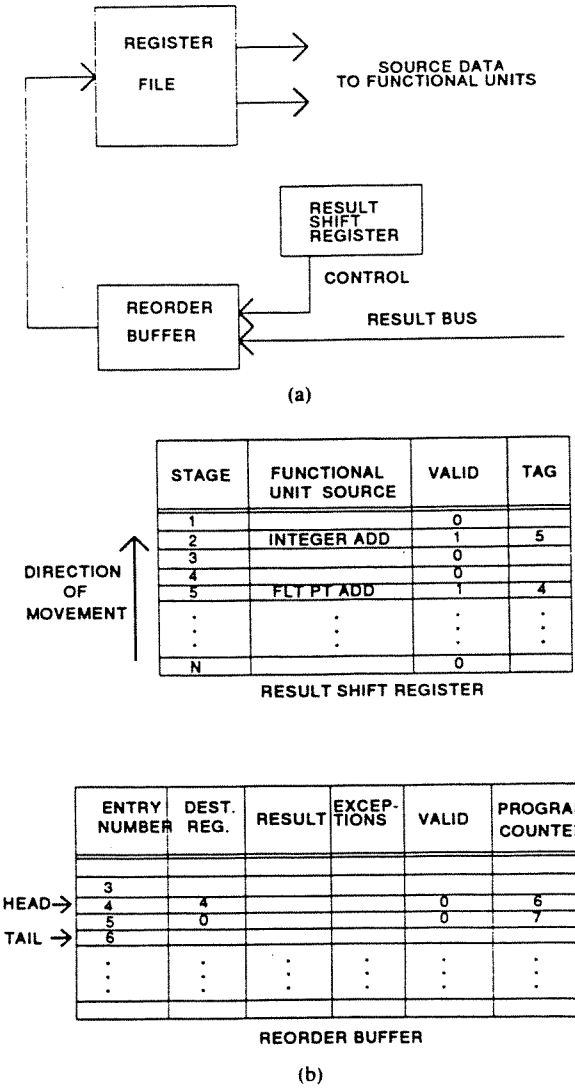


Fig. 3. (a) Reorder buffer organization. (b) Reorder buffer and associated result shift register.

(Section III-B). The simplest method holds stores in the issue register until all previous instructions are known to be free of exceptions. In the more complex method, a store signal is sent to the memory pipeline as a "dummy" store is removed from the reorder buffer. Stores are allowed to issue, and block in the store pipeline prior to being committed to memory while they wait for their dummy counterpart.

C. Program Counter

To maintain preciseness with respect to the program counter, the program counter can be sent to a reserved space in the reorder buffer at issue time [shown in Fig. 3(b)]. While the program counter could be sent to the result shift register, it is expected that the result shift register will contain more stages than the reorder buffer and thus require more hardware. The length of the result shift register must be as long as the longest pipeline stage; as will be seen in Section VII, the number of entries in the reorder buffer can be quite small. When an instruction arrives at the head of the reorder buffer with an exception condition, the program counter found in the reorder buffer entry becomes part of the saved precise state.

D. Bypass Paths

While an improvement over the method described in Section III, the reorder buffer still suffers a performance penalty. A computed result that is generated out of order is held in the reorder buffer until previous instructions, finishing later, have updated the register file. An instruction dependent on a result being held in the reorder buffer cannot issue until the result has been written into the register file.

The reorder buffer method may, however, be modified to minimize some of the drawbacks of finishing strictly in order. In order for results to be used early, bypass paths may be provided from the entries in the reorder buffer to the register file output latches, see Fig. 4. These paths allow data being held in the reorder buffer to be used in place of register data. The implementation of this method requires comparators for each reorder buffer stage and operand designator. If an operand register designator of an instruction being checked for issue matches a register designator in the reorder buffer, then a multiplexer is set to gate the data from the reorder buffer to the register output latch. In the absence of other issue blockage conditions, the instruction is allowed to issue, and the data from the reorder data are used prior to being written into the register file.

There may be bypass paths from some or all of the reorder buffer entries. If multiple bypass paths exist, it is possible for more than one destination entry in the reorder buffer to correspond to a single register. Clearly only the *latest* reorder buffer entry that corresponds to an operand designator should generate a bypass path to the register output latch. To prevent multiple bypassing of the same register, when an instruction is placed in the reorder buffer, any entries with the same destination register designator must be inhibited from matching a bypass check.

When bypass paths are added, preciseness with respect to the memory and the program counter does not change from the previous method.

The greatest disadvantage with this method is the number of bypass comparators needed and the amount of circuitry required for the multiple bypass check. While this circuitry is conceptually simple, there is a great deal of it.

