Cheap Out-of-Order Execution using Delayed Issue

J.P. Grossman
Dept. of EECS, MIT
jpg@ai.mit.edu

Abstract

Out-of-order issue mechanisms increase performance by dynamically rescheduling instructions that cannot be statically reordered by the compiler. Such mechanisms are effective but expensive in terms of both complexity and silicon area. It is therefore desirable to find cost-effective alternatives which can provide similar performance gains.

In this paper we present Delayed Issue, a novel technique which allows instructions to be executed out-of-order without the hardware complexity of dynamic out-of-order issue. Instructions are inserted into per-functional unit delay queues using delays specified by the compiler. Instructions within a queue are issued in order; out of order execution results from different instructions being inserted into the queues at various delays. In addition to improving performance, delayed issue reduces code bloat when loops are pipelined.

1. Introduction

The order in which instructions are executed on a superscalar architecture can have a dramatic impact on performance. It is often impossible to optimally schedule instructions at compile time due to variable latencies or complicated program flow graphs; out-of-order issue mechanisms address this problem by allowing instructions to be safely reordered at run time. For example, consider the program flow across two basic blocks depicted in Figure 1a, and suppose that for some reason the compiler is unable to optimize across the boundary between them. Ignoring branches, four assembly instructions are generated (Figure 1b). Assuming a pipelined architecture that performs floating point addition and multiplication in three cycles, these instructions will take 10 cycles to execute when issued in-order (Figure 1c) and 7 cycles when issued out-of-order (Figure 1d).

Out-of-order issue is effective, but it is also costly. If the architecture maintains a window of M instructions waiting to be issued and is capable of generating N results per cycle, then the hardware complexity of out-of-order issue is at least O(MN) as each result may be an operand for any of the M instructions. This is acceptable for scalar architectures in which the goal is to use all available silicon resources to produce the fastest (and most complicated) uniprocessor possible. However, when the focus shifts to parallel architectures in which multiple processors are placed on a single die, overall area efficiency becomes more important than the raw speed of any individual processor. In this case it becomes desirable to find cost effective alternatives that can achieve similar performance with less area overhead.

2. Delayed Issue

A key observation is that while the compiler is often unable to fill holes in the pipeline, it can predict where these holes will occur. For example, in the code sequence shown in Figure 1b, the compiler can easily figure out that the second instruction cannot be issued until three cycles after the first one due to the data dependency (t1). A delayed issue mechanism allows the compiler to communicate this information to the hardware by specifying these delays explicitly as part of the instruction.

Assume that on each cycle the hardware can decode a group of instructions, specified by the compiler, where there is at most one instruction in a group destined for each functional unit. Using explicit delays, we can rewrite the code sequence of Figure 1b as follows:

```
1   FMUL t1, y, z;  FADD w, x, t1@3
2   FMUL a, t2, d@3; FADD t2, b, c
```

where we use the notation op@N to denote an operation which is delayed for N cycles. To implement these delays in hardware we introduce per-functional unit delay queues. When a group of instructions is decoded, the
instructions are inserted into the corresponding queues using the specified delays. This is depicted in Figure 2, which shows the first few cycles of execution for the above instructions. The order of execution is the same as for full out-of-order issue (Figure 1d), and the execution time is therefore the same (7 cycles).

2.1. Rationale

It may at first seem counter-intuitive that delaying instructions can expedite program execution. The reason that delayed issue works is that it allows the hardware to make better use of its instruction issue logic. In a dynamic out-of-order issue processor, this resource is replicated as many times as there are instructions waiting to be issued so that any of these instructions can be issued on any cycle. In an in-order processor, the logic is not replicated, and so if an instruction stalls due to a data dependency it ties up the resource until the dependency is resolved. On cycles in which it can be statically determined that the instruction will stall, this is a waste of silicon. Delayed issue avoids this wastage; placing such instructions in delay queues keeps the issue mechanism available for instructions that can actually use it.

2.2. Inserting Instruction Groups

The above example was simple in that it was possible to insert each instruction at the exact delay specified by the compiler. In general this is not the case; a delay slot may already be occupied, or a data hazard with some other instruction already in one of the queues may require the delay to be increased in order to avoid program errors. Given an instruction group $I_1@d_1; I_2@d_2; \ldots ; I_M@d_M$, the challenge of delayed issue is to find delays $d'_j \geq d_j$ such that instruction $I_j$ may be inserted in the $j^{th}$ queue at depth $d'_j$ without altering the semantics of the program. This is the dominant source of complexity in an implementation of delayed issue as it requires keeping track in hardware of which registers are used by which instructions in the queues.

