OUT-OF-ORDER COMMIT LOGIC WITH PRECISE EXCEPTION HANDLING FOR PIPELINED PROCESSORS

Balaji Vijayan, Manivel Rajendran and Senthilkumar Veluswami
{balaji, rmanivel, senthil}@cs.annauniv.edu
School of Computer Science and Engineering
College of Engineering
Anna University, Chennai
India-600 025

ABSTRACT

The major bottleneck in today's pipelined microprocessors has been data dependencies and branch prediction. Existing approaches have focused on issuing instructions out-of-order but have limited themselves to committing in-order considering the amount of overhead involved. This paper proposes an architecture in which instructions are allowed to commit out-of-order thereby increasing the throughput. The problem of precise exception handling in out-of-order commit has also been handled without involving significant hardware overhead. The design proposed in this paper can also be extended in future to incorporate branch prediction.

1. INTRODUCTION

As the demand for processing power increases, computer system designers are forced to use techniques that result in high-performance processing units. A widely used technique is pipelining [1], in which the overall logic of the system is split into several stages with each stage performing a sub-task of a complete task. Considerable overlap can be achieved because each stage can perform a sub-task for a different task. Pipelined CPUs have two major impediments to their performance: i) data dependencies and ii) branch instructions. An instruction cannot begin execution until its operands are available. If an operand is the result of a previous instruction, the instruction must wait till the previous instruction has completed execution, thereby decreasing throughput. The performance degradation due to branch instructions is even more severe. Not only must a conditional branch instruction wait for its condition to be known (resulting in "bubbles" in the pipeline), an additional penalty is incurred in fetching an instruction from the taken branch path to the instruction decode and issue stage.

Also, a major problem that arises in pipelined computer design is that an interrupt can be imprecise [2, 3]. This problem is especially severe in multiple functional unit computers in which instructions can complete execution out of program order [4]. 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-degrading factors.

In this paper, the issues of dependency-resolution and preciseness of state are combined. This paper proposes a design wherein instructions commit out-of-order and, at the same time, guarantees a precise state of the machine, without a significant hardware overhead.

The paper is organized as follows. Section 2 discusses the previous work done in the area of data dependencies. Section 3 explains the proposed design. Section 4 describes the simulation setup. Section 5 discusses the results of simulation. Section 6 concludes the paper and points out future directions of research.

2. PREVIOUS WORK

Previous work in the area of data dependencies was based on Tomasulo Algorithm which supported out-of-order execution of instructions. The drawbacks of the Tomasulo Algorithm led to significant work on simplifying the hardware and on out-of-order issue of instructions.

2.1. Tomasulo's Algorithm

The algorithm works as follows: An instruction, whose operands are not available when it enters decode and issue stage is forwarded to a Reservation Station (RS) [5] associated with the functional unit that it will be using. It waits in the RS until its data dependencies have been resolved, i.e., its operands are ready. An instruction waiting at the RS resolves its dependencies by monitoring the Common Data Bus (Result Bus). 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 [6] or before it is about to leave the functional unit [7].

Each source register is assigned a bit that determines if the register is busy. A register is busy if it is the destination of an instruction that is still in execution. A destination register is also called a sink register [7]. Each sink register is assigned a tag which identifies the result that must be written into the register. Each reservation station has the following fields:

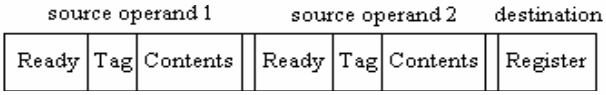

Figure 1: RS entry

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. If the sink register is busy, the instruction fetches a new tag, updates the tag of the sink register and proceeds to a reservation station. The registers as well as the reservation stations monitor the result bus and update their contents when a matching tag is found.

The RS allows instructions to finish execution out of order but updates the state of the machine (registers, memory, etc.), i.e., commits the instructions in the order that the instructions were present in the program, thereby assuring that a precise state of the machine is recoverable at any time.