V. History Buffer

The methods presented in this section and the next are intended to reduce or eliminate performance losses experienced with a simple reorder buffer, but without all the control logic needed for multiple bypass paths. Primarily, these methods place computed results in a working register file, but retain enough state information so a precise state can be restored if an exception occurs.

Fig. 5(a) illustrates the history buffer method. The history buffer is organized in a manner very similar to the reorder buffer. When an instruction issues, a buffer entry is loaded with control information, as with the reorder buffer, but the current value of the destination register (to be overwritten by the issuing instruction) is also read from the register file and written into the buffer entry. Results on the result bus are written directly into the register file when an instruction completes. Exception reports come back as an instruction

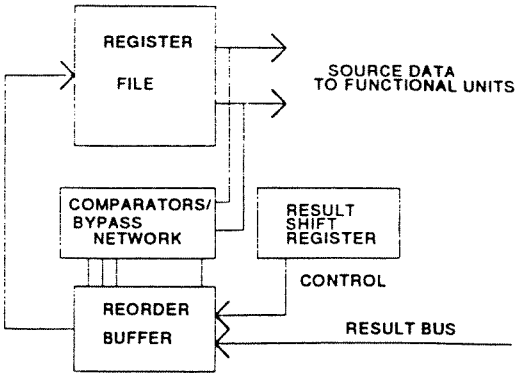
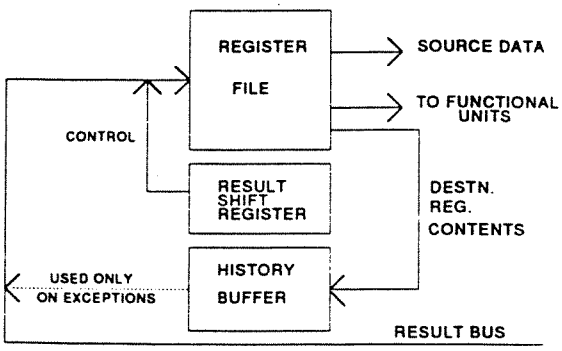
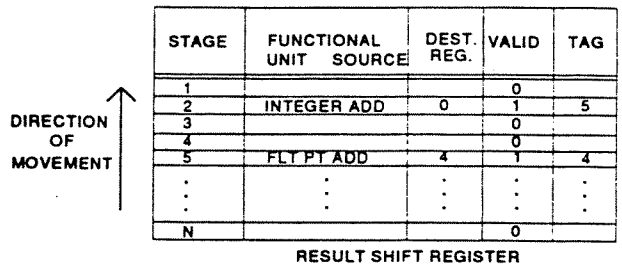
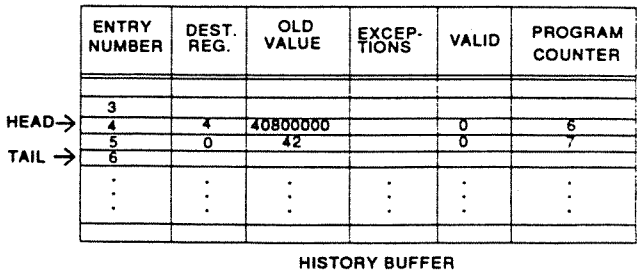


Fig. 4. Reorder buffer method with bypasses.



(a)





(b)

Fig. 5. (a) History buffer organization. (b) History buffer and associated result shift register.

completes and are written into the history buffer. As with the reorder buffer, the exception reports are guided to the proper history buffer entry through the use of tags found in the result shift register. When the history buffer contains an element at the head that is known to have finished without exceptions, the history buffer entry is no longer needed and that buffer location can be reused (the head pointer is incremented). As with the reorder buffer, the history buffer can be shorter than the maximum number of pipeline stages. If all history buffer entries are used (the buffer is too small), issue must be blocked

until an entry becomes available. Hence, the buffer should be long enough so that this seldom happens. The effect of the history buffer on performance is determined in Section VII.

Example 5: The entries in the history buffer and result shift register shown Fig. 5(b) correspond to our code in Example 1, after the integer add has issued. The only differences between this and the reorder buffer method shown in Fig. 3(b) are the addition of an "old value" field in the history buffer and a "destination register" field in the result shift register. The result shift register now looks like the one shown in Fig. 2.

When an exception condition arrives at the head of the buffer, the buffer is held, instruction issue is immediately halted, and there is a wait until pipeline activity completes. The active buffer entries are then emptied from tail to head, and the history values are loaded back into their original registers. The program counter value found in the head of the history buffer is the precise program counter.

To make main memory precise, when a store entry emerges from the buffer, it sends a signal that another store can be committed to memory. Stores can either wait in the issue register or can be blocked in the memory pipeline, as in the previous methods.

The extra hardware required by this method is in the form of a large buffer to contain the history information. Also the register file must have three read ports since the destination value as well as the source operands must be read at issue time. There is a slight problem if the basic implementation has a bypass of the result bus around the register file. In such a case, the bypass must also be connected into the history buffer.

VI. Future File

The future file method (Fig. 6) is similar to the history buffer method; however, it uses two separate register files. One register file reflects the state of the architectural (sequential) machine. This file will be referred to as the *architectural file.* A second register file is updated as soon as instructions finish and therefore runs ahead of the architectural file (i.e., it reflects the future with respect to the architectural file). This *future file* is the working file used for computation by the functional units.

Instructions are issued and results are returned to the future file in any order, just as in the original pipeline model. There is also a reorder buffer that receives results at the same time they are written into the future file. When the head pointer finds a completed instruction (a valid entry), the result associated with that entry is written in the architectural file.

Example 6: If we consider the code in Example 1 again, there is a period of time when the architecture file and the future file contain different entries. With this method, an instruction may finish out of order, so when the integer add finishes, the future file contains the new contents of $R0$. The architecture file, however, does not, and the new contents of $R0$ are buffered in the reorder buffer entry corresponding to the integer add. Between the time the integer add finishes and the time the floating point add finishes, the two files are different. Once the floating point finishes and its results are written into $R4$ of both files, $R0$ of the architecture file is written.

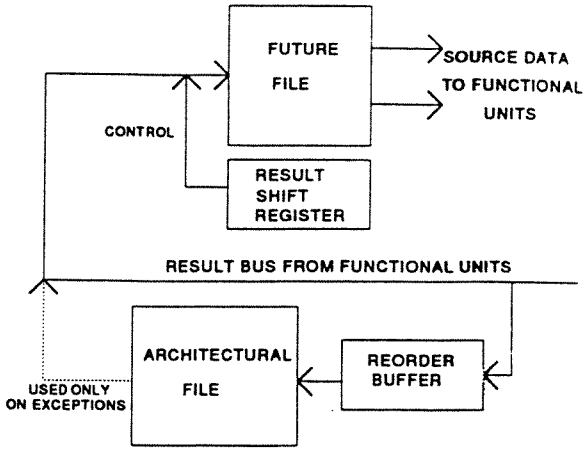


Fig. 6. Future file organization.

Just as with the pure reorder buffer method, program counter values are written into the reorder buffer at issue time. When the instruction at the head of the reorder buffer has completed without error, its result is placed in the architectural file. If it completes with an error, the register designators associated with the buffer entries between the head and tail pointers are used to restore values in the future file from the architectural file.[2]

The primary advantage of the future file method is realized when the architecture implements interrupts via an "exchange" where all the registers are automatically saved in memory and new ones are restored (as is done in CDC and Cray architectures). In this case, the architectural file can be stored away immediately; no restoring is necessary as in the history buffer method. There is also no bypass problem as with the history buffer method.

VII. Performance Evaluation

To evaluate the effectiveness of our precise interrupt schemes, we use a CRAY-1S simulation system developed at the University of Wisconsin [13]. This trace-driven simulator is extremely accurate, due to the highly deterministic nature of the CRAY-1S, and gives the number of clock periods required to execute a program.

The scalar portion of the CRAY-1S is very similar to the model architecture described in Section II-A. Thus, casting the basic approaches into the CRAY-1S scalar architecture is straightforward.

For a simulation workload, the first 14 Lawrence Livermore Loops [12] were used. Because we are primarily interested in pipelined implementations of conventional scalar architectures, the loops were compiled by the Cray Fortran compiler with the vectorizer turned off.

In the preceding sections, five methods were described that could be used for guaranteeing precise interrupts. To evaluate the effect of these methods on system performance, the methods were partitioned into three groups. The first and second group, respectively, contain the in-order method and the simple reorder buffer method. The third group is com-

[2] The restoration is performed from the architectural file since the future file is the register file from which all execution takes place.

posed of the reorder buffer with bypasses, the history buffer, and the future file. This partitioning was performed because the methods in the third group result in identical system performance. This is because the future file has a reorder buffer embedded as part of its implementation, and the history buffer length constrains performance in the same way as a reorder buffer: when the buffer fills, issue must stop. All the simulation results are reported as for the reorder buffer with bypasses. They apply equally well for the history buffer and future file methods. The selection of a particular method depends not only on its effect on system performance but also the cost of implementation and the ease with which the precise CPU state can be restored.

For each precise interrupt method, both methods for handling stores were simulated. For those methods other than the in-order completion method, the size of the reorder buffer is a parameter. Sizing the buffer with too few entries degrades performance since instructions that might issue could block at the issue register. The blockage occurs because there is no room for a new entry in the buffer.

Table I shows the relative performance of the in-order, reorder buffer, and reorder buffer with bypass methods when the stores are held until the result shift register is empty. The results in the table indicate the relative performance of these methods with respect to the CRAY-1S across the first 14 Lawrence Livermore Loops; real CRAY-1S performance is 1.0. A relative performance greater than 1.0 indicates a degradation in performance. The number of entries in the reorder buffer varies from 3 to 10.

The simulation results for in-order completion are constant because this method does not depend on a buffer that reorders instructions. For all the methods, there is some performance degradation. Initially, when the reorder buffer is small, the in-order completion method produces the least performance degradation. A small reorder buffer (less than three entries) limits the number of instructions that can simultaneously be in some stage of execution. Once the reorder buffer size is increased beyond three entries, either of the other methods results in better performance. As expected, the reorder buffer with bypasses offers superior performance when compared to the simple reorder buffer. When the size of the buffer was increased beyond ten entries, simulation results indicated no further performance improvements. (Simulations were also run for buffer sizes of 15, 16, 20, 25, and 60.) One can expect at least 12 percent performance degradation when using a reorder buffer with bypasses and the first method for handling stores.

Table II indicates the relative performance when stores issue and wait at the same memory pipeline stage as for memory bank conflicts in the original CRAY-1S. After issuing, stores wait for their counterpart dummy store to signal that all previously issued register instructions have finished. Subsequent loads and stores are blocked from issuing.

As in Table I, the in-order completion results are constant across all entries. For the simple reorder buffer, the buffer must have at least five entries before it results in better performance than in-order completion. The reorder buffer with bypasses, however, requires only four entries before it is performing more effectively than with in-order completion. Just as in Table I, having more than eight entries in the reorder buffer does not result in improved performance. Comparing Tables I and II, the second method for handling stores offers a clear improvement over the first method. If the second method is used with an eight-entry reorder buffer that has bypasses, a performance degradation of only 3 percent is experienced.

TABLE I
RELATIVE PERFORMANCE FOR THE FIRST 14 LAWRENCE LIVERMORE LOOPS, WITH STORES BLOCKED UNTIL THE RESULTS PIPELINE IS EMPTY

Number of Entries	In-order Completion	Reorder Buffer	Reorder with Bypasses
3	1.2322	1.3315	1.3069
4	1.2322	1.2183	1.1743
5	1.2322	1.1954	1.1439
8	1.2322	1.1808	1.1208
10	1.2332	1.1808	1.1208

TABLE II
RELATIVE PERFORMANCE FOR THE FIRST 14 LAWRENCE LIVERMORE LOOPS, WITH STORES HELD IN THE MEMORY PIPELINE AFTER ISSUE

Number of Entries	In-order Completion	Reorder Buffer	Reorder with Bypasses
3	1.1560	1.3058	1.2797
4	1.1560	1.1724	1.1152
5	1.1560	1.1348	1.0539
8	1.1560	1.1167	1.0279
10	1.1560	1.1167	1.0279

Clearly there is a tradeoff between performance degradation and the cost of implementing a method. For very little cost, the in-order completion method can be combined with the first method of handling stores. Selecting this "cheap" approach results in a 23 percent performance degradation. If this degradation is too great, either the second store method must be used with in-order completion or one of the more complex methods must be used. If the reorder buffer method is used, one should use a buffer with at least three or four entries.

It is important to note that in the performance study just described, some indirect causes for performance degradation were not considered. These include longer control paths that would tend to lengthen the clock period. Also, additional logic for supporting precise interrupts implies greater board area which implies more wiring delays which could also lengthen the clock period.

VIII. Extensions

In previous sections, we described methods that could be used to guarantee precise interrupts with respect to the registers, the main memory, and the program counter of our simple architectural model. In the following sections, we extend the previous methods to handle additional state information, virtual memory, cache memory, linear pipelines, and vectors.

A. Handling Other State Values

Most architectures have more state information than we have assumed in the model architecture. For example, a

process may have state registers that point to page and segment tables, indicate interrupt mask conditions, etc. This additional state information can be precisely maintained with a method similar to that used for stores to memory. If using a reorder buffer, an instruction that changes a state register reserves a reorder buffer entry and proceeds to the part of the machine where the state change will be made. The instruction then waits there until receiving a signal to proceed from the reorder buffer. When its entry arrives at the head of the buffer and is removed, then the signal is sent to cause the state change.

In architectures that use condition codes, the condition codes are state information. Although the problem condition codes present to conditional branches is not totally unrelated to the topic here, solutions to the branch problem are not the primary topic of this paper. Hence, it is assumed that the conditional branch problem has been solved in some way, e.g., [3]. If a reorder buffer is being used, condition codes can be placed in the reorder buffer. That is, just as for data, the reorder buffer is made sufficiently wide to hold the condition codes. The condition code entry is then updated when the condition codes associated with the execution of an instruction are computed. Just as with data in the reorder buffer, a condition code entry is not used to change processor state until all previous instructions have completed without error (however, condition codes can be bypassed to the instruction fetch unit to speed up conditional branches).

Extension of the history buffer and future file methods to handle condition codes is very similar to that of the reorder buffer. For the history buffer, the condition code settings at the time of instruction issue must be saved in the history buffer. The saved condition codes can then be used to restore the processor state when an exception is detected.

B. Virtual Memory

Virtual memory is a very important reason for supporting precise interrupts; it must be possible to recover from page faults. First, the address translation section of the pipeline should be designed so that all the load/store instructions pass through it in order. In-order memory operations have been assumed throughout this paper. Depending on the method being used, the load/store instructions reserve time slots in the result pipeline and/or reorder buffer that are read no earlier than the time at which the instructions have been checked for exception conditions (especially page faults). For stores, these entries are not used for data; just for exception reporting and/or holding a program counter value.

If there is an addressing fault, then the instruction is cancelled in the addressing pipeline, and all subsequent load/store instructions are cancelled as they pass through the addressing pipeline. This guarantees that no additional loads or stores modify the process state. The mechanisms described in the earlier sections for assuring preciseness with respect to registers guarantee that nonload/store instructions following the faulting load/store will not modify the process state; hence, the interrupt is precise.

For example, if the reorder buffer method is being used, a page fault would be sent to the reorder buffer when it is detected. The tag assigned to the corresponding load/store instruction guides it to the correct reorder buffer entry. The reorder buffer entry is removed from the buffer when it reaches the head. The exception condition in the entry causes all further entries of the reorder buffer to be discarded so that the process state is modified no further (no more registers are written). The program counter found in the reorder buffer entry is precise with respect to the fault.

C. Cache Memory

Thus far, we have assumed systems that do not use a cache memory. Inclusion of a cache in the memory hierarchy affects the implementation of precise interrupts. As we have seen, an important part of all the methods is that stores are held until all previous instructions are known to be exception-free. With a cache, stores may be made into the cache earlier, and for performance reasons should be. The actual updating of main memory, however, is still subject to the same constraints as before.

1) Store-Through Caches: With a store-through cache, the cache can be updated immediately, while the store-through to main memory is handled as in previous sections. That is, all previous instructions must first be known to be exception-free. Load instructions are free to use the cached copy, however, regardless of whether the store-through has taken place. This means that main memory is always in a precise state, but the cache contents may "run ahead" of the precise state. If an interrupt should occur while the cache is potentially in such a state, then the cache should be flushed. This guarantees that prematurely updated cache locations will not be used. However, this can lead to performance problems, especially for larger caches.

An alternative is to treat the cache in a way similar to the register files. One could, for example, keep a history buffer for the cache. Just as with registers, a cache location would have to be read just prior to writing it with a new value. This does not necessarily mean a performance penalty because the cache must be checked for a hit prior to the write cycle. In many high-performance cache organizations, the read cycle for the history data could be done in parallel with the hit check. Each store instruction makes a buffer entry indicating the cache location it has written. The buffer entries can be used to restore the state of the cache. As instructions complete without exceptions, the buffer entries are discarded. The future file can be extended in a similar way.

2) Write-Back Cache: A write-back cache is perhaps the cache type most compatible with implementing precise interrupts. This is because stores in a write-back cache are not made directly to memory; there is a built-in delay between updating the cache and updating main memory. Before an actual write-back operation can be performed, however, the reorder buffer should be emptied or should be checked for data belonging to the line being written back. If such data should be found, the write-back must wait until the data have been stored in the cache. If a history buffer is used, either a cache line must be saved in the history buffer, or the write-back must wait until the associated instruction has made its way to the end of the buffer. Notice that in any case, the write-back will sometimes have to wait until a precise state is reached.

D. Linear Pipeline Structures

An alternative to the parallel functional unit organizations we have been discussing is a linear pipeline organization. Refer to Fig. 7. Linear pipelines provide a more natural implementation of register–storage architectures like the IBM 370. Here, the same instruction can access a memory operand and perform some function on it. Hence, these linear pipelines have an instruction fetch/decode phase, an operand fetch phase, and an execution phase, any of which may be composed of one or several pipeline stages.

In general, reordering instructions after execution is not as significant an issue in such organizations because it is natural for instructions to stay in order as they pass through the pipe. Even if they finish early in the pipe, they proceed to the end where exceptions are checked before modifying the process state. Hence, the pipeline itself acts as a sort of reorder buffer.

The role of the result shift register is played by the control information that flows down the pipeline alongside the data path. Program counter values for preciseness may also flow down the pipeline so that they are available should an exception arise.

Linear pipelines often have several bypass paths connecting intermediate pipeline stages. A complete set of bypasses is typically not used, rather there is some critical subset selected to maximize performance while keeping control complexity manageable. Hence, using the terminology of this paper, linear pipelines typically achieve precise interrupts by using a reorder buffer method with bypasses.

E. Vectors

Implementing precise interrupts in a pipelined vector architecture is more difficult than for a scalar architecture. In this section, we consider extensions of our previous methods to vectors.

When considering precise interrupts with respect to vector instructions, preciseness must be carefully defined. Unlike the scalar instructions described thus far, vector instructions do not produce a single result and change the system state as they complete. Rather, they produce a series of results that change the system state over the course of many clock periods. The sequential architecture model, as applied to vectors, requires that one vector instruction completes its last result before the next begins producing results. Furthermore, requirement 3) for precise interrupts implies that a vector instruction must either complete in the presence of an exception condition, or it must be made to look as if it has not started. This implies that some buffering of vector results may be required in a pipelined implementation regardless of the method used for implementing precise interrupts.

There are two primary classes of vector architectures: those with vector registers, and those with memory-to-memory vector operations. For vector register architectures, we extend our earlier methods for maintaining scalar registers precisely. For memory-to-memory architectures, the second method for handling scalar stores to memory is extended.

1) Register Architectures: In-order completion (our first method) as extended to vectors implies that one instruction is finished producing results before the next begins. This implies

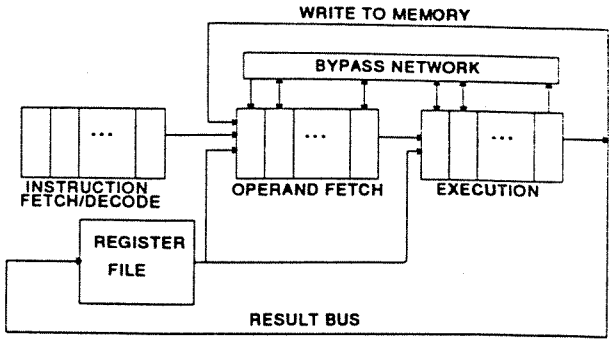


Fig. 7. Example of a linear pipeline implementation.

no overlap of vectors with scalars, and no vector chaining. This can be implemented in instruction issue logic by blocking issue as long as a vector instruction is in progress.

There still remains the problem of interrupts that occur in the middle of the execution of a vector instruction. If the interrupt is an external interrupt, it is simply a matter of waiting until the instruction completes. For many types of program interrupts (e.g., page faults), however, it may not be possible to allow the instruction to complete, so it must be backed up. A simple solution is to buffer results and write them into a vector register after all results are complete, but this leads to performance problems. A vector instruction using the results must wait for the copy from the buffer to the register to complete. A better method is to have two copies of each vector register. A 1-bit pointer for each register indicates the "current" copy. When a vector operation is initiated, vector results are placed in the "new" (noncurrent) copy. When the vector instruction is complete, then the "current" pointer can be changed to the new copy. If there is a fault, then the pointer is not updated, and the register copy saved from before the vector instruction started remains the current copy.

A reorder buffer method can be used to permit limited overlap of scalar and vector operation. Due to the length of many vector operations, however, the buffer would have to be very long, or relatively little scalar operation would be possible. Also, unless bypasses are used, all the scalar instructions would have to be independent of each other. This would limit the usefulness of this method even more. For vectors, complete buffering of results would still be needed, but without bypasses, chaining would not be possible.

A practical method based on reorder buffers is to save scalars in the buffer as before, but save vector register pointers, rather than the vectors themselves, in the buffer. There would, again, be two copies of each vector register: a current copy and a new copy. A pointer to a specific vector register could appear in the buffer only once. This pointer would indicate not only a vector register, but which of the pair contains the new results. As the pointer is removed from the buffer, the "current" pointer for the vector would be updated. If a fault is detected, the pointer is not updated, so the old copy of the vector register is kept. This method overlaps dependent scalar operations with vectors, and chaining can be implemented by bypassing from the new copy pointed to by vector pointer in the buffer, rather than the current copy.

We now give an example of the method just described.

...e following two instructions: $V1 = V2 + V3$; $V3 = V4$. Assume at the beginning of the sequence, the ...nter for $V1$ is 0. After the first vector instruction is ... vector designator $V1$ and 1 (the complement of its ...nter) are placed in the reorder buffer, and vector ...placed into copy 1 of $V1$ as they are generated. The ...lues in copy 0 of $V1$ are retained. The second ... can begin and chain to the first; here copy 1 of $V1$ ... the chain. The chain is set up using the pointer ...e reorder buffer. After the first instruction is known ...plete without errors, the current pointer for $V1$ is ... 1. Else if there is an error, it is kept at 0. Note that ... $V1$ can only be reused after the first instruction has ...

...e history file method, the size of buffers is again a ...blem; the file would be written as vector results are ... vector chaining would imply multiple results being ... simultaneously so that the file would have to be able ...rt multiple simultaneous writes. Similarly, scalar ...uld be produced simultaneously with vector results ...ey would probably need their own history file with ...age between the two. A better method is to once ...pairs of vector registers. When a vector instruction is ... makes an entry in the history file indicating the ...gister copy to be used as a backup. If a backup is ...he "current" pointer is adjusted to achieve the ...s with the other methods, using pairs of vector ...equires that only one instruction that uses a specific ...ister can be active in the system at any given time; ...of the pair is used for new results, the other is used ... old results.

... time it should be clear that extending the future file ...r vector registers can also easily be done with pairs ...registers in a manner very similar to the history file ...st described.

...arize, the various methods for implementing ...errupts can be extended for vector registers, but the ...ubling of the number of hardware registers plus ...itional control hardware to keep track of the ... pointers.

...ory-to-Memory Vectors: In the case of a memory-... vector architecture, it may be necessary to buffer ... the CPU until all the operations associated with a ...truction are completed without exception before ...emory stores to begin. This is done in the most ...ward way by extending store method 2 described ... it may require a much larger store buffer. This is ... used in the CDC CYBER 180/990 where vectors ...n up to 512 elements. In some architectures which ... vector lengths, the size of this buffer may be ..., however.

IX. Architectural Solutions

... we have assumed a sequential architectural model ...on, and have attempted to work around it for ...implementations. However, the root cause of the ...e architectural model. Hence, it seems reasonable ...e problem at the architectural level. That is, one might be able to define an architectural model where an interrupted state assumes an underlying parallel implementation. In this section, we briefly discuss a few such architectural solutions.

One architectural solution is to "freeze" the pipeline when an interrupt is detected, and simply "dump" the state of all the registers in the pipeline to memory as part of a saved context. Then, to restart the process, all the pipeline registers are restored, and the pipeline is started. Although this leads to processes that may be restarted, this approach has some disadvantages. One disadvantage is that freezing a pipe is difficult in practice due to fan-out problems. The fan-out problem comes from the need to control overwriting every pipeline stage. If writing all the stages is conditional on a single signal, for example a signal indicating no interrupt conditions, then the "no interrupt" condition must be fanned out to all the flip flops in all the registers in the pipeline. The large fan-out required for such a signal can lengthen the critical path of the pipeline control. Because of the tremendous fan-out for larger pipelined systems, this method may only be useful for the very smallest pipelined systems [10]. Such an approach also means that implementation details, for example the number of pipeline stages, become part of the architecture. This could lead to compatibility problems.

A variation of the above method, directed at vectors, is to define the vector architecture so that at the time of the interrupt, the intermediate state of vector instructions is saved. This information is primarily in the form of length counters. If done properly, a vector instruction is stopped in the middle, and restarting upon returning from the interrupt is accomplished by reissuing the stopped vector instruction.

Another architectural solution to the precise interrupt problem is to save a series of program counter values, ending with a final program counter value that is much like the one in the sequential model. Each program counter points to an instruction, prior to the final one, that has not been executed. To restart a process, the instructions pointed to by the series program counters must first be executed before the machine is in a precise state with respect to the final one.

Example 7: Again, we consider the code shown in Example 1 and the case where the floating point add overflows after the loop increment in statement 7 has completed. Using an architecture as defined above, in the event of floating point overflow in statement 6, the program counter pointing to statement 6 could be saved along with the program counter pointing to statement 8. A program counter for statement 7 is not needed because it has successfully completed. The overflow handler could "fix up" the overflow, possibly by rescaling, and then return to the process. Then using both saved program counters, the processor would execute statement 6 before resuming regular execution with statement 8.

Finally, one could save a sequence of instructions that must be executed before the saved program counter is precise. This has the advantage of fetching the instruction sequence as part of the context being restored. The program counter method requires first fetching the program counters as part of the restored context, then fetching the instructions themselves. In the example just given, the floating point add instruction itself

would be saved, not its program counter. Then, as part of the return from the interrupt the processor would execute the floating point add before fetching instructions beginning with statement 8.

X. Summary and Conclusions

Five methods for implementing precise interrupts in the pipelined processors were described. These methods were then evaluated through simulations of a CRAY-1S implemented with these methods.

The first method forces in-order instruction completion, and our simulation study indicates a performance degradation of about 23 percent when store instructions are held in the instruction issue register and about 16 percent when stores are held in the memory pipeline. Performance is lost primarily because of added instruction issue blockages not related to data dependencies. The significant performance difference due to the way stores are handled is noteworthy.

To improve the performance provided by the first method, a reorder buffer is proposed to permit instructions to complete out of order, but to reorder the results going into the register file. For a reorder buffer of size eight, this method results in performance loss of 18 percent or 12 percent, depending on the way stores to memory are handled. Here performance is lost because results cannot be used because they are being held in the reorder buffer prior to result register update.

The third method studied adds bypass paths to the reorder buffer permitting data to be used prior to the result register update. With this method, performance loss is cut to 12 percent or 3 percent, again depending on the handling of stores. These final results indicate that performance losses can be significantly reduced, but only if stores are blocked in the memory pipeline to wait for previous instructions to complete.

The final two methods, the history buffer and future file methods, permit alternative implementations that give the same performance as with the reorder buffer using bypasses. The implementation differences among the final three methods are relatively minor, and any final choice should be based on technology related issues affecting implementation cost and complexity.

There are many other interesting issues related to implementing precise interrupts. These include the handling of virtual memory faults, caches, vectors, and alternative pipeline structures. Although they were briefly touched on in this paper, they deserve further research.

Finally, the basic concepts of process interruptability and restartability should be studied extensively. We feel that methods of saving and restoring state which do not rely on a serial model of execution are essential to the development of parallel general purpose systems.

Acknowledgment

The authors would like to thank R. G. Hintz and J. B. Pearson of the Control Data Corporation.

References

[1] Amdahl Corp., *Amdahl 470V/8 Computing System Machine Reference Manual*, pub. G1014.0-03A, Oct. 1981.
[2] ——, "580 Technical Introduction," 1980.
[3] D. W. Anderson, F. J. Sparacio, and F. M. Tomasulo, "The IBM system/360 Model 91: Machine philosophy and instruction-handling," *IBM J. Res. Develop.*, vol. 11, pp. 8–24, Jan. 1967.
[4] P. Bonseigneur, "Description of the 7600 computer system," *Computer Group News*, pp. 11–15, May 1969.
[5] W. Bucholz, Ed., *Planning a Computer System*. New York: McGraw-Hill, 1962.
[6] Control Data Corp., *CDC Cyber 180 Computer System Model 990 Hardware Reference Manual*, pub. 60462090, 1984.
[7] ——, *CDC CYBER 200 Model 205 Computer System Hardware Reference Manual*, Arden Hills, MN, 1981.
[8] Cray Research, Inc., *CRAY-1 Computer Systems, Hardware Reference Manual*, Chippewa Falls, WI, 1979.
[9] Floating Point Systems, *FPS-100 Programmers Reference Manual*, Beaverton, OR, 1980.
[10] J. Hennessy *et al.*, "Hardware/software tradeoffs for increased performance," in *Proc. Symp. Architectural Support Programming Languages Oper. Syst.*, Apr. 1982, pp. 2–11.
[11] R. G. Hintz and D. P. Tate, "Control data STAR-100 processor design," in *Proc. COMPCON 72, IEEE Comput. Soc. Conf. Proc.*, Sept. 1972, pp. 1–4.
[12] F. H. McMahon, "FORTRAN CPU performance analysis," Lawrence Livermore Labs., 1972.
[13] N. Pang and J. E. Smith, "CRAY-1 simulation tools," Tech. Rep. ECE-83-11, Univ. Wisconsin-Madison, Dec. 1983.
[14] R. M. Russell, "The CRAY-1 computer system," *Commun. ACM*, vol. 21, pp. 63–72, Jan. 1978.
[15] D. Stevenson, "A proposed standard for binary floating point arithmetic," *Computer*, vol. 14, pp. 51–62, Mar. 1981.
[16] J. E. Thornton, *Design of a Computer—The Control Data 6600*. Glenview, IL: Scott, Foresman, 1970.
[17] W. P. Ward, "Minicomputer blasts through 4 million instructions a second," *Electron.*, pp. 155–159, Jan. 13, 1982.

James E. Smith (S'74–M'76) received the B.S., M.S., and Ph.D. degrees from the University of Illinois in 1972, 1974, and 1976, respectively.

Since 1976, he has been on the faculty of the University of Wisconsin, Madison where he is an Associate Professor in the Department of Electrical and Computer Engineering. He spent the summer of 1978 with the IBM Thomas J. Watson Research Center, and from September 1979 until July 1981 he worked for the Control Data Corporation, Arden Hills, MN. While at CDC he participated in the design of the CYBER 180/990. He is currently on leave from the University of Wisconsin, working for the Astronautics Corporation of America on the design of a large scale scientific computer system.

Andrew R. Pleszkun (S'82–M'82) received the B.S. degree in electrical engineering from the Illinois Institute of Technology, Chicago, in 1977 and the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois, Urbana, in 1979 and 1982, respectively.

He is an Assistant Professor in Computer Sciences Department at the University of Wisconsin, Madison. His research interests include computer architecture, with an emphasis on pipelined systems, and the impact of VLSI on computer architecture.