2.3. Software Pipelining

In addition to improving performance when the compiler is unable to schedule across basic block boundaries, delayed issue allows the compiler to produce more compact code when it pipelines loops. As an example, consider the simple loop shown in Figure 3a. Assuming the existence of a hardware looping construct, Figure 3b gives the equivalent assembly code. If integer multiplication takes 2 cycles, integer and pointer addition take 1 cycle, and the data resides in the cache so that memory references take 1 cycle, we can apply software pipelining to the loop with an initiation interval of two (Figure 3c).

Note that the number of instructions has increased dramatically from 6 to 18 due to the prologue (PR) and epilogue (EP). The reason these are necessary is that there are several instances of instructions A, B such that A precedes B within each iteration, but in order to pipeline the loop B must be scheduled closer to the start of the loop kernel than A. Without delayed issue, the only way to achieve this schedule is to actually place B earlier than A in the kernel assembly code. We therefore need a prologue containing A to precede B the first time the loop is entered, and we need an epilogue containing B to succeed A when the loop is exited (Figure 4a). Using delayed issue, however, we can place B after A in the loop and specify a delay. This has the effect of scheduling B closer to the start of the loop when the loopback branch is taken, and after the loop when the loop is exited. We can...
therefore eliminate the prologue and epilogue code (figure 4b). In figure 4c we have rewritten the loop in 3c using delayed issue; the number of instructions has dropped from 18 back down to 6.

3. Related Work

Out-of-order execution was introduced in 1967 by R. M. Tomasulo as part of the design of the IBM System/360 Model 91 [Tomasulo67]. Since then, much effort has been directed at either reducing the complexity of out-of-order issue without sacrificing performance, or developing alternate architectures and compiler technologies which achieve similar performance at a lower hardware cost. A full summary of these efforts is beyond the scope of this paper, but we will briefly mention some previous work related to the use of FIFO queues for instruction issue.

Very Long Instruction Word (VLIW) architectures [Fisher83] are gaining in popularity as a way of obtaining high performance with relatively simple hardware. In a pure VLIW machine, all responsibility for scheduling instructions and avoiding data hazards is placed on the compiler, which results in extremely simple issue logic. In [Rau93] the VLIW model was extended by splitting instructions into two phases corresponding to initiation and completion of the original instruction, with the second phase being placed in a delay buffer to execute at the correct time. This allows VLIW code with strong timing assumptions to execute correctly in the presence of variable latencies; in particular, it allows dynamic scheduling techniques to be implemented in a VLIW machine.

An excellent presentation of ways to simplify superscalar architectures without seriously impacting performance is given in [Palacharla97]. Their central proposal is to replace the issue window with a small number of FIFO buffers, where an attempt is made to direct co-dependent instructions into the same buffer. As with delayed issue, the issue logic only needs to monitor the heads of the queues as opposed to the entire issue window, and is therefore greatly simplified. Unlike delayed issue, an instruction from a given queue can issue to any functional unit, which leads to greater complexity and wiring requirements. It should be noted that, in the absence of delayed issue, per-functional unit FIFO queues are not effective; in [Butler92] this arrangement was found to perform consistently and significantly worse than all other out-of-order strategies considered.

4. Conclusion

In domains for which area efficiency is of primary concern, delayed issue may be a cost-effective way to improve performance via out-of-order execution. While the mechanisms required to correctly implement delayed issue are not trivial, our estimates indicate that they are less expensive than dynamic out-of-order issue hardware. Delayed issue has the added benefit of allowing the compiler to produce more compact pipelined loops.

The ideas presented in this paper are preliminary and should therefore be read with an appropriate amount of scepticism. In order to validate delayed issue as a useful microarchitectural technique, further work is required to:

1. Determine in simulation the exact performance gains afforded by delayed issue
2. Compare the hardware complexity of a delayed issue processor with that of an out-of-order issue processor

This is the subject of future research.

Acknowledgements

Thanks to Tom Knight, Jeremy Brown, Andrew Huang, Shana Nichols and the anonymous reviewers for their careful reading and helpful suggestions. This work was sponsored by DARPA contract MDA972-97-C-0025 and Air Force Research Laboratory, Rome Labs, agreement number F30602-98-1-0172.

References