2.2. Extension to Tomasulo's Approach

All the reservation stations were combined into a common RS Pool called Register Update Unit (RUU) [7] rather than having disjointed pools of reservation stations associated with each functional unit. The RUU along with the Register Status Unit (RSU) helps in implementing precise interrupt handler. The RSU indicates whether the register is a destination for an instruction in execution and contains the corresponding tag for the register. The RUU is implemented as a circular queue of entries, each of which consists of the details of instructions in the order in which they were dispatched.

- The instructions are issued from the RUU to the functional units if the operands are ready – *Out-of-order issue*.

- As instructions finish execution they can leave the RUU (i.e. commit) only if they are at the top of the RUU. If they are not at the top they have to wait until all instructions before it commit i.e. though the result is available the registers cannot be updated – *In-order completion*.

The proposed design to commit instructions in out-of-order with exception handling is discussed in the next section.

3. PROPOSED DESIGN

For an instruction to commit out-of-order, the corresponding RUU entry has to be removed and the other entries should be shifted. This involves a lot of hardware considering the amount of information to be shifted each time. To overcome this problem, a new unit called Order Unit (OU) is used. When an instruction gets committed before its preceding instructions, only the entries in OU are shifted.

In case of an exception, the old values should be recovered. A new unit called Register Update Buffer (RUB) is used to handle exceptions.

3.1. Register Update Unit (RUU)

Certain modifications are to be carried out in the RUU described in the previous section to implement this design. In order to recover from an exception, the previous value should be known. So a new field that contains the result tag is used. It specifies the RUU entry that has the previous value of the same destination register. Also, a busy bit is used to identify whether the RUU entry is available or busy.

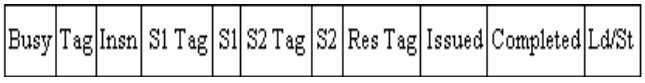

Figure 2: RUU entry

As an instruction is dispatched, the register status of the destination register is also checked. If it is 'busy' then the tag is sent to the 'Result Tag' field of the RUU entry.

3.2. Order Unit (OU)

The OU contains an array of pointers to the instructions in the RUU and maintains the program order among the remaining instructions.

- When an instruction is dispatched, the instruction details are sent to RUU and the tag of the RUU entry is appended to the OU.
- When an instruction commits out-of-order, the subsequent entries are shifted only in the OU.

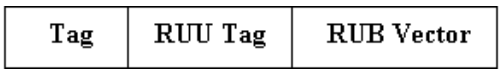

Figure 3: OU entry

In Figure 3, 'Tag' ranges from 0 to (n-1) as in the case of RUU. 'RUU Tag' is the tag of the corresponding RUU entry. 'RUB Vector' corresponds to the entries in RUB that are associated with the instruction. It consists of 1's and 0's. A '1' in n^{th} position means that the n^{th} entry in RUB is associated with the instruction. It is used to recover from an exception.

The shifting of entries in case of an out-of-order commit in OU reduces the overhead involved.

3.3. Register Update Buffer (RUB)

When an instruction commits, an entry is made in the RUB so that in case of an exception, old values can be recovered. While committing, the old value and the register number are entered as a new entry in the RUB and the corresponding bit of the 'RUB vector' of the immediately preceding instruction in the OU is set. The 'RUB vector' of the immediately preceding instruction in OU is updated by ORing it with the 'RUB vector' of the committing instruction.

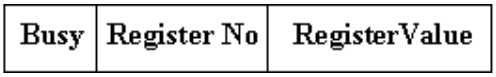

Figure 4: RUB entry

An instruction can commit out-of-order if

- there is a free entry in RUB or
- there is an entry for the same register in RUB corresponding to the immediately preceding instruction. In this case, the old entry is squashed i.e. the value in RUB is updated.

When the instruction pertaining to the first entry in OU commits, the corresponding entries in RUB are squashed, as there is no need to recover these values on an exception.

