p793

1999

IAP

Comparative Architectures / IAP Q1 Extended Answer

Why might a processor supporting 'dynamic execution' (out-of-order execution with speculation) yield better performance than a statically-scheduled processor with a similar number of execution units? [5 marks]

Dynamic execution processors will typically achieve a lower average CPI (cycles per instruction) than statically scheduled ones.

Rather than stalling on the first data-dependency or structural hazard that occurs in the instruction stream, dynamic execution processors continue fetching past this point in order to try and find other non-dependent instructions to execute in the meantime.

Internally, the processor uses data-flow driven execution, where instructions are held until all their source operands are available, and then executed.

Speculation enables the processors to fetch instructions across basic block boundaries, before the true outcome of the branch or jump is known. Prediction units that record the past behaviour of jumps and branches are used to guide which paths the processor should speculate down. Although some statically scheduled processors employ speculation, the technique is essential to out-of-order processors to enable them to search many instructions into the future to find other work to do.

Statically scheduled processors rely on the compiler to organise the code such that instructions are grouped so that the number of data and structural hazards are avoided. A dynamically scheduled processor does this at run-time, and can thus take advantage of the dynamic behaviour of the program that could not be known to a compiler, thus exploiting greater Instruction Level Parallelism. Furthermore, the dynamic execution processor is likely to be less sensitive to code quality than the statically scheduled processor, and hence more forgiving of code optimized for older processor versions.

The ability to search ahead in the instruction stream tends to make dynamic execution processors more tolerant to greater L1 D-cache latencies (and other functional units with longer latency e.g. FP divide). The processor may even find enough non-dependent work to sometimes hide the effects of L1 D-cache misses (current processors are typically unable to exploit enough ILP to avoid stalling during an L2 miss).

Outline the operation of a dynamic execution processor. You may wish to address the following issues in your answer:

- *register re-use (name dependencies)*
- *arithmetic exceptions*
- *control-flow misprediction*
- *load/store ordering preservation*

[10 marks]

Dynamic execution processors consist of three main units: fetch/decode, dispatch/execute and retire.

The fetch unit works in program order, using control-flow speculation to fetch and decode a dense stream of program instructions ready for execution. Register renaming is typically employed, whereby architecturally visible registers are renamed to a pool of internal registers, thus avoiding stalls due to 'name dependencies' where no data is carried between references to a register.

Every instruction that results in assignment to a destination register will have that register renamed to a new internal register. The register mapping unit keeps track of what the current internal names of architectural registers are, and will remap the instruction's source operands accordingly.

Name dependencies can result when the compiler chooses to reuse a register to hold a different variable (after live variable analysis), or if the processor is speculatively executing multiple iterations of a loop concurrently.

After being decoded, instructions are placed in a re-ordering buffer, where they wait until all the values of their source operands are known. When both operands are ready, the instruction is eligible for dispatch to a suitable function unit. Current processors have up to 4 integer units and 2 FP units. Each unit may be slightly specialised. For example, only one integer unit may be equipped with a divider, so the dispatcher must take this into account.

The algorithm for selecting which instructions to dispatch typically favours instructions that have been resident in the re-order buffer for longer periods of time, as executing these is likely to help progress of the retire unit. At the end of execution, the instruction's result value is made available for all other instructions that are waiting for it as a source operand.

The retire unit removes completed instructions from the re-order buffer in program order, recycles internal registers that are no longer needed, and updates architecturally visible state. (the strict in-order retire implemented by current processors is actually more conservative than necessary – future processors may relax this a little).

The retire unit ensures that instructions that were executed speculatively can not cause changes to the architecturally visible state until it is known that it is safe to do so.

When a branch or jump misprediction is detected, the processor will be forced to backtrack and begin fetching down the correct path. The backtracking will typically require junking all instructions in the re-order buffer that are from after the point when the misprediction occurred, and reverting to the register remapping state that was valid at that time. Exceptions are handled similarly, causing state to be backed off to the point where the exception occurred so that the handler can be dispatched with the program state appearing identical to that which would have been produced if the program had been executed serially.

Because of the necessity of having all the backtracking logic to cope with speculation, the handling of exceptions that occur late in the execution pipe (e.g. FP overflow) is not as painful as it is with statically-scheduled processors.

Although load instructions can be issued speculatively, store instructions change architectural state, and can thus not be issued until the retire stage when it is known that their execution is actually warranted. Thus, care must be taken to ensure that loads that refer to the same location as a store can not overtake the store (otherwise they'd read the location's old value rather than the new). However, the location referred to in the store may not be resolved until long after its insertion into the re-order buffer; it may be waiting due to a data dependency. Issuing loads as early as possible is usually good for performance, as it can hide some L1 cache misses. If the addresses referenced by all outstanding stores are not known when other loads are issued, the processor must be prepared to backtrack and 'replay' the load if a conflict is detected later.

What factors are likely to affect the amount of instruction-level parallelism that will be exploited by future processors? [5 marks]

In order to exploit greater ILP, future processors will have to:

a) search many instructions ahead into the instruction stream. Current processors e.g. the 21264 can search upto 80 instructions, future processors may need to search hundred of instructions ahead, storing state on all those fetched, and retaining the ability to be able to back-track to any one of them that could generate an exception or mispredict.

b) employ many physical registers into which architectural registers can be renamed to avoid name-dependence stalls due to register reuse by the compiler, or due to loops. The number of rename registers required is typically related to the size of the search window.

c) be surrounded by high-performance memory hierarchies that minimize the number of cycles that are lost due to cache misses causing the instruction search window to max-out.

d) perform very accurate control-flow prediction if the large instruction search window is to contain useful work.

e) perform accurate memory reference alias analysis to spot loads that should not be allowed to bypass stores (and thus require replaying). This is hard since the actual store address may not be known until too late.

Many of the mechanisms required to implement these functions are gate-intensive structures that are not easy to scale. Since they are typically on the critical path, scaling them to exploit greater ILP may result in increased cycle time.