3.4. Dependency Resolution

3.4.1. RAW
This hazard is handled as in Tomasulo's Algorithm.

3.4.2. WAW
In order to recover from exceptions, writes to registers are done only in program order. So writes to same register get committed only in order and there is no WAW hazard.

3.4.3. WAR
To overcome this problem, the source operands are read as soon as they are available. If an instruction is waiting for an operand it monitors the result bus and if there is a match the operand value is read.

3.4.4. Memory Operations
The memory operations are issued in-order because the addresses of consecutive loads/stores may point to the same memory location resulting in wrong execution. Also, the memory operations are committed only in-order.

3.4.5. Branch instruction
Branch instructions are not predicted. The instructions following the branch instruction are not dispatched to the RUU, until the result of the branch is known. When the target of a branch is found, fetching is continued from the new Program Counter (PC) location.

3.4.6. Exceptions
When an instruction causes an exception, the registers may have a wrong value as the instructions are committed out-of-order. To recover the original values, the instructions in OU are considered backwards till the instruction that caused exception since recovery should be from the most recent value. The corresponding RUB entries contain the original values of those registers that were updated by the succeeding instructions that committed out of order.

After recovering the original values, all the entries from the instruction that caused the exception to the last entry in the OU are flushed. The corresponding RUU entry, RUB entries and functional units are also flushed. All the other remaining instructions are committed and the control is given to the exception handler.

The memory needs no recovery as writes to memory are issued in-order and only when the instruction is at the top of the OU. If memory operations are committed out-of-order then recovery will decrease throughput as memory accesses require long latencies.

4. SIMULATION

The simulations were carried out using the DLXView simulator [8]. The simulation programs had dependencies involving all the three hazards. All the three hazards were detected and resolved resulting in higher throughput. Exceptions were also handled properly, thus guaranteeing precise state of the machine. The results are discussed in the next section.

5. RESULTS AND DISCUSSIONS

In the existing model, the instructions wait in the RUU even after they complete execution out of order. Because of this, the issue of new instructions gets stalled. But in the proposed model, the entries in the RUU are squashed as soon as the instructions complete execution.

This model performs better when the load and store latencies are high or when there are high latency instructions. In the existing model, the instruction following load/store or any high latency instruction cannot commit or execute until these preceding instructions finish. But, in this model, if instructions are not dependent on the loads or high latency instruction, they can execute and commit. As a result, throughput increases.

Also on decreasing the RUU size, this model works better as it accommodates more instructions in a given time.

6. CONCLUSION

Thus, the issue of dependency resolution for out-of-order with exception handling has been discussed in detail. A scheme was designed that can resolve dependencies and thereby allow out-of-order instruction execution and out-of-order commit. This scheme also handles exceptions without involving a significant hardware overhead. This design can be extended to support execution of instructions from a predicted branch path. In case of a misprediction, the exception handling mechanism can be used for recovery.

REFERENCES

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

2. D.W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "*The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling*", IBM Journal of Research and Development, pp. 8-24, January 1967.

3. J.E. Smith and A. R. Pleszkun, "*Implementation of Precise Interrupts in Pipelined Processors*", Proceedings of the 12th Annual Symposium on Computer Architecture, pp. 36-44, June 1985.

4. R. M Russel, "*The CRAY-1 Computer System*", CACM, vol. 21, pp. 63-72, January 1978.
5. Gurindar S. Sohi, Sriram Vajapeyam, "*Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors*", ISCA 1987.
6. S. Weiss and J. E. Smith, "*Instruction Issue Logic for Pipelined Supercomputers*", Proceedings of the 11th Annual Symposium on Computer Architecture, pp. 110-118, June 1984.

7. R. M. Tomasulo, "*An Efficient Algorithm for Exploiting Multiple Arithmetic Units*", IBM Journal of Research and Development, pp. 25-33, January 1967.

8. Larry B. Hostetler and Brian Mirtich, "*DLXSim – A Simulator for DLX*", Technical Report, January 